UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

VISUALIZING DATA IN TRADITIONAL TEXT LAYOUTS WITH APPLICATION TO CLASSICAL LATIN

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

JEYACHANDRAN RATHNAM
Norman, Oklahoma
2017

VISUALIZING DATA IN TRADITIONAL TEXT LAYOUTS
WITH APPLICATION TO CLASSICAL LATIN

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Chris Weaver, Chair

Dr. Christan Grant

Dr. Dean Hougen

Acknowledgements

I would first like to thank my advisor Dr. Chris Weaver, for providing the resources and freedom to explore and try new things in the area of data visualization. Without his help and guidance, none of this would have been possible. I would also like to thank my committee members, Dr. Christan Grant and Dr. Dean Hougen for their time and effort.

A special thanks to everyone in the Digital Latin Library project for all the work that has been done to support this research.

To my friends here in the US and back home in India, thank you all the memorable fun times we spent together. Without you all, this journey would have been boring.

To all the faculty members in Computer Science and OU in general, thank you for always being there whenever I needed help.

I also would like to thank my parents, my sister and their family for their continued support and motivation throughout these years.

Contents

List of Figures

Abstract

Scholarship in linguistics, library science, journalism, and classics has long depended upon reading and analyzing text in traditional printed formats. Increasingly, manual and automatic text processing techniques are being used to transform text into structured data that can be queried and visualized. Such data often consists of spreadsheets or databases that contain information extracted from the text, such as document metadata, entities and relationships, sentiment, or grammatical structure. Many techniques have been developed to visualize text as structured data. A few display that data in a recognizably traditional format. Representing data within its own text provides a common visual context that allows for more efficient reading and effective interpretation.

In this thesis, we describe a design framework and a reference implementation for visually representing information in traditional text layouts. The framework provides ways to visually represent information about text directly in the text itself, including text styling, highlighting, decoration, and embedding of entire data views. By integrating data graphics into text, readers can see and interact with both the text and its associated data in a unified visual context. The implementation builds upon the declarative language in Improvise to support interactive queries including visual encoding, filtering, and sorting of data for display in the text. We present several application examples and use them to

assess the expressiveness, effectiveness, and performance of the framework for navigation and query features desirable to scholars of classical Latin. From the assessment, we conclude that the framework supports cognitive tasks such as reading, problem solving, and decision making.

Chapter 1

Introduction

Classic scholars primarily use printed documents to read and analyze text. Text analysts use manual and automatic text processing techniques to transform text into structured data. Manual processes might involve reading through the text and recording individual data items. Automatic processes can involve algorithms to convert text to data based on certain conditions. For example, on occurrence of a specific word, it is converted to a new data item. The data can contain metadata, entities and relationships, sentiment, or grammatical structure. The processed data can be stored in various data storing formats such as XML or databases for easier access. This processed data can be visualized to look for patterns and anomalies.

A variety of visualization techniques have been proposed. Poemage [16] visualizes words that are similar sounding. Tag clouds [23] visualize word frequency. TexTile [3, 4] visualizes textual variants. However, few visualization techniques use traditional text format to visualize text together with its associated data. When text is converted to data, it is often represented as a collection of individual entities, rather than as a stream of interlinked words. *Text is context*

sensitive. Maintaining familiar reading format and continuity allows for efficient reading. Embedding data along with the text allows for effective interpretation of the data.

One of the areas of research that works with textual data is Digital Humanities. Digital Humanities scholars who specialize in classic Latin texts aim to create critical editions. Classic Latin texts were composed in antiquity. The original texts are often partially or completely lost. Most of the texts we see today are copies of supposed originals. Historically, copying of text was manual, which was prone to errors and often introduced different meanings from the original text. Scholars prefer to see and compare multiple versions of text at the same time to draw conclusions from it. Analyzing large documents by manually comparing different versions of the same text takes a lot of effort and time.

In this thesis, we have designed and implemented a framework for designing interactive visualizations that use traditional prose text layouts to display text and its associated data. An example application of this framework is shown in Figure 1.1. The data set for this figure consists of random sentences generated online[1]. The framework supports the display of scrolling, wrapping lines of text with embedded data graphics. Visual encodings of embedded data include text styling, highlighting, decorating, and embedding of entire data views. This combination of features provides design flexibility to represent a variety of data attributes. Additional information such as titles, line numbers, and speakers can be included in the layout to give context to the entire document.

Background work in data visualization, related work in text visualization, features that inspired the design of the new text visualization framework, and current tools available for variant text analysis are discussed in **Chapter 2**.

[1] `https://randomwordgenerator.com/sentence.php` Accessed: 2017-07-31.

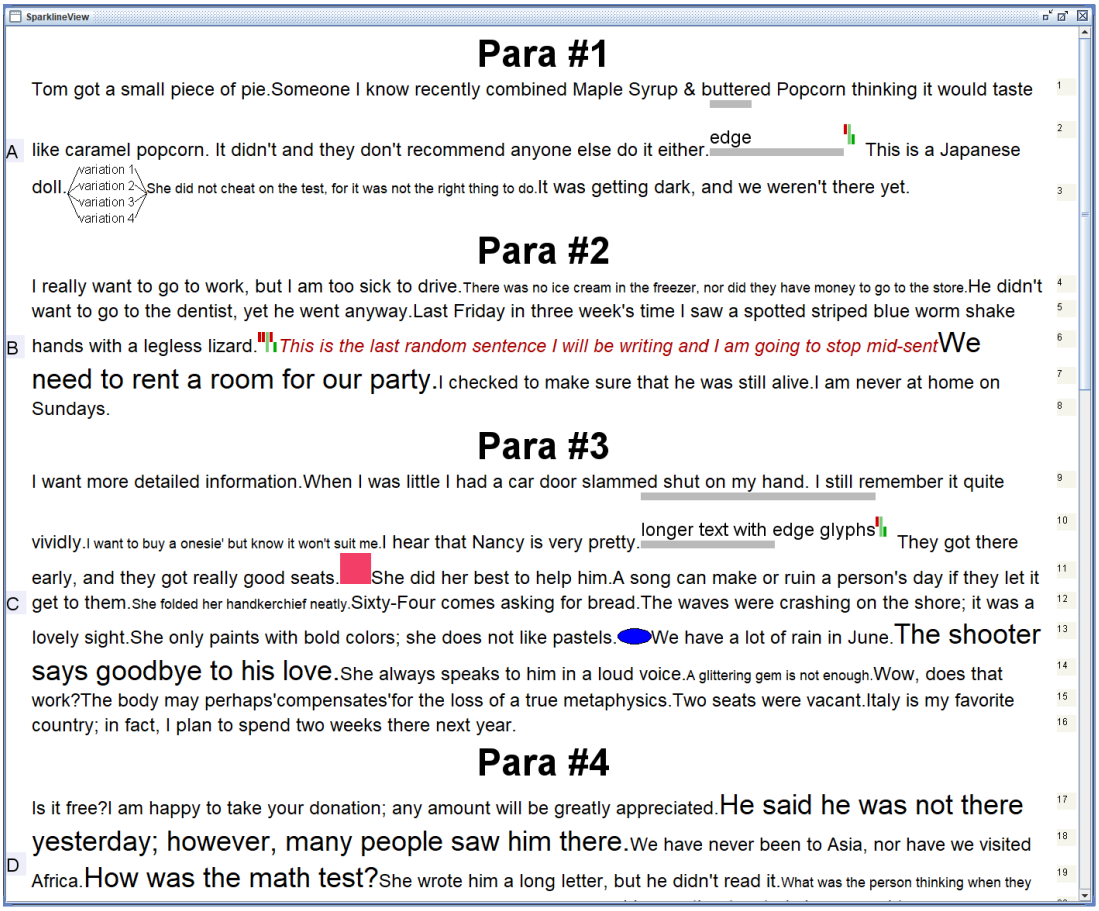


Figure 1.1: Screenshot of a visualization designed using the text view framework.

Framework design goals, the process followed to achieve the design, and expected benefits based on several examples are discussed in **Chapter 3**.

Implementation of the framework including the components of the layout, introduction to the Java components used, and the design of data views are described in **Chapter 4**.

An introduction to digital humanities and the humanities process of scholarship, and evaluation of the framework based on tasks that scholars of classical Latin perform during analysis, are discussed in **Chapter 5**.

Conclusions with suggestions for future work and a summary of contributions are given in **Chapter 6**.

Chapter 2

Background and Related Work

This chapter gives an introduction to visualization, discusses related work in text visualization, and provides background on various visualization techniques that influenced the design of the framework.

Visualization is a technique for creating images, diagrams, or animations from data sets to explore data, analyze data, or communicate a message [9]. Common visual representations of data include scatter plots, bar charts, line plots, pie charts, maps, calendars and many others.

The text view framework draws inspiration from past work in the areas of text visualization, typography, and coordinated multiple views. For text visualization, we discuss visualization techniques that focus on textual information. Most text visualization techniques focus at the level of discrete words. Only a few preserve some of the higher-level structure of traditional text formats. Typography concerns itself with various ways of laying out text on the page. Coordinated multiple views connect multiple views in a single visualization to show different aspects of the data simultaneously.

2.1 Visualization

Visualization is representing data visually in a form that is understandable by the user. Visualization reduces the overhead on the user's end to remember, recollect, and process the data. This allows for better sense making. Even tabular views, such as spreadsheets, are themselves representations of data. Coordinated multiple views can allow users to see and interact with multiple representations of data in a single user interface. For example, a bar chart and a map can be used to show the population of an area in complementary ways. The map can show regions to provide a geographic overview. The bar chart can show an accurate population for each region. Interactions between multiple views allow the user to explore the data, find interesting patterns, and analyze them. For instance, selecting a region in the map would highlight it both in the map and the bar chart. In general, multiple views can be linked through interactions that trigger item selection, sorting, or filtering. Coordinating multiple views in this way allows users to see and interact with more complex data than could otherwise be done in an individual view. In the text view framework, coordination is a way to both link different parts of the layout such as margins and headers, and to link the text view overall with other kinds of data views.

2.2 Text

Although text can be treated as data, it is also something that can be seen and interpreted. Presented text in itself is a form of visualization. Traditional presentations of text include prose, poetry, and plays in books, papers, and tablets. Traditional forms of text usually have layout, styling, and decoration properties.

Printed text layouts can involve different directions—such as left-to-right, top-to-bottom in most Roman alphabet languages—and different groupings—such as single or multiple vertical columns. Styling properties such as italics and bold can be used, for instance, to add emphasis to the text.

Traditionally, styling text involved annotating or marking up parts of that text by hand. *Markup* is defined as a "system for annotating a document in a way that is syntactically distinguishable from the text" [7]. Traditionally, typographers would mark up text documents to indicate the typeface, style, and size of text, which is later typeset for printing [10].

Today, text is read and interpreted on digital screens in various applications such as word processors and web browsers. In the digital medium, controlling the style of text presentation is usually done by including markup information in text storage formats. Prominent markup languages include HTML and TeX.

2.2.1 Visualizing Text

Many text visualizations have been developed. Most visualize individual words. Tag clouds are a well known example [23]. A few preserve text structure above the level of individual words.

- In *tag clouds*, individual tags encode word frequency as font size [23]. An example tag cloud is shown in Figure 2.1. There are a variety of ways of laying out tags in tag clouds. The position of a tag can depend on the alphabetical order, importance, or even be random. The colors of tags can be determined by a variety of different data attributes such as file names, size of files, date created, number of occurrence of words, or importance of words.

Figure 2.1: Barack Obama's 2009 State of the Union speech visualized as a tag cloud [23].

- WordTree [25] is another type of text visualization that focuses on finding word repetitions in a document by displaying the common words and all its associated sentences in a tree-like structure. A screenshot of WordTree is shown in Figure 2.2.

- Poemage [16] is a coordinated multi-view visualization to explore sonic structures in a poem. Figure 2.3 shows an example. It has three views: a set view, a poem view, and a path view. The poem view contains the full poem text displayed in the familiar way. Similar sounding words are uniformly color-coded with an oval around them. The set view displays words that are lexically or phonetically similar. The path view shows the "sonic topology" of the poem.

- TexTile [3, 4] combines pixel-based and focus+context techniques to visual-

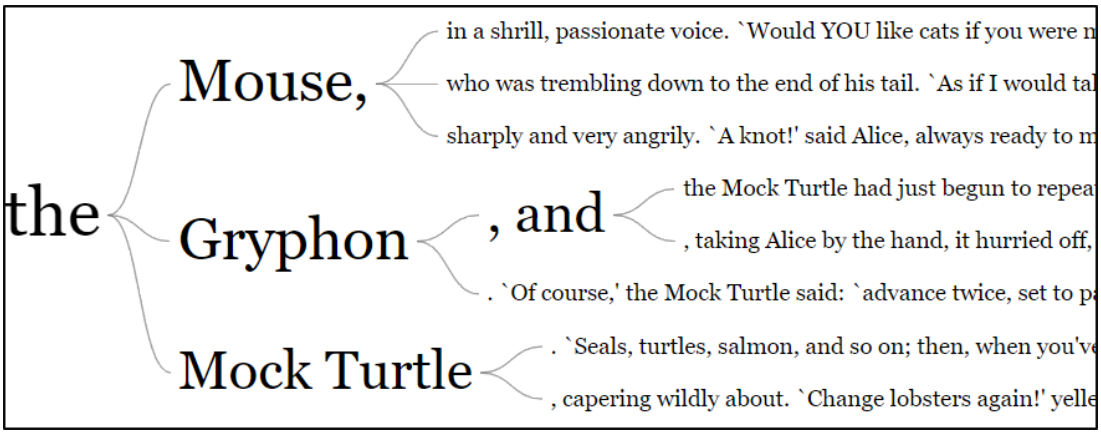


Figure 2.2: Snippet of Alice in Wonderland visualized using WordTree [25].

ize variations between multiple copies of text. Information about variants are mapped to small squares that use color to encode attributes such as lexical similarity. It efficiently uses screen space to represent large amounts of information about text. A screenshot of a TexTile visualization is shown in Figure 2.4.

- Story Tracker [15] provides an interactive visualization for analyzing similar news topics that split and merge over time. It extracts keywords from daily news articles and vertically groups them at each point in time. Related articles are color-coded and linked together.

Most text visualizations like tag clouds and Story Tracker convert text to individual data items and do not represent the higher-level structure of text. These types of visualizations are useful for exploring word-level relationships. However, these techniques do not represent the higher-level text structures that provide the context needed to navigate complex information in large text documents.

A few visualizations such as Poemage and WordTree preserve some aspects of text structure. The text visualization framework preserves traditional printed

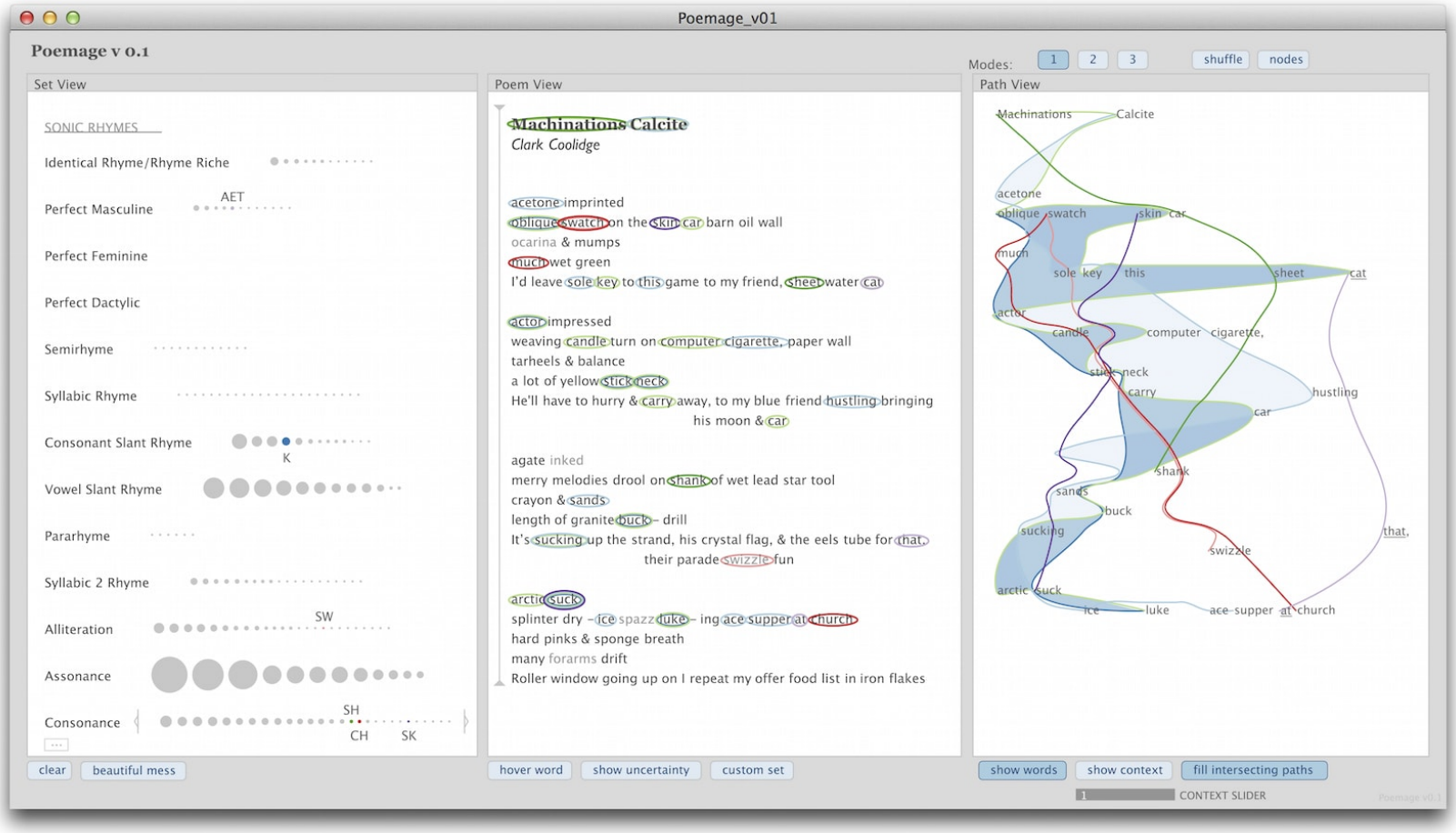


Figure 2.3: The set view shows words that are lexically or phonetically similar. The poem view shows the poem with words linked directly in the text. The path view shows the "sonic topology" of the poem. McCurdy, et al. Poemage [16] © 2016 IEEE.

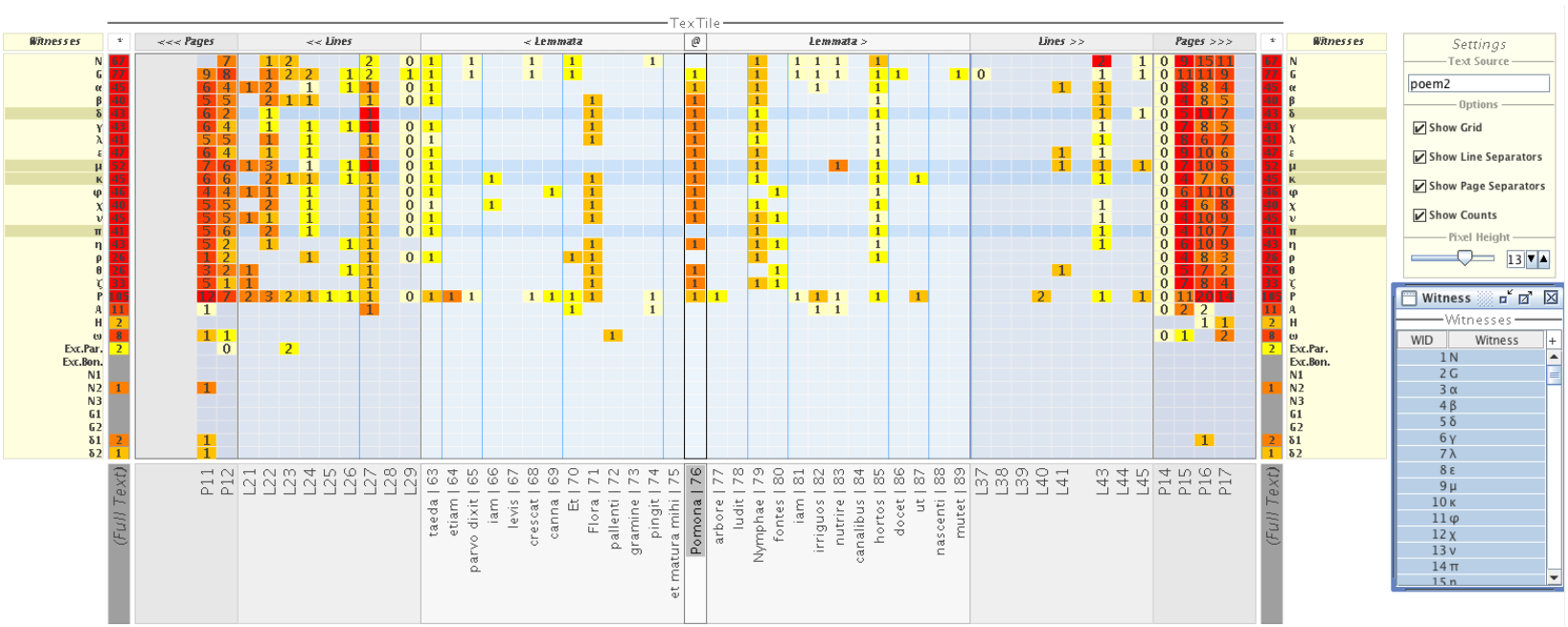


Figure 2.4: TexTile visualization [3, 4] showing pages, lines, lemmata, and witnesses. Witnesses are shown in the left and right margins. The center panel shows pages, lines, and lemmata, and can be scrolled horizontally.

layout generally by presenting the entire text at once with embedded data views to provide additional information.

2.3 Typography

Typography refers to the way characters are placed or rendered. Most, but not all, typographic techniques can be replication on a digital screen [8]. Moreover, a variety of factors such as screen size, render window size, and pixel density need to be considered when rendering text to a digital screen. Text display on early digital screens offered a limited number of typefaces and were usually constrained by screen size and resolution. On more recent displays a variety of layout factors such as line length, typefaces, inter-line spacing, letter spacing, margins, alignment, and kerning can be accounted for when rendering characters. In designing the text visualization framework, we focused on line length, typefaces, inter-line spacing, and margins, because these were most desired formatting features for the Digital Latin Library application.

2.3.1 Line Length

Line length is the physical length in which text is rendered. The length of a line depends on window size and side margins. The number of characters in a line depends on font size, typeface, styling properties, and character density—the number of characters rendered for a particular length. In a traditional text format, when the width of the line is filled the successive characters are broken up and moved to the next line; this process is called *line* or *word wrapping*. In the text view framework, text and graphics that can be rendered on a line are affected by the number of characters, typeface, and graphic dimensions.

2.3.2 Typeface

A *typeface* is a collection of glyphs, each of which represents an individual letter, number, punctuation mark, or other symbol [10]. Each font of a typeface has a specific weight and style. For example, typeface "Helvetica" can have fonts such as "Helvetica bold," "Helvetica bold italic," etc. The size and style of a font can be affected by design choice or by interactive properties. In the text view framework, a variety of typefaces from the Improvise framework are available to the visualization designer. The framework also supports multiple typefaces in a single line.

2.3.3 Inter-line Spacing

Inter-line spacing refers to the spacing between lines. It can be used to vary the amount of text displayed on the screen. Traditional line spacings are single and double line spacing. In printed text, double line spacing is easier to read and allows for adding comments and annotations between lines. However, in this framework, we use single line spacing as it is compact and could render more graphics and characters at any given time.

2.3.4 Margin

Margin refers to an area along the edge of a page. Margins have been used as space for showing additional content or annotations. Familiar margins in a page are on the left, right, top, and bottom. The text view framework uses margins to show additional information, such as line number or speaker of a paragraph. The contents and layout of the margins depend on the main content of the page.

2.4 Composite Views

Composite views involve two or more data views combined together to form a single view. In this framework, for a given text and its associated data the designer has the capability to build compound data graphics. Composite visualizations [13] primarily follows one of four different patterns of merging two representations into one: nesting, juxtaposition, superimposing, and overloading.

- *Nesting* is placing one data view inside another. The text view framework supports nesting by *nested view* encoding. Nested views allows for placing an entire data view inside the text display.

- *Juxtaposition* is placing data views side-by-side or in similar non-overlapping layout. The text view framework supports juxtaposition through *compound encoding*. Compound encoding allows for placing encoded items on the sides and at the corners of other encoded items.

- *Superimposing* is placing one data view on top of another. A *stack encoding* in the framework can take multiple data graphics and render them on top of each other in the specified order.

- *Overloading* is utilizing the space of one data view for another. In the text view framework, we are using the layout of the text as a way to position different data graphics in it.

Sparklines are a common form of nested visualization that embed plots in text. In sparklificator, "small high-resolution data graphics, included alongside words or word sequences in text documents, can often communicate information that could not be succinctly conveyed by the text itself" [11]. The graphic can

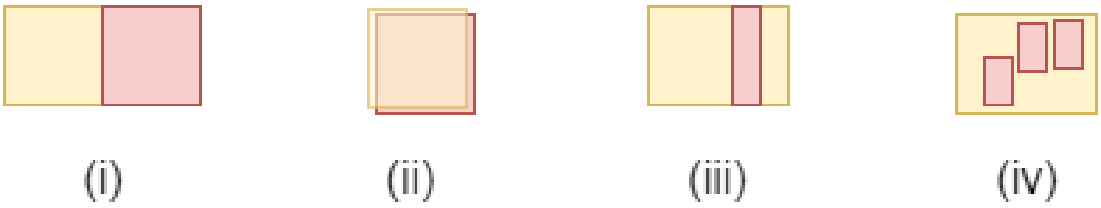


Figure 2.5: Illustration of various composite views. (i) Juxtaposition, (ii) superimposing, (iii) overloading, and (iv) nesting.

be placed in three positions—at the top, on the right, or along the baseline. Top places it above the word, right places it to the right of the word, and baseline draws it behind the word. In the framework, these options can be reproduced with a combination of nested and compound encodings.

2.5 Visual Encoding Design

Visual encoding is the process of mapping data to visual entities. Text has properties like text styling, highlighting, and decorating to differentiate data values. Data views can be treated as independent objects or can provide context by being spatially connected. A data view can have a variety of visual properties such as color, shape, orientation, 2D, and 3D. A data view can also contain primitive shapes such as rectangle and oval combined together to represent the encoded data. For example, a data set containing attributes X,Y and Z can be mapped to position of a circle with a fixed size. With various visual channels like shape, color, size, orientation, and position it is important to choose and combine these design aspects carefully to convey the information clearly.

We are framing our design decisions for the text view framework based on glyph design guidelines provided by Chung et al [6], as follows.

Typedness refers to choosing the appropriate visual encoding for the data type.

The data types may include: nominal, ordinal, interval, ratio and directional. The framework supports typedness by providing the tools to the visualization designer to map data values to different visual encoding channels.

Channel capacity refers to the number of values that can be encoded. For example, a piece of text can have various properties like font size, color, typeface, decorations like underline and strikethrough, and styles like bold and italics to represent different channels. The design should be flexible to represent multiple values.

Learnability refers to easiness to interpret and remember the data views. With contextual data like text, it is important to have its associated graphic linked spatially.

Attention balance says attention should be given to important variables. A variable can show its significance by changing the channel capacity. For example, to show emphasis of a word, font size can be increased or the styling can be set to bold.

Focus and context is about how individual elements should be identifiable under certain operations. For example, interactions like selection should highlight the selected items. In this framework, laying out the text and graphics in a unified traditional format allows for focus and context. The individual graphic is a focal point and text provides the context.

Chung also lists visual orderability, separability, attention balance and searchability; they were not considered in the development of this framework as they did not directly influence our decision making.

2.6 Data Mapping

Data mapping is the process of mapping individual data items to a graphical object. By modifying the order of the dataset we can create multiple views from the same dataset. It is important to figure out which order would be beneficial to the user. Borgo et al. [5] have suggested several possibilities:

Correlation driven: Grouping a data set based on correlation creates clusters of similar variables. Correlation helps identify outliers and understand relationships between different clusters. As an example, correlation within a nested view by embedding data views into text has been implemented in sparklines [11]. The framework supports correlation by allowing the designer to embed entire data views or have similar views in different parts of the text.

Complexity and symmetry-driven: Gestalt principles [14] have found that we as humans have a preference for simple shapes and we tend to remember symmetry patterns. We support complexity and symmetry driven design by providing a set of data views to set alignments to left, right, and center or inherit the width of another data graphic.

User-driven: The end users control the data items that they want to view. Features in Improvise can modify views based on user actions. Coordinated multiple views [19] allows actions like selection in one view to affect the data mapping in other connected views.

2.7 Summary

In this chapter, we have discussed visualization in general, representing text, various text visualization techniques currently used, and how they relate to our

framework.

Chapter 3

Framework Design

In this chapter we discuss the various components of our design framework, various activities of users who develop and use this framework, our design goals, and our approach to achieving the goals. The contributions to this thesis discussed in this chapter are:

- identifying design goals for the framework,
- designing various visual encodings, and
- deciding on types of interactions for the framework.

3.1 Components of the Design Framework

This framework is intended to take traditional text layout and integrate data graphics into it, preserving as much of the structure of traditional text layout as possible to provide linguistic *context*. Some of the major components of this framework are text layout, data graphics, margin and interaction.

- Text layout dictates how text is laid out on screen.

- Data graphics allows for different types of visual mapping.
- Margin gives additional reference information.
- Interaction allows for coordinated between multiple views.

Each of these component is motivated by practical limitations in implementation, needs of the Digital Latin Library[1] project, and maintaining the generalizability of the framework.

3.1.1 Text Layout

A variety of properties affect the way text is laid out. Some of the major properties are alignment, orientation, justification, line wrapping, single and double line spacing, and character spacing. These properties are specific to text and combinations of these properties can be used to generate sophisticated text layouts. The framework is built by extending an existing text rendering package. Hence these text layout properties can be supported by this framework.

3.1.2 Margin

Margin is the area surrounding the contents of a page. Margins have long been used to record additional information like comments and notes relating to the text. Margins can be at the top, bottom, left, and right of a page. Dimensions of the margin can affect some text layout properties. For example, a right margin with a larger width will affect line wrapping. For the framework design, we choose to have the left and right margin be linked to line height and paragraph height in the text layout. This allows the visualization designer to specify additional

[1] `http://digitallatin.org` Accessed: 2017-07-31.

information specifically mapped to those parts of the text layout. The top and bottom margins can be used to show the header and footer of the page.

3.1.3 Data Graphics

Data graphics are visual representations of data that are presented in a way that is easier and quicker to understand and interpret than tabular data. The graphic objects can contain geometrically primitive shapes, combinations of these shapes, and sometimes non geometric graphics like text. These graphics usually have parameters that can change their appearance. Some of the common parameters are width, height, foreground color and background color. Using these parameters, multiple variations can be designed. We also have the ability to compose the graphics by combining two or more primitive shapes to generate interesting designs. For this framework, we choose to design a suitably comprehensive list of visual representations consisting of primitive and embedded graphics to support variety of data dimensions.

3.1.4 Interaction

Interactions in a view are primarily interfaced with a mouse, keyboard or a combination of both. Some of the forms of interactions include navigating the document, altering the size of the entire view, and clicking to select data items. In a traditional text layout, clicking and dragging to select a portion of the text is a common form of interaction. In data visualization, selecting multiple data items by lassoing is another form of interaction. In this framework we intend to support multiple forms of interaction like single selection, text selection and lasso selection.

3.2 Levels of Users

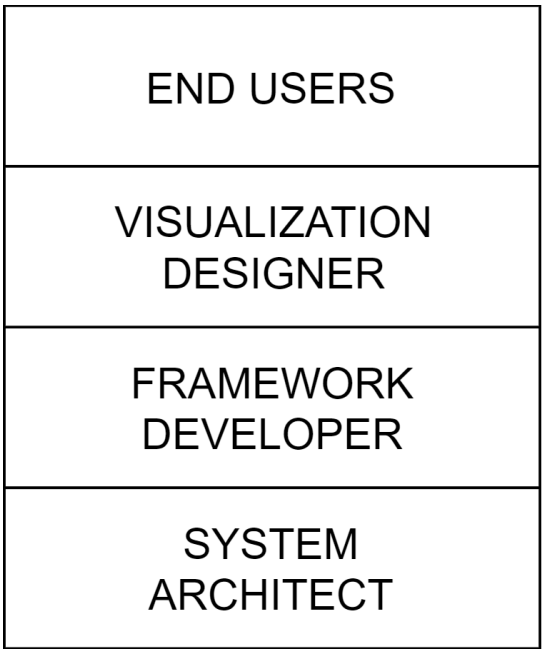


Figure 3.1: Levels of users of the text view framework.

Figure 3.1 shows the levels of each class of users. We filled the roles of framework developer and visualization designer. As it was an iterative development process, we had to switch roles when necessary to satisfy the needs of the project and the view requirements.

3.2.1 End User

The end user would interact directly with the visualization designed from the framework. Some of the users who work with text are from humanities, linguistics, and journalism. The look and feel of the view is important to the end user. The end users would read the text and interpret the data graphics to look for patterns and anomalies. They need knowledge on navigating and interacting with the view. Some of the interactions that an end user might perform are selecting text

to copy, selecting data items from any of the panels, and scrolling to move around the text.

One of the primary users of visualizations designed with the text view framework are scholars in digital humanities. The list of tasks that scholars perform was captured by Abbas, et al [1, 2]. Some of the tasks that scholars perform are reading and analyzing documents, finding related documents, gathering data from these documents, visualizing the data to look for word variants and relationships between documents, and finally creating critical editions.

3.2.2 Visualization Designer

The visualization designer uses the framework to create visualization tools. The designer uses the interface-level features in the Improvise application to design visualizations. Visualization designers would choose the layout for their particular use case. They sometimes need to import the required data set to the application, choose the appropriate views for the use case, map data to these views, and link coordination properties between multiple views. In the text view framework, the visualization designer would need to project data items to the appropriate data graphics, and pass the projections to the appropriate panels in the view. *Projection* refers to ways data can be mapped to data graphics in Improvise. The designer might need features like titles and hard line breaks to begin a new section or end a paragraph. Ultimately, the designer is responsible for the overall look and feel of the view.

3.2.3 Framework Developer

The framework developer writes software to implement the design goals of this framework. The developer is responsible for choosing the tools like packages and libraries required to accomplish the goals. In this text view framework, the developer needs knowledge on Improvise view implementation, designing the individual panels, passing data from one panel to another, designing data graphics, linking to Improvise interactive properties, and providing view level interactions like navigation and selection. The framework developer can also choose to provide a fixed layout, a collection of layouts or a framework to design layouts. Panels in a view might depend on another panel for data, layout, and interactive actions. The developer also needs to ensure that the different panels in the view align as intended. For example, a margin panel would need to be aligned to the top of the page. Additionally, the different data graphics objects need to be provided in the interface design for the visualization designer to map data to these graphics.

3.2.4 System Architect

The system architect lays the foundation of the base application. This foundation is designed and developed in a way to provide tools and resources to the framework developer. The base application should readily be extensible and modifiable depending on the framework developer requirements.

3.3 Design Goals

A wireframe of the framework design is shown in Figure 3.2. Although there are a variety of layout designs available, we chose the Digital Latin Library text

layout as it meets the requirements of the project and can also be used in many other general cases. Figure 5.1 shows the layout of Digital Latin Library text. The layout contains base text in the center, line numbers on the right margin, and author of the paragraph on the left margin.

Analyzing the functions and tasks that a user might perform (see Section 3.2), we have formed a set of goals for this framework design:

- *Use traditional text layout to show text and embed data graphics.* Most end users are familiar with the traditional text layout. Embedding data graphics inline with the text provides additional information while maintaining the layout.

- *Provide variety of data graphics representations.* A variety of graphic representations provide the designer with flexibility in representing data. For example, when visualizing text variations, the designer can choose to list all the variants or show the count of variants as superscripts to the main text.

- *Provide multiple visual channels for the visualization designer to modify the appearance of the text and its associated graphics.* Multiple visual channels allow the designers to use single encodings in different ways. For example, in a dataset with frequency of words, color and size can be used to differentiate between words with higher and lower frequency.

- *Allow for additional context information like header, footer, and line numbers in margins.* A header can be used to show the title of a page or a section header. Headers are used in classic Latin documents to show the title of the poem and list all the authors in that poem. Margins are used

in the documents to show line numbers and speakers of paragraphs.

- *Allow for coordinated multiple views.* Coordination between multiple views provides the benefits of other views combined with text view. For example, textual data that involves geographic location, a map view can be designed to plot data points on the map and the text view can display the text with its data in a desirable encoding.

- *Allow for navigation within the view.* Navigation allows the user to scroll between texts. To traverse large documents, navigation is essential.

- *Allow selection interactions.* Selection can allow the user to filter the data to look for specific details or to select text to copy to clipboard.

3.4 Panel Design

A variety of layout designs can be formed by combining multiple panels. Some of the common layouts are two panel layout—two panels are laid side by side to occupy the full window width, and margin layout—a center panel is surrounded on four sides by margins. We chose the Digital Latin Library text layout due to practical reasons and to focus on visual encoding design.

To achieve the layout, the framework is split into three panels: Center panel to hold text and data graphics in traditional layout, left margin panel to provide paragraph context, right margin panel to provide line context.

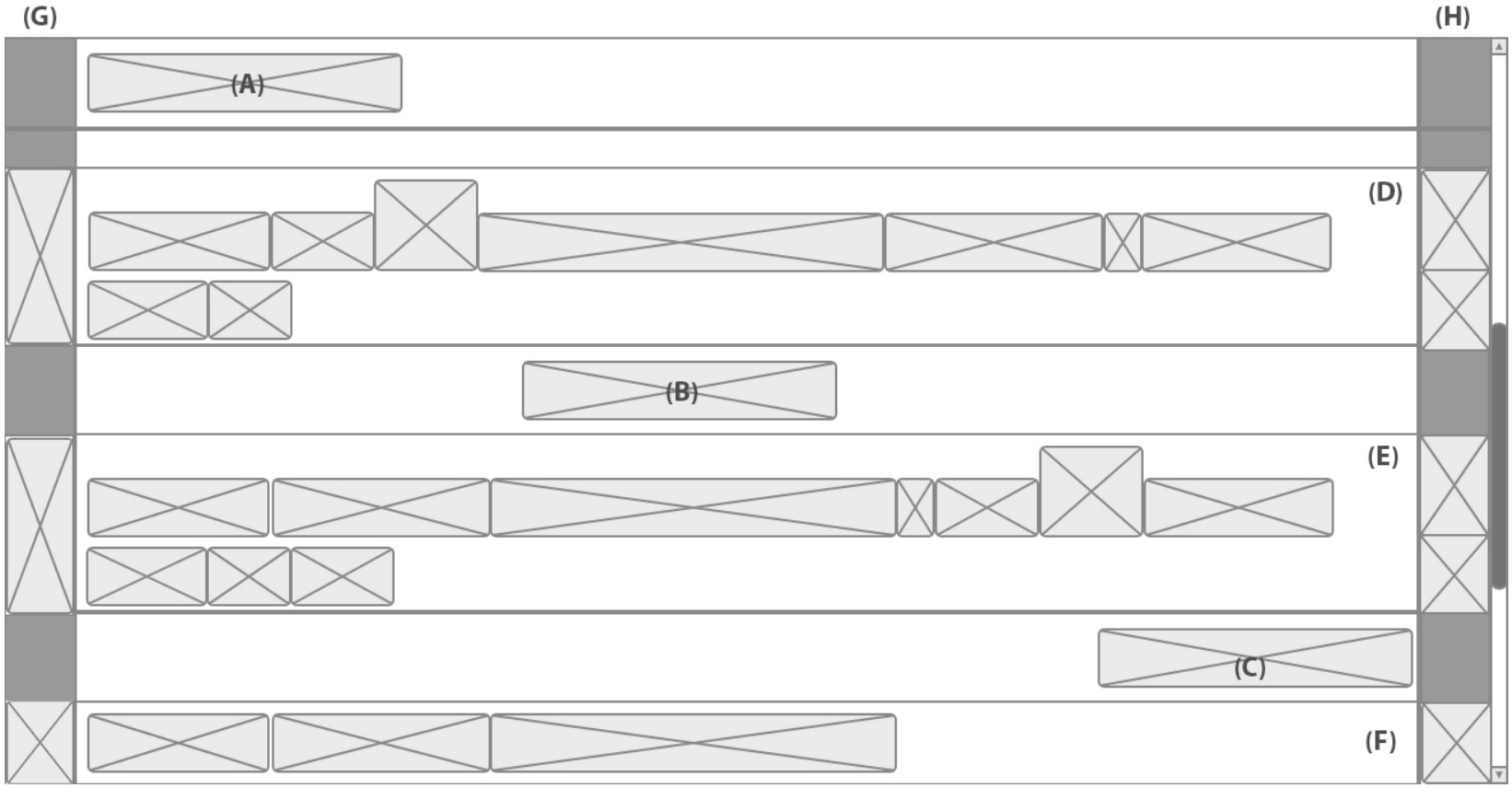


Figure 3.2: Wireframe design of a text view. (A), (B) and (C) are titles, (D), (E) and (F) are paragraphs, (G) and (H) are left and right panels, respectively.

3.4.1 Center Panel

In the center panel, the text and data graphics are laid out in traditional text layout with line wrapping at the end of a line. The typeface, size, and style can be changed to the designer's preference. The parameters of visual encoding are modifiable from the projection designer in Improvise.

3.4.2 Right Margin Panel

The right margin panel is designed to work like a table with a single column and multiple rows. The number of rows is dependent on the number of lines in the center panel. Data for the right panel is fed from the center panel. Data contains line number and height of the specific line. When the center panel is relaid and the structure of the text in it changes, the right panel updates the height of each line. The projection for the right panel is designed similar to that of the center panel.

3.4.3 Left Margin Panel

The left margin panel is populated using Improvise data and projection. It is similar to the right margin panel in layout. It contains a single column table with multiple rows, with the difference being the height of the row is tied to the height of the paragraph.

3.5 Visual Encoding

Visual encoding refers to ways data can be transformed to a visual representation. Visual encoding can be primitive or nested. A primitive graphic contains only

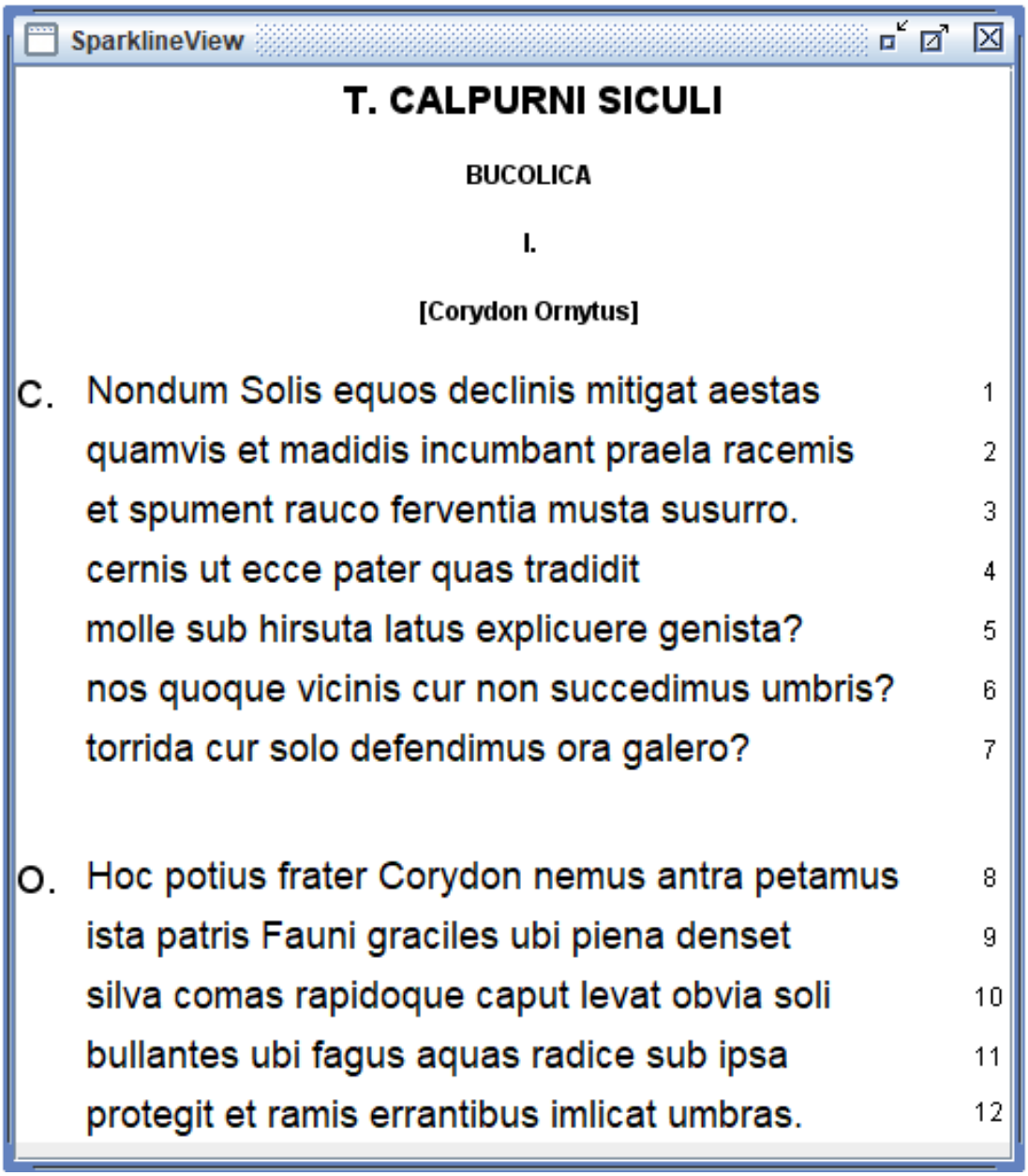


Figure 3.3: Screenshot of the classical Latin poem Calpurnius Siculus [20] visualized as a text view.

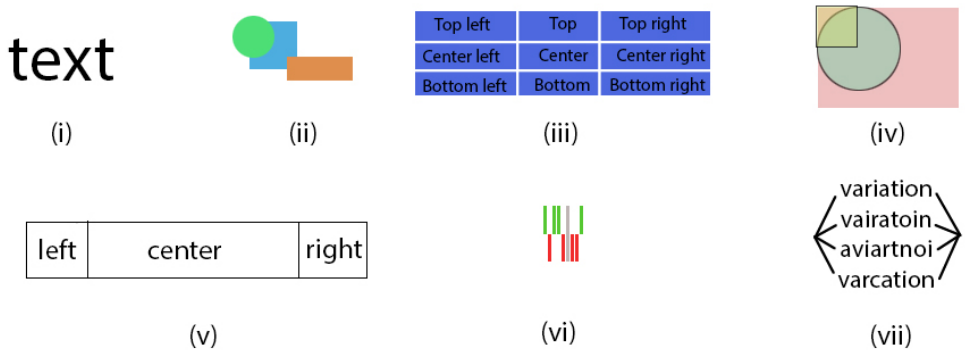


Figure 3.4: Visual encoding designs for the text view framework. (i) Text (ii) Icon (iii) Compound (iv) Stack (v) Title (vi) Tristate (vii) Nested Views

itself; a nested graphic can hold multiple primitive graphic elements in its own custom layout. In this section we will look at different types of visual encoding that are supported by the text view.

- **Text**: *Text encoding* is responsible for rendering the given input text in the appropriate panel. Text can be used independently, or combined with other data graphics to form complex images. Input to the text can be a character or a stream of characters. Text encoding features that can be altered are font, size, styles such as bold and italics, and additional styles such as underline and strike-through. This framework is built around text, hence text had to be one of the primary visual encodings.

- **Icon**: *Icon encoding* contains primitive graphics that can be added inline with texts. Primitive graphics in this framework are squares (or rectangles) and circles (or ovals). Primitive icons have a special case—they have can inherit the width of another element. Icons are independent, i.e., they do not rely on other encodings to draw themselves. Icons are essential, as they can be combined together to form complex images. For example, squares

(or rectangles) can be combined together to draw bar graphs.

- **Compound**: *Compound encoding* allows for placing graphics near sides and corners of text items. Placing around text preserves the relative position of the text. Text has line spacing above and below, which can be used to place data graphics. However, the line spacing may need to be altered to avoid overlapping graphics with other text in the layout. Combinations of vertical, horizontal, and corner positioning allows flexible encoding of multiple pieces of information around text items.

- **Stack**: *Stack encoding* allows for piling graphics on top of another. It can take n number of data graphics as input. The data graphics are drawn in the specified order. By varying the opacity channel, different graphics can be designed.

- **Title**: The title is a special-case design. A *title encoding* can be used to denote section headers, the start of a new page or page numbers. A title has three alignment properties—left, center and right. This allows the designer to changes the alignment depending on the use case.

- **Nested Views**: A *Nested View encoding* is self-contained view that can hold one or more data graphics. An example of a nested view is a scatter plot, where data points can be mapped to primitive shapes like rectangle or oval.

The text view framework can be extended by framework developers to add their own icons and nested views. As an example, I implemented variation specifically for the Digital Latin Library project. *Variation encoding* can be used to show variations in a word. It can take n data graphics as input. The variation

encoding displays the data graphics in order, with a line connecting from the sides of each data graphic to the base line.

Similarly, a tristate encoding is a custom designed nested view that can be used to show three states—positive, negative, and neutral. A tristate takes overall width and height, bar width, bar spacing, positive color, negative color, and neutral color. A tristate encoding is also designed as part of an icon as it has a fixed dimension for any given data value and does not require base line adjustments to render. In designing this framework, I added a few encodings like the tristate encoding for the Digital Latin Library application.

3.6 Interactions

Interaction with the text view is primarily through selection. *Selection* is the process of choosing data items. In Improvise, selection can be coordinated with other views in the same visualization. Three ways of selection have been identified for the text view with the help of previous knowledge of the Improvise framework.

- *Single selection* is selecting a single element from the document. Pointing and clicking on a text or graphic element in the text view should select the element.

- *Multiple selection* is similar to selecting text in a text processor or a web page. The user clicks and drags the mouse across the text to make a selection from start to end. The framework should provide feedback to the user by highlighting only the selected text. The data items within that selection range should also be selected. The user should also be able to copy only the selected text to clipboard. Since other forms of selection do

not provide feedback, the copy function can only be provided in multiple selection.

- *Disconnected multiple selection* is useful when selecting multiple elements that are not in order. Using a keyboard modifier and single selection, multiple elements that are spatially separated can be selected.

Some of the advantages of selection are it can help reduce clutter and provide details on demand. The designer can choose to reduce clutter by showing certain visualizations on request by the user. The request can be a mouse click or by selecting the data item. This reduces the amount of information that needs to be shown at a given time and give the user more control over the visualization. Specific details can be shown to the user based on certain interactions. For example, a secondary visualization can be designed to show only the selected information from the parent visualization. This helps the user to request information on demand instead of dumping it all at once.

3.7 Examples

To demonstrate the design of this framework, a news article was recreated with the text view. It is shown in Figure 3.5. Player names are encoded in a color representing their respective country. Each country has its win and loss stats for the last five matches encoded as a tristate graphic. The players and their associated countries are easily identifiable.

Another example designed with random sentences[2] is shown in Figure 3.6. This visualization was designed as a proof of concept to show text layout, various

[2] `https://randomwordgenerator.com/sentence.php` Accessed: 2017-07-31.

Jonny Bairstow made a dashing 60 not out as England thrashed South Africa by nine wickets in the first Twenty20 international at Southampton on Wednesday.

England, set just 143 to win, reached their target for the loss of only one wicket with 33 balls left.

Opener Alex Hales was 47 not out, his unbroken second-wicket stand with Bairstow worth 98 runs.

But the foundations for a victory which put England 1-0 up in this three-match series were laid by their bowlers.

They restricted South Africa to 142 for three after Proteas skipper AB de Villiers won the toss.

De Villiers made 65 not out and Farhaan Behardien an unbeaten 64 in an innings where fast bowler Mark Wood took two for 36.

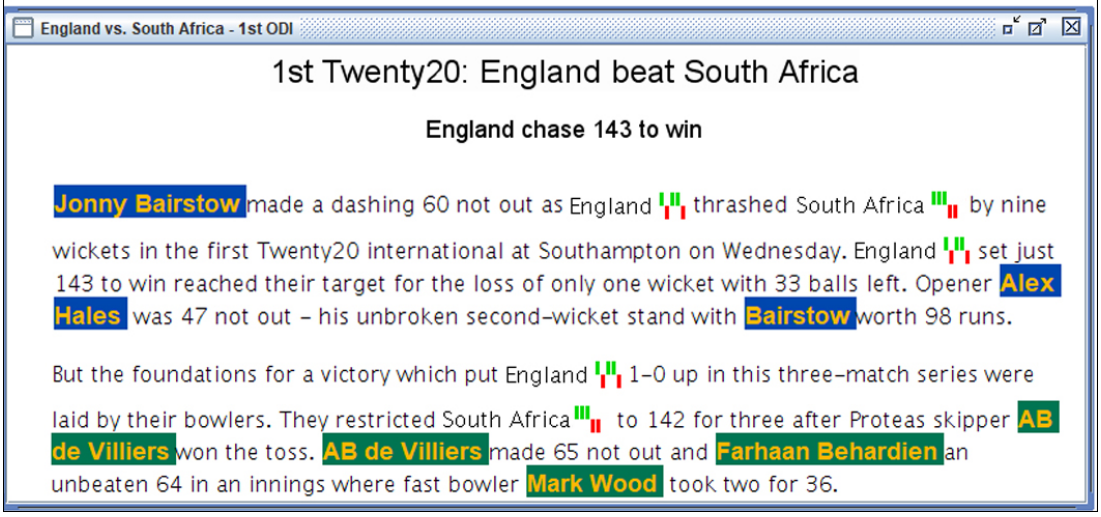


Figure 3.5: Comparison of static text and a text visualization. At the top is a news article and at the bottom is its visualization using the text view framework.

visual encodings, and the different panels. The center panel is marked (15), left and right panels are marked (14) and (13) respectively.

The center panel holds the text, primitive data graphics and combinations of primitive graphics in a traditional format. Text is shown in (9) and (10), (9) has a larger font size and (10) has red as font color. Primitive visual encodings are (7), (8), and (11). (7) shows a square with red foreground, (8) is an oval with blue foreground and (11) is a tristate with positive marked in red, neutral in lighter shade of green and negative in darker shade of green. Complex graphics are marked (5), (6), and (12). (5) and (6) are compound encodings, with four inputs: text in center, rectangle in top and bottom, and a tristate in center-left. The top rectangle inherits the width from the center, hence (5) has a shorter width than (6). (12) is variation encoding, to show different variations in text. (1), (2), (3), and (4) are title encoding. Titles do not flow with the rest of the text. In this example, title alignment is set to center.

(13) shows the right margin panel, with line numbers. The height of each line number is mapped to the height of that particular line in the center panel. (14) is the left margin panel; it has the characters in the alphabet as data mapped to text encoding. The height of each data item is mapped to paragraphs. Having too many visual encoding can make the visualization look distracting. It is up to the designer to maintain an elegant design.

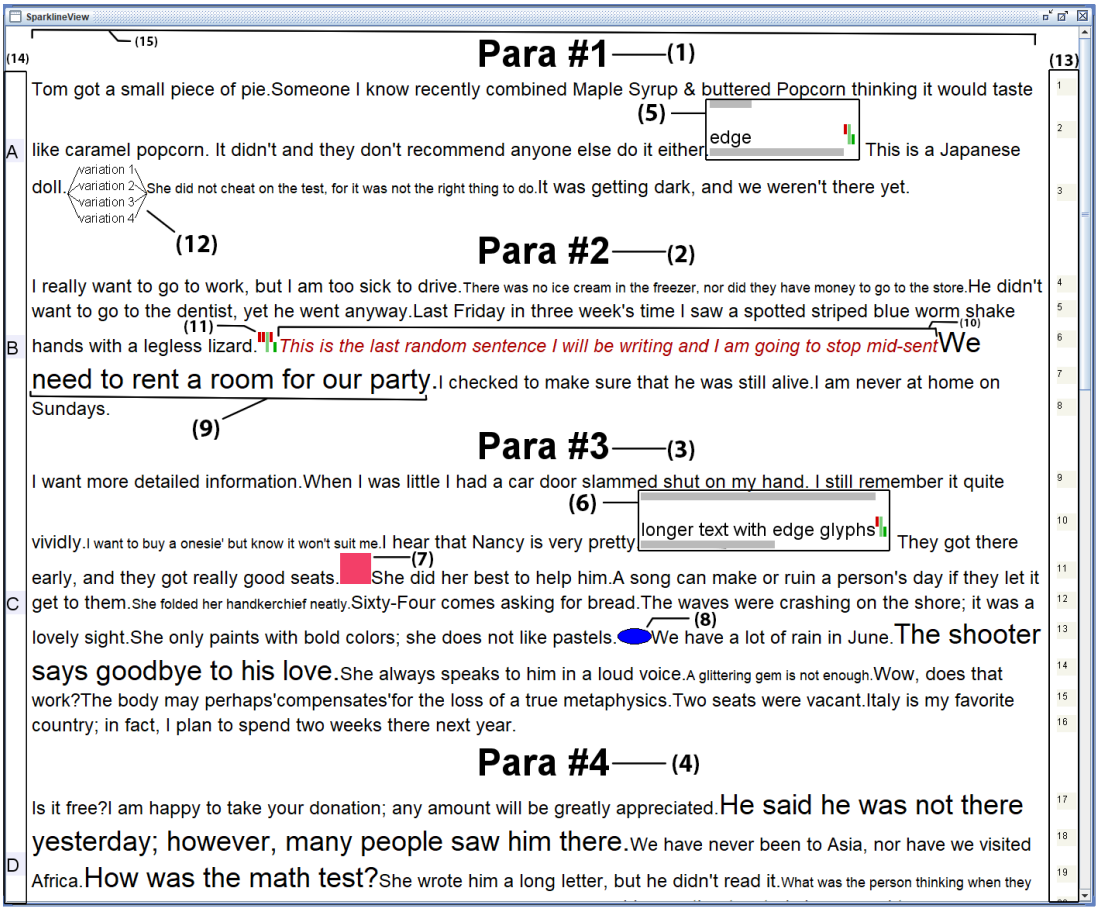


Figure 3.6: Screenshot of a visualization designed using random sentences generated online. (1), (2), (3) and (4) are titles, (5) and (6) are compounds, (7) and (8) is square and oval, respectively, (9) and (10) are texts, (11) is tristate, (13) is right margin panel, (14) is left margin panel, and (15) is center panel.

Chapter 4

Implementation

This chapter discusses about the software implemenation of the text view framework. The framework is implemented in the Improvise [26] visualization environment, a visualization tool to design interactive and coordinated visualizations. The framework is implemented using Java Swing (*javax.swing.text* package) and Improvise. For this thesis, I implemented the following:

- a new text view module that can be used alongside other view modules in Improvise,
- existing UI components from Java Swing into the new text view module (the added UI components are JTextPane and JPanel),
- an extension of JPanel to support multi-view scrolling,
- an implementation of EditorKit to add styled text and data graphics to JTextPane,
- an extension of EditorKit to support special case views like TitleView, EdgeView and VariationView,

- glyph classes to draw data graphics—a glyph here refers to the individual data graphic in Improvise environment,
- an interface implementation of ViewFactory to associate glyphs to new and existing EditorKit View classes,
- new View classes (extension of *javax.swing.text.View*) to encapsulate glyph objects for use by EditorKit—The View classes implemented are EdgeView, TitleView, and VariationView,
- integration of existing View classes in Java for use by EditorKit—the existing View classes are LabelView and IconView,
- mouse and keyboard handlers for UI components,
- passing data between different UI components,
- adding UI components with data mapping to margin panel,
- code to access and initialize glyphs with Improvise data processing language, and
- code to register text views, data objects, projections, and selections with Improvise.

The software stack of the text view framework is show in Figure 4.1. The Improvise framework is developed by the system architect and EditorKit is a part of Java. I implemented the text view using components from EditorKit and Improvise. Visualizations are designed using the Improvise declarative language by visualization designers.

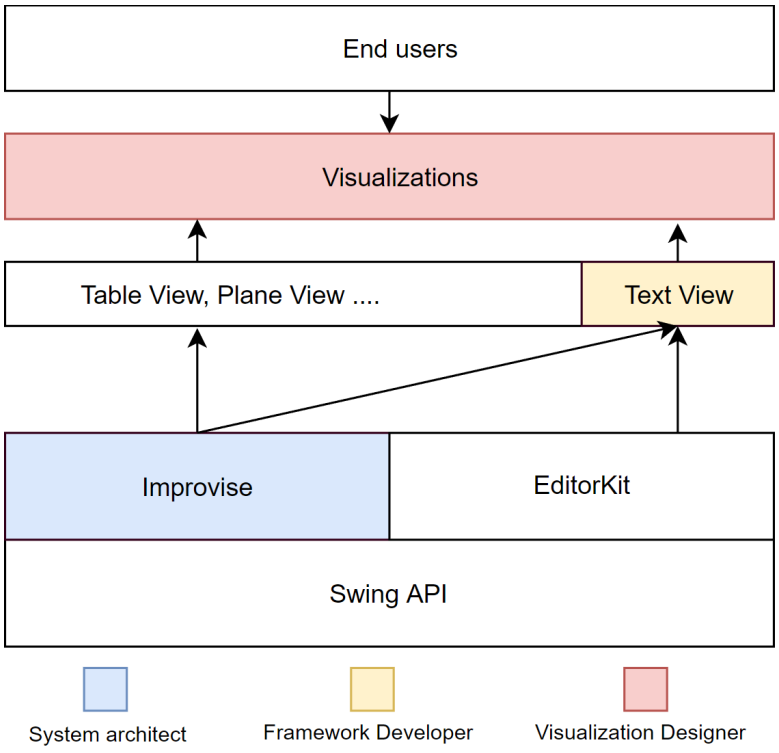


Figure 4.1: Software stack of the text view framework.

To achieve the design specifications in Chapter 3, the overall view is subdivided into three views: a main center panel to hold text and data graphics, a left margin panel to provide paragraph context information, and a right margin panel to show line numbers. This chapter gives an introduction to the components in Java that were used to build this framework, discusses the view designs, describes the design pipeline, and talks about the implementation process.

4.1 Introduction to Java Components

The major Java components used in the text view framework are EditorKit, JTextPane and JPanel. JTextPane is the container component that can hold multiple Java views together. EditorKit is responsible for adding the necessary

views to JTextPane. EditorKit and JTextPane form the center panel. JPanel is a lightweight container to hold multiple JComponents. JPanel is used for the left and right margins. The functions and classes mentioned in this section are part of Java.swing.text package. We will look at each component in detail.

4.1.1 EditorKit

EditorKit is part of the javax.swing.text package. The EditorKit is an abstract class that uses the Document model to read, write, and modify contents in a JTextComponent. Content refers to text, icons, and images. JTextComponent is a base class for various text components in Java. In-built Java implementations of JTextComponent are JEditorPane, JTextArea, and JTextField. The Document model holds characters in sequence with its reference to the position in the overall text. An in-built implementation of the Document model is StyledDocument. The StyledDocument provides styling to text at the character level. The styling properties include font family, font size, color, underline, and strikethrough. JTextComponent creates and returns the Document model via the getStyledDocument() method. Inserting strings to the Document model is done by calling the InsertString() method. The InsertString() method takes start position, character (or string), and Style. Style holds the styling property for the given string. The Document model can also take icons as input. To add an icon, the Style object uses the StyleConstants.setIcon() to set an icon. The icon must implement the Java Icon interface. Inserting to the Document model is similar to inserting a character, but the character does not render, instead the icon is rendered. Insertion of a string or an icon is implemented internally by initializing a LabelView or an IconView, respectively. Figure 4.2 shows LabelView marked

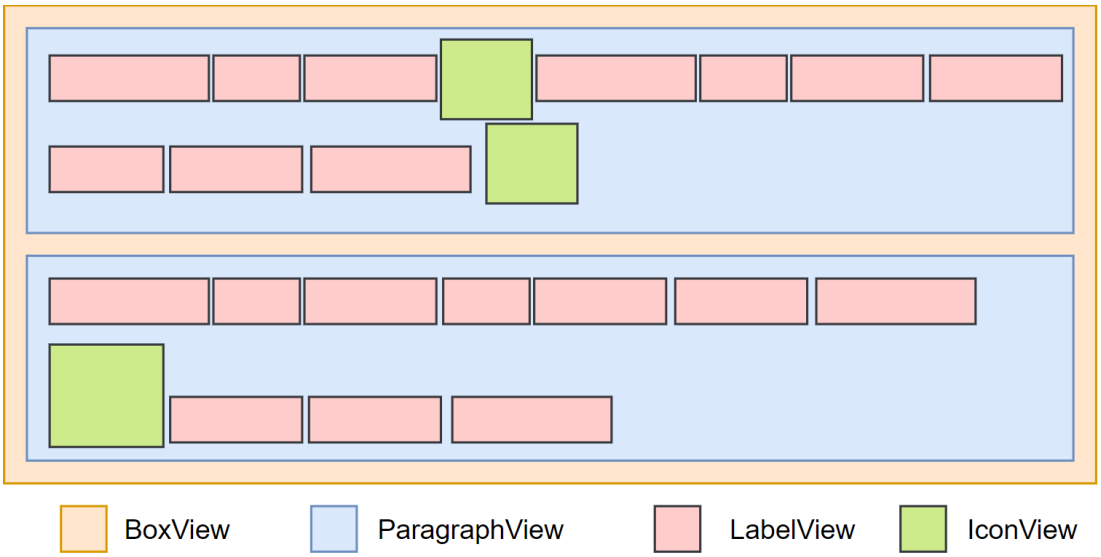


Figure 4.2: Elements structure in EditorKit.

as red boxes and IconView as green boxes.

EditorKit uses views to represent different elements. The main views in the EditorKit are BoxView, ParagraphView, LabelView and IconView. The structure of these views are shown in Figure 4.2. The BoxView holds all the elements added to the EditorKit. The BoxView can hold multiple child elements that can be laid out vertically or horizontally. In Figure 4.2, it is laid out vertically, i.e., along the Y axis. ParagraphView is an extension of BoxView to hold multiple rows in vertical alignment. The width of the ParagraphView depends on the width of its parent BoxView. As content gets added, it calculates the width of the row. If the width of the row exceeds the width of the parent ParagraphView, it creates a new row and flows the content to the next row. This flow is handled by the FlowStrategy class in Java. This gives EditorKit the line wrapping feature. Each row can hold content views like LabelView and IconView. LabelView is an extension of GlyphView, which is responsible for rendering characters and its attributes. IconView is an extension of the View class to draw Java Icons.

4.1.2 JTextPane

JTextPane is a subclass of JEditorPane. JEditorPane allows editing content whereas JTextPane can only add and remove content. We chose JTextPane because editing data items is not supported by Improvise. Elements can be added to the JTextPane in the form of JComponents or by setting an EditorKit and using the Document of the EditorKit to add content. We chose EditorKit because it gives flexibility in text layout like line wrapping, text alignment, and inter-line spacing.

4.1.3 JPanel

JPanel is a lightweight container to hold JComponents. A JPanel can have different types of layouts. The layout manager determines how the components added to the JPanel are laid out. Some of the prominent layouts are BorderLayout, BoxLayout, GridLayout and GridBagLayout. JPanel has an add() method to add components like JLabel, JButton, JTextField, etc.

4.2 View Implementations

This section discusses the needs and process for implementing additional views. I implemented EdgeView for compound encoding, VariationView for variation encoding, and TitleView for titles and section headers. LabelView and IconView are part of EditorKit. I also extended IconView to change the position of the icon. Rendering the icon is handled by EditorKit. Each view has a paint() method that has X and Y for position, Graphics object to draw content, and setPreferredSpan() to set dimensions.

- **LabelView**: LabelView implements the text encoding in Figure 3.4 (i). LabelView by default in the EditorKit given the visual encoding like typeface, font size, foreground and background color and styling properties like bold and italics. No changes were required to satisfy the design requirements.

- **IconView**: IconView implements the primitive shapes and tristate in Figure 3.4 (ii) and (vi). IconView extends from the default IconView in Java's EditorKit. Alignment of the icon along the Y axis is overridden to draw the icon above the text base line.

- **EdgeView**: The EdgeView is an extension of the View class in Java. EdgeView implements the compound encoding in Figure 3.4 (iii). It can take nine data graphics as input and lays them in a 3×3 table in order, starting from top, left to right. To avoid overlapping icons, the maximum width of each column and maximum height of each row needs to be calculated. The maximum width of each column is calculated by using the icon.getIconWidth() method for each icon. The maximum height of each row is calculated by using the icon.getIconHeight() method.

  The first icon is drawn at the X and Y position, the second icon is drawn with X offset to the maximum first column width, and the third icon is drawn with the X offset to the sum of the maximum width of first and second columns. Similarly, the second and third rows are drawn by offsetting Y by maximum height of first row and maximum height of second row respectively.

- **TitleView**: TitleView implements the title encoding in Figure 3.4 (v). The TitleView adds a new line to separate the title from the previous line before

rendering the title. The alignment of the title is passed as a PositionH class, which shows the position to align the title. PositionH is a class in Improvise available to the framework developer. PositionH class has three states: Left, right, and center. The X position is moved depending on the state returned from position. At the end, another new line is added to separate from the next line.

- **VariationView**: VariationView implements variation encoding in Figure 3.4 (vii). The VariationView take n data graphics as input. The VariationView is an extension of the View class. Line alignment on the Y axis is overridden to allow for centering the variation to the base line. Each data graphics is rendered in order, with offsetting to the height of the previous icon.

4.3 Design Pipeline

The design pipeline of this application is shown in Figure 4.3. The raw data is usually in the form of XML. To covert XML to tabular data a variety of XPath [24] queries can be performed. XPath can extract values from specific tags and attributes. The extracted data is usually stored in multiple relational tables. The input data can be filtered, sorted, and projected to different forms in Improvise.

Once the data is processed through Improvise, it can be mapped to data graphics in the text view. Various visual encoding types that can map data to graphics are discussed in Chapter 3. The graphics mapping returns a data object containing the list of data graphics in order. Some graphics are designed as an implementation of Java's Icon interface. Icons know their dimensions and can

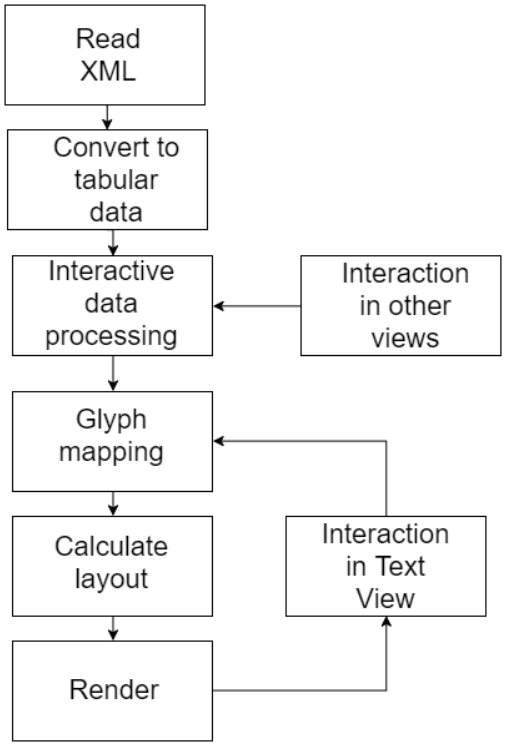


Figure 4.3: Step-by-step processing of data in the text view.

draw themselves. Some graphics are implemented as Views in EditorKit. Each View has a setPreferredSpan() method to store dimensions. With each graphic size information, the center view has a renderer that calculates the layout for the list of data graphics and renders them in its Cartesian coordinates. The renderer passes the dimensions of each line and paragraph to the right and left margin view respectively. The left side panel calculates the height of each paragraph and renders the given author data, if any. The right panel calculates the line height and adds line numbers, if a projection is specified. Improvise can notify its views when the underlying data has changed. This can be triggered by filtering, sorting or changing projections in another view. The graphic mappings are recalculated and the process continues until the side panels are rendered. Selecting elements in the text view triggers the render() function, which redraws the entire view.

4.4 Panel Implementation in Java

From an implementation perspective, the three views are contained in a container ScrollableJPanel. Since JPanel by default does not support continuous multiple child view scrolling, it had to be extended to override the getPreferredScrollableViewportSize() method. The extended class is called ScrollableJPanel. I designed and implemented the ScrollableJPanel to hold center and margin panels. In addition to that, ScrollableJPanel is used by margin panels to render data graphics using JLabel. The center panel uses the Java swing component JTextPane, which is an extension of JEditorPane. The left and right margin views are ScrollableJPanels with BoxLayouts. ScrollableJPanel is added into an implementation of JScrollPane, BetterScrollPane. BetterScrollPane is part of the Improvise architecture. BetterScrollPane is added into an implementation of JComponent, which is also a part of Improvise. Initializing these panels are handled by a parent class called TextView and initializing the TextView is handled by Improvise.

4.4.1 Center Panel

The EditorKit is comprised of layers. Lower-level views like Text and Icons are contained inside a ParagraphView, the ParagraphView contains multiple rows to wrap the text. Multiple ParagraphViews are contained inside a BoxView with a specified orientation, in this use case its Y axis orientation. An individual view rendering space is shown in Figure 4.2. A vanilla JTextPane can take text and icons as input. For purposes of our application, we need the JTextPane to take additional views such as TitleView, EdgeView, and VariationView. Higher level views such as ParagraphView and BoxView need not be changed. To implement these, the JTextPane needs to have an extended EditorKit. The extended Edi-

torKit would implement the ViewFactory interface. The ViewFactory interface has a create() method to initialize EditorKit views. The ViewFactory implementation handles initializing the required view as and when needed. Data is populated into the view by inserting to the JTextPane's StyledDocument object. Once the Text or Icon is inserted into the Document object, it handles passing the appropriate view to the ViewFactory. Each view has an index variable, which holds the order of the object. This is useful for getting and setting selections to the elements in the text view. Visual encodings are passed to the EditorKit by projection in the Improvise framework. A screenshot of projection to visual encoding mapping is shown in Figure 4.7.

4.4.2 Right Margin Panel

The right margin panel is designed to hold line numbers for lines in the center panel. The right margin panel is built using ScrollableJPanel with its layout set to BoxLayout along the Y axis. BoxLayout is part of Java. The data for this panel is passed from the center panel. The data contains the line number and height of that particular line. Each line number is added as a JLabel with its preferred height set to the height from the dataset. The width of the right margin panel is set as a constant, due to time constraints in project development. Projections for the line numbers can be specified by using the Improvise declarative language. The input schema for the projection design is two attributes of type String. The first attribute holds the line number and the second attribute holds the height of the line. The line number and line height is calculated after the center view layout is calculated.

4.4.3 Left Margin Panel

The left margin panel requires data and projection to render. The left margin panel is built using ScrollableJPanel with its layout set to BoxLayout along the Y axis. The height of paragraphs are passed from the center panel. The data is specified by the visualization designer. The left margin panel adds JLabel for each paragraph in the center panel with its height set to the height of the paragraph. Similar to the right margin panel, the paragraph height calculation is done after the center view layout is calculated.

4.5 Interactions

Interacting with the data is currently only available in the center panel. Three forms of interactions are currently supported for selecting data items:

1. Simple point and click to select the data.

2. Run text selection.

3. Disconnected multi-selection.

Different selections can be activated by mouse and keyboard inputs. In single selection, individual document objects like LabelView, IconView, EdgeView, etc. provide call back on click with the index value, which is assigned to Improvise selection. Run text selection is handled by an extension of the CaretListener class. The CaretListener object returns the start and end position of text selection. With the start and end values, a list of indexes within that range is calculated and added to Improvise. The modifier key “alt” is used for disconnected multi-selection. It is functionally similar to that of single selection, with the addition

that it holds a list of all selected indexes until one of the other selection modes is activated.

Copying selected text could not be implemented as selection can sometimes redraw the contents of the center panel. Due to the redraw, the selection highlighter is lost and setting up custom text highlighting in EditorKit is cumbersome.

4.6 Text Visualization Design in Improvise

In this section, we will look at how structured data is converted to tabular data for use in Improvise, data processing to convert tabular data to a form suitable for text view, and finally project data to a visual encoding that can be used by the text view.

4.6.1 Background on Data Processing

This section discusses one of the ways of processing data in Improvise, as an example. The processed data is available across all views in this Improvise visualization. This framework does not support data processing, however understanding the data design in Improvise is essential to the visualization designer.

For our visualization design, we will use Giarratano's 1910 critical edition of Calpurnius Siculus. An image of a page from the critical edition is shown in Figure 5.1. Latin scholars have converted the printed text to an XML file with the information encoded in various tags. It is a TEI [12] (Text Encoding Initiative) encoded XML file that contains the list of lemma entries and list of variants with the associated witness. *Lemma* refers to text that been chosen by the author to be the original text. A portion of the XML file is shown in Figure 4.4. Since Improvise does not take XML as input, the data had to be

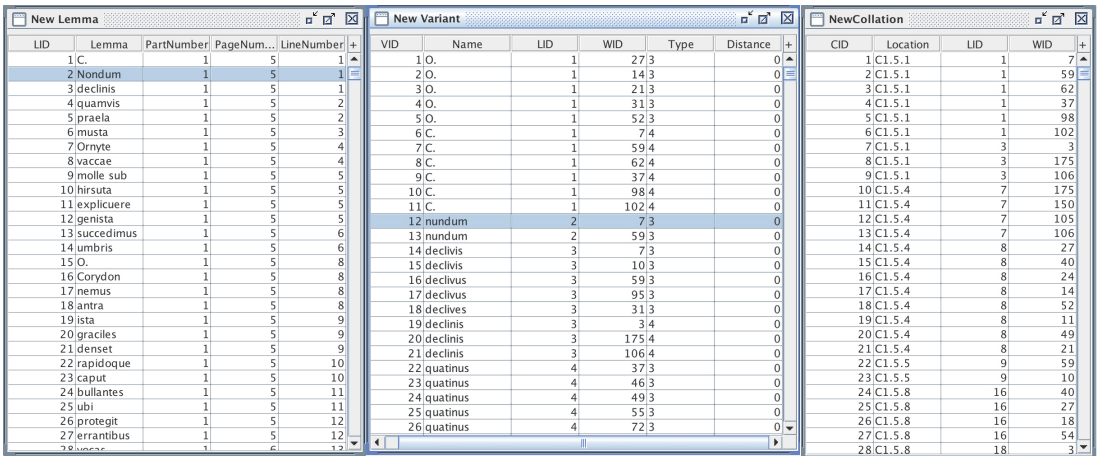
Figure 4.4: A portion of the XML file for the printed edition of Calpurnius Siculus. Line numbers are placed in the ⟨*l*⟩ tag. Lemma information is encoded in the ⟨*lem*⟩ tag, variants for the lemma are recorded in the ⟨*rdg*⟩ tag.

converted to relational tables. With the use of XPath [24], the XML data was converted to relational tables. XPath can select nodes and extract information from a given input XML file. It can be used to target certain nodes to extract information. In our example, an $\langle app \rangle$ node can be targeted to get the lemma and its variants. Additional XPath queries can be implemented to gather the lemma values into a table and the variants to another table. The two tables can be linked via a key.

Figure 4.5 shows the XML converted to relational tables. The lemma table contains the list of lemmas indexed with the lemma ID, LID. It also contains additional information such as the part number, page number and line number. The variant table contains the list of variants for the lemmas, witness and type information. Each variant has a unique variant ID, VID. The lemma ID and variants are linked via foreign keyv in the variant table. For example, in the above image, LID 2 has the lemma "Nondum." Corresponding entries for the lemma are present in the variant table. Variants with LID 2 has values "nundum" and "nundum" with witness ID 7 and 59, respectively. This shows that the variant for the Lemma 2 "Nondum" in poem 1, page number 5, line number 1 were suggested as "nundum" by witness 7 and "nundum" by witness 59. The collation table holds the location of the lemma, lemma ID, and witness IDs for that lemma.

4.6.2 Projection Design

The center panel can take multiple different visual encodings to display different types of data. The data would need additional information containing the type of encoding the designer chooses for that particular data item. For example, to



New Lemma

LID	Lemma	PartNumber	PageNum...	LineNumber
1	C.	1	5	1
2	Nondum	1	5	1
3	declinis	1	5	1
4	quamvis	1	5	2
5	praela	1	5	2
6	musta	1	5	3
7	Ornyte	1	5	4
8	vaccae	1	5	4
9	molle sub	1	5	5
10	hirsuta	1	5	5
11	explicuere	1	5	5
12	genista	1	5	5
13	succedimus	1	5	6
14	umbris	1	5	6
15	O.	1	5	8
16	Corydon	1	5	8
17	nemus	1	5	8
18	antra	1	5	8
19	ista	1	5	9
20	graciles	1	5	9
21	denset	1	5	9
22	rapidoque	1	5	10
23	caput	1	5	10
24	bullantes	1	5	11
25	ubi	1	5	11
26	protegit	1	5	12
27	errantibus	1	5	12

New Variant

VID	Name	LID	WID	Type	Distance
1	O.	1	27	3	0
2	O.	1	14	3	0
3	O.	1	21	3	0
4	O.	1	31	3	0
5	O.	1	52	3	0
6	C.	1	7	4	0
7	C.	1	59	4	0
8	C.	1	62	4	0
9	C.	1	37	4	0
10	C.	1	98	4	0
11	C.	1	102	4	0
12	nundum	2	7	3	0
13	nundum	2	59	3	0
14	declivis	3	7	3	0
15	declivis	3	10	3	0
16	declivus	3	59	3	0
17	declivus	3	95	3	0
18	declives	3	31	3	0
19	declinis	3	3	4	0
20	declinis	3	175	4	0
21	declinis	3	106	4	0
22	quatinus	4	37	3	0
23	quatinus	4	46	3	0
24	quatinus	4	49	3	0
25	quatinus	4	55	3	0
26	quatinus	4	72	3	0

NewCollation

CID	Location	LID	WID
1	C1.5.1	1	7
2	C1.5.1	1	59
3	C1.5.1	1	62
4	C1.5.1	1	37
5	C1.5.1	1	98
6	C1.5.1	1	102
7	C1.5.1	3	3
8	C1.5.1	3	175
9	C1.5.1	3	106
10	C1.5.4	7	175
11	C1.5.4	7	150
12	C1.5.4	7	105
13	C1.5.4	7	106
14	C1.5.4	8	27
15	C1.5.4	8	40
16	C1.5.4	8	24
17	C1.5.4	8	14
18	C1.5.4	8	52
19	C1.5.4	8	11
20	C1.5.4	8	49
21	C1.5.4	8	21
22	C1.5.5	9	59
23	C1.5.5	9	10
24	C1.5.8	16	40
25	C1.5.8	16	27
26	C1.5.8	16	18
27	C1.5.8	16	54
28	C1.5.8	18	3

Figure 4.5: XML data transformed into lemma, variant, and collation tables.

show plain text the data can have an additional column called "EncodingType" that would contain text. The designer can use this information to encode the data as text. This approach would allow for a straightforward projection design. Figure 4.7 shows the Improvise declarative language that maps data items to graphics. The first case is for "title." If the data attribute "type" is title, then add title data graphic with text and align to center. The text has foreground color set to black, background color set to transparent, font set to dialog 18 bold, text set to data attribute "data," and underline and strikethrough set to false. The title graphic is aligned to center. The result is shown in Figure 4.6. In the figure, the table view has the first data item type as "title" and data as "T. CALPURNI SICULI." In the text view, the data item is shown as title and aligned to center.

An example visualization of the Calpruni Siculi poem is shown in Figure 4.6. The visualization tries to recreate the original document with additional information. A line under a word represents a lemma that has a variant. Selected data items are represented in bold.

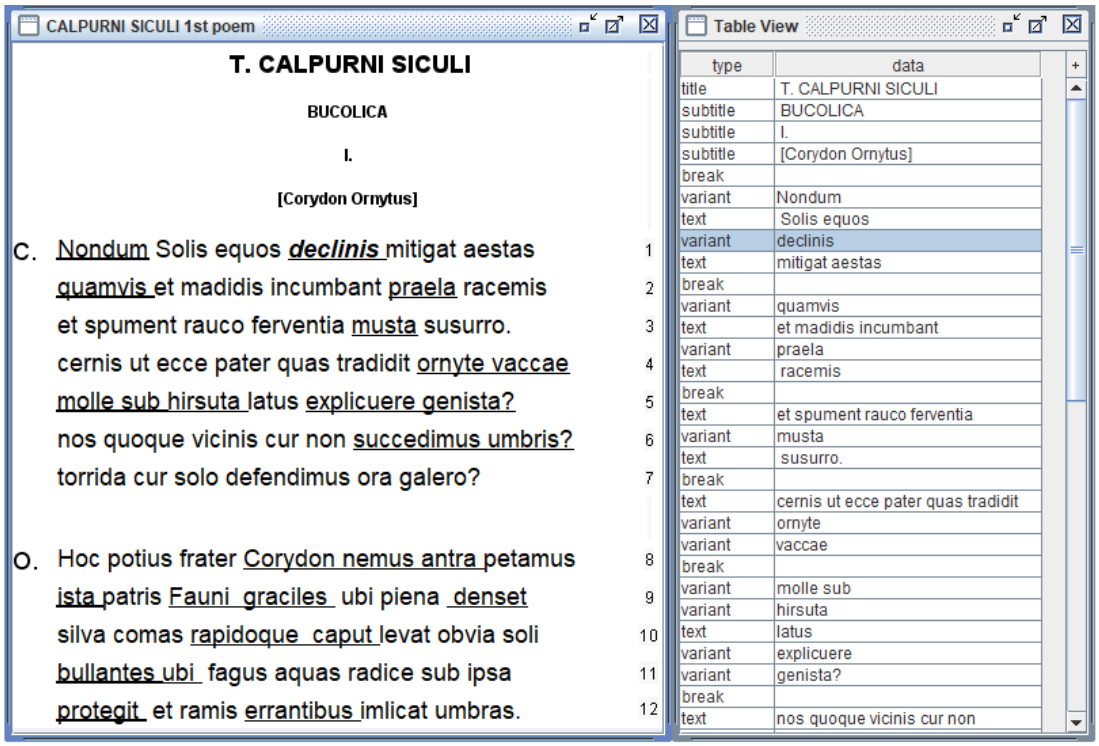


Figure 4.6: Visualization of lines 1 to 12 in the first poem of Calpurnius Siculus. On the right is the data that was used for this visualization.

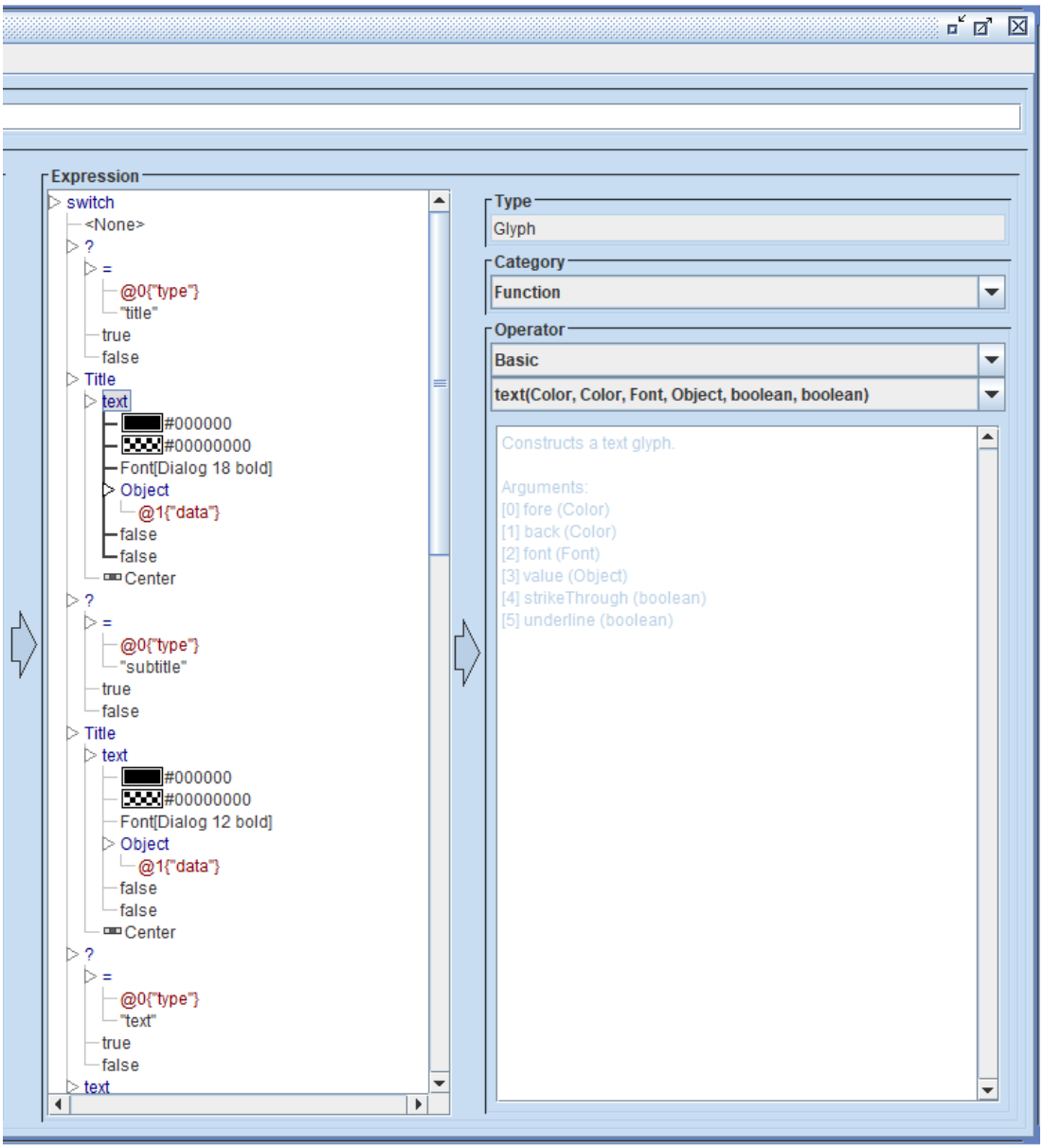


Figure 4.7: An example of mapping data attributes to text view graphics in the Improvise declarative language. The expression column shows the mappings. In this example, a switch statement is used to identify the type of encoding.

Chapter 5

Case Study

This case study is focused on the tasks that scholars perform when working with classic Latin texts. Classic scholars aim to create critical editions by analyzing the texts and their associated data. Classic Latin texts would be a good case study for this tool, as the data often involves large textual documents which require complex visual encoding and data mapping techniques to gain useful information from them.

For my thesis, I used the findings from the information study conducted by Dr. June Abbas, to assess the framework against tasks performed by classic scholars and assessed the goals set in the framework design. The Mellon information study conducted a survey with various user groups who work with Latin texts to analyse various tasks that they perform. The goals of this study were to gain a deeper understanding of how user groups who work with Latin interact with the text and the processes they follow when working on a critical edition, determine the usage of critical editions, and gather a list of resources that the user groups used. This framework is designed to support scholars who work on creating and editing critical editions and users that interact with these text documents. The

processes followed by the humanity scholars can be linked to the sense-making process in the Pirolli and Card model [18]. Each of the tasks will be analyzed against this framework to assess the abilities of this tool.

This chapter gives an introduction to classic Latin texts, an example of a printed edition with a description of each part, and the research process followed by humanity scholars. In the next section, we list the tasks that classic Scholars perform and assess the capability of this framework against each task. Finally, we conclude with a summary of the case study.

5.1 Background on Classic Latin Texts

Old Latin texts were primarily copied by scribes from the original text to a manuscript. Due to the manual nature of the process, errors and variations in text were introduced. The copied manuscript is further replicated to multiple copies introducing new errors and variations. Over time, the manuscripts are printed and published as copies, but the original manuscript is discarded or lost. Due to this, what usually remains is copies of the manuscript that vary significantly from the original text. The manuscripts and early printed versions are called *witnesses* and the textual variant copies are called *variants*. Scholars try to recreate the original text by verifying and comparing the different versions of text. They aim to recreate the original text as closely as possible. The recreated text is called a *critical edition*.

5.1.1 Printed Edition

The main components of the document are the base text and the critical apparatus. The *critical apparatus* holds the variants information which are found in

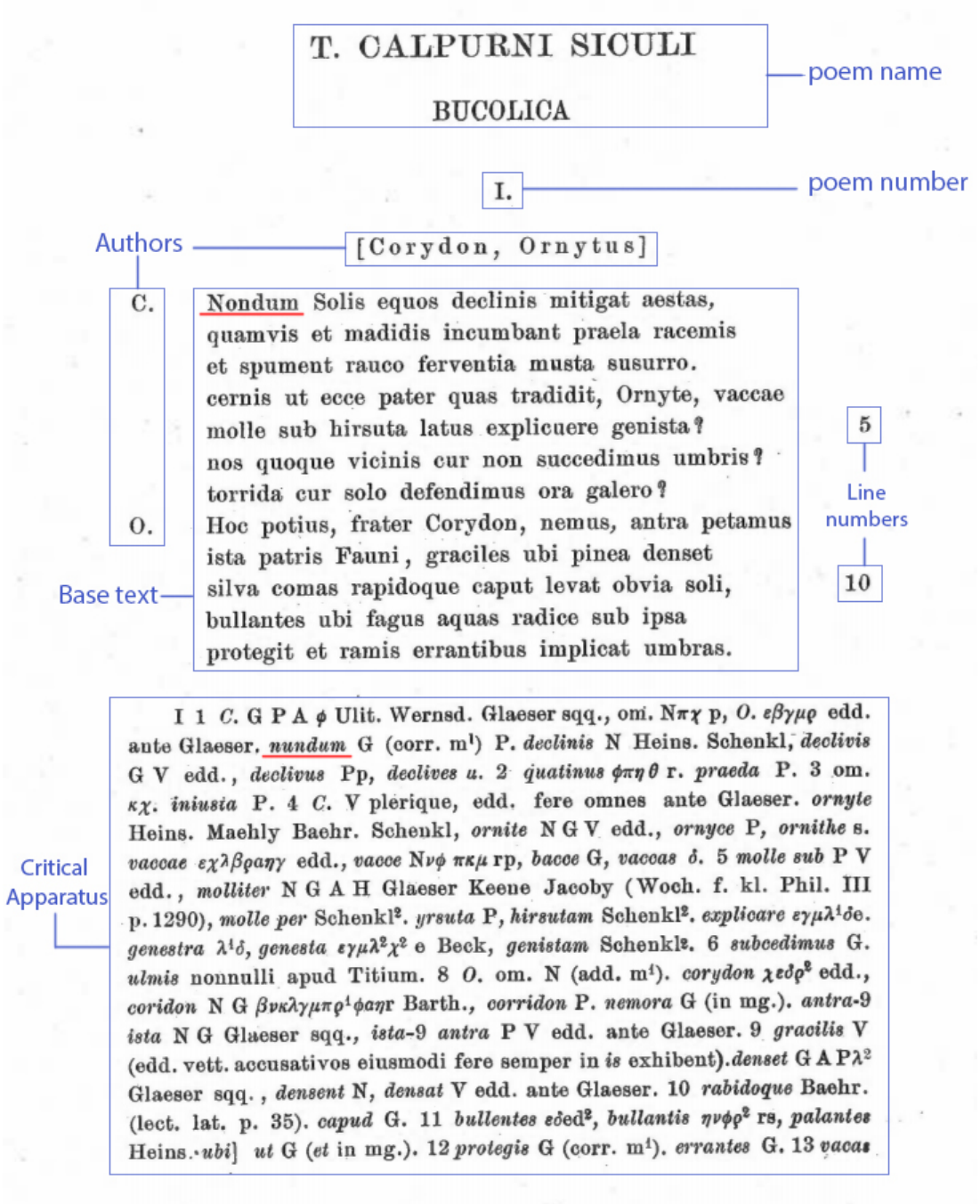


Figure 5.1: Giarratano's critical edition of the classical Latin poem Calpurnius Siculus [20].

various other copies along with their author information. A sample image of the printed document is shown in Figure 5.1. A *lemma* (e.g., underlined in the base text, *nondum*) is a part of the base text that is referred to in the critical apparatus. The critical apparatus of a lemma describes the variant, the witness for that variant and the type of variant. Additional information in the printed document include the poem name at the top, followed by the poem number and authors. The authors are also associated with the paragraph by shorthand notations on the left side margin of the base text. Line numbers are added on the right side margin of the base text for easy referencing.

5.1.2 Humanities Research Process

Humanities scholars generally initiate their research process by choosing a literature and building upon that by looking for relevant materials. From the relevant set of documents, the scholars analyze patterns and anomalies between these sets of documents to draw conclusions. The analysis can include looking for variations of text between multiple authors, identifying words that are similar and different between multiple authors, identifying similar word usages by the same author, and correcting mechanical and spelling errors in documents. Using the gathered information, the scholars construct the text to use in a critical edition and arrange the critical apparatus.

5.2 Information Study

An information behavior study was conducted by members of the Digital Latin Library[1] project under the Mellon funded grant project. The study was conducted

[1] `http://digitallatin.org`. Accessed: 2017-07-31.

by Dr. June Abbas, Professor at the University of Oklahoma. The development of this framework was funded under the Andrew W. Mellon Foundation. This framework, along with a couple of other text visualization techniques like Pixel-based visualization [3, 4] and Storyline visualization [21, 22] were designed to be a part of a desktop application for use by classic Latin scholars.

Abbas et al. have captured a detailed analysis of this study [2]. We used the findings of this study to gather the processes a potential end user of this application might perform. The user groups or participants of this study included classic scholars who focus on Latin, Graduate PhD students who are studying classics in Latin, and high school teachers who teach Latin. The study identified processes, analysis methods and tools used by user groups who work with classic Latin texts. The information study identified a set of tasks that the classic scholars perform.

1. Decide the author and manuscript for their preliminary analysis in developing a critical edition.

2. Find manuscripts and documents related to the primary manuscript.

3. Form collation. Collation involves comparing multiple documents and noting down differences, similarities, errors, and discrepancies between texts.

4. Review collation table. Move erroneous entries to another table for further inspection, remove spelling and mechanical errors.

5. Develop stemma. Visualize relationships between manuscripts and documents. Scholars find text that are close to the original (archetypal text) and eliminate documents that may not contribute to the original text.

6. Move from archetypal text to original. Review manuscripts and printed editions from other scholars to look for patterns and word uses to form original text.

7. Reconstruct critical edition. With the information from archetypal text and collation table, form the critical edition.

8. Construct critical apparatus. Encode witness and text variation information.

9. Condense critical apparatus. Rewrite entries in the apparatus to save space. This often involves noting only part of text that varies, replacing manuscript references with special IDs and removing unwanted text.

10. Develop preface and conspectus. Compile list of sources used in the document.

The list of tasks from this study [2] gives us the ability to assess this framework.

5.3 Task Analysis and Validation

The information study conducted by Abbas et al. gathered insight into the process classic scholars performs when developing critical editions. The framework will be assessed based on its ability to accomplish each of the tasks.

The application can help scholars identify primary and related manuscripts, form a collation table, develop stemma to visualize the critical edition and critical apparatus. These tasks involve reading and interpreting the text and its associated data. The application is capable of representing text and data graphics

in traditional form. This system will be analyzed against the tasks that classics scholars perform when working with classic Latin texts.

1. *Decide the author and manuscript to develop critical edition.* Deciding author and manuscript is a cognitive process. The scholars would need to read and analyze the various documents before deciding. One of the ways the application can support the scholars is by providing documents by details-on-demand. For example, a visualization can be created to list all available documents (table view or list view). On selecting a document, the complete text of the document can be rendered in the text view. The scholars can quickly switch between multiple documents to make better decisions.

2. *Look for related manuscripts.* Similar to deciding on the manuscript, the text view can be utilized to show all the relevant documents to the scholars for reading.

3. *Form collation table.* Forming a collation table requires reviewing all printed documents and noting differences between the manuscript and each printed document. The framework does not directly support comparison between texts. However, with manual help, the application can aid the process. For example, scholars can use two spatially separated text views, one showing the manuscript and another the selected printed edition. The user can manually select text from the printed edition that is different from the original text. Location, lemma ID and witness ID can be extracted from the selected item in the text view and added to the tabular data.

4. *Review collation table.* A collation table is usually represented in tabular form as it contains unstructured text. This framework does not directly

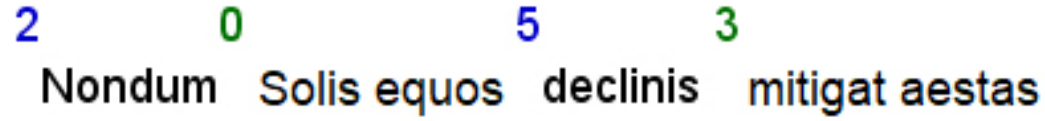


Figure 5.2: Screenshot of a visualization designed using the text view framework, showing the number of variants and witnesses for lemmas "nondum" and "declinis". For instance, lemma "nondum" has two variants and zero witnesses.

support reviewing collation tables. However, similar to forming a collation table, the original manuscript and the selected text can be shown again to the user to remove entries with spelling and mechanical errors. However, this method may not be as efficient as reviewing a tabular view.

5. *Develop stemma.* One of the primary features that classic scholars require is to identify word variants and the witnesses associated with each variant. In this tool, the designer has the capability to mark texts with variants by using various text visual channels or by adding additional data views to the text. Some of the ways this can be accomplished are as follows.

    - Compound encoding can be used to differentiate words with variants. The word itself can be added to the center of the compound encoding. The count of variants for that word can be added to top-right. Additional data such as the number of witnesses can be added to the top-left. Color can be used to differentiate between variants and witness counts. An example is shown in Figure 5.2. The word is displayed in the center, the count of variants is shown at the top left of the word in blue and the count of witnesses for the word is show on the top right corner in green.

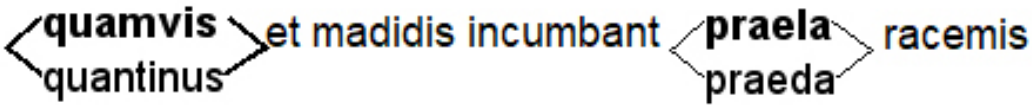


Figure 5.3: Screenshot of an example showing two lemmata using variation encoding. The lemmata themselves are shown in bold.

- The variation encoding is designed for this specific use case. It can list the set of variants in a top-down list fashion, with a link to the base line of the text to show direction. An example is shown in Figure 5.3. In the figure the base text is highlighted in bold and its variants are shown below it. The word "quamvis" has a variant "quantinus" and the word "praela" has a variant "praeda." Additionally, an compound encoding containing the text and the number of witnesses can be added as a list to the variation encoding.
- Words with variants can be marked with an underline. An example is shown in Figure 4.6.

6. *Move from archetype text to original text.* Moving from archetype text to original requires reviewing manuscripts and documents from other scholars. The findings from other scholars can be visualized in a separate text view and manually compared with the main text view.

7. *Reconstruct critical edition.* The text view framework does not support the ability to input text from the user. This framework cannot help in reconstructing critical edition.

8. *Construct critical apparatus.* Constructing critical apparatus requires text input from the user which is not supported by this framework.

9. *Condense critical apparatus.* Condensing critical apparatus requires rewriting parts of the critical apparatus and removing unwanted texts. This framework does not support editing text.

10. *Develop preface and conspectus.* The text view can list the sources used to form the critical edition. However, a table view might be better suited for this task.

5.3.1 Analysis of Goals

In this section we will analyze the goals set by this framework design. The list of goals and their analysis:

- *Use traditional text layout to show text and embed data graphics.* As shown in Figures3.6 and 3.3, we can see that this framework uses traditional text layout and embeds graphics in it.

- *Provide variety of data graphics representations.* Figure 3.6 shows the various encoding that this framework is capable of. The encodings shown in the figure are text, primitive shapes, compound, title, variation and tristate. Encodings like compound and variation can be used to form unique encodings as they take more than one primitive shape or text as input.

- *Provide multiple visual channels for the visualization designer to modify the appearance of the text and its associated graphics.* Visual encodings have multiple visual channels. For example, in Figure 5.2, text has different colors. In Figure 3.5, the background color of text has been changed. This shows that text can change it foreground and background color. Also,

nested views can take more than one primitive shape or text graphic as input. This can be used to form a variety of different data graphic patterns.

- *Allow for additional context information like header, footer, and line numbers in margins.* Figure 3.6 shows line numbers in right margin, paragraph context in left margin and headers in center panel. Footer is not shown in the image, but it can be created by adding title encoding at the end of the page.

- *Allow for coordinated multiple views.* This framework is built into Improvise. Improvise has a variety of views that can form coordinated multiple views.

- *Allow for navigation within the view.* The framework implements ScrollableJPanel to allow for multi-view scrolling. Scrolling is the primary form of navigation within this view.

- *Allow selection interactions.* Three types of selection has been implemented to select and interact with data in the view.

5.4 Summary

The primary goal of the Digital Latin Library[2] is to help classic scholars create, publish, and collaborate with other scholars who work on critical editions of Latin texts. The information study conducted by Abbas et al. [2] aimed to study the process classic scholars use to develop critical editions. With the help of this study, this framework was validated against each task. This framework

[2] `http://digitallatin.org` Accessed: 2017-07-31.

supports tasks that involve cognitive thinking by visually representing text and its associated data in a traditional layout.

Chapter 6

Conclusion and Future Work

This thesis introduced a new text visualization framework that uses the traditional text layout to show text and its associated data. This framework draws inspiration from existing principles and established techniques that are related to text visualization. In this framework, we have designed and implemented a fixed layout design to visualize text. We have provided a substantial set of visual encodings needed to design data graphics. We implemented various selection interactions within the view and linked selections to Improvise. We anticipate that this framework gives visualization designers the necessary tools to visualize text and its associated data to analyze and find patterns efficiently. Combined with the querying and interactive properties from Improvise, we believe that this framework would benefit scholars and users who work with text. There are several future directions to improve upon this visualization.

- *Interaction in embedded data views*: Interaction in embedded data graphics that take primitive and text visual encodings as input is currently not implemented. This feature would allow for multiple layers of interaction.

- *Inbuilt text searching*: Since our visualization represents text in a traditional format, we think the visualization could greatly benefit from an inbuilt text search feature. This would allow the user to search for a keyword directly within the visualization without depending on data queries.

- *Text highlighting inside data graphics*: Text inside data graphics cannot be selected in a traditional fashion as it is converted to Icons.

- *Variety in data graphics*: This framework would benefit from variety of new embedded data graphics. Complex data graphics gives designers multiple choices to visualize data.

- *Declarative layout design*: The current layout is fixed to the three column structure. Implementing custom layouts that can be defined by the visualization designer will provide more flexibility.

- *Text copy to clipboard*: Copying text is not supported due to the implementation strategy. It would be desirable to end users to be able to copy text to the clipboard.

This thesis contributes:

- a design framework that uses a traditional text layout to embed text and data graphics inline,

- a variety of text styling properties and a set of data graphic design options for use by the visualization designer,

- an implementation of the design framework that uses the Improvise visualization environment for visual encoding, data processing, interaction and querying,

- an application of the design framework to recreate and enhance the traditional printed layout in critical editions used by classic Latin scholars, and

- an assessment of the ability of this framework to support cognitive tasks performed by classic Latin scholars.

Although this application was designed primarily for use by Latin scholars, the framework can be applied to other textual data sets. We believe that analysts who work with large textual documents will benefit from this framework.

Bibliography

[1] J. Abbas, S. R. Baker, S. J. Huskey, and C. Weaver. Digital Latin Library: Information Work Practices of Classics Scholars, Graduate Students and Teachers. In *Proceedings of the Annual Meeting of the Association for Information Science and Technology (ASIS&T)*, St. Louis, MO, November 2015.

[2] J. Abbas, S. R. Baker, S. J. Huskey, and C. Weaver. How I Learned to Love Classical Studies: Information Representation Design of the Digital Latin Library. In *Proceedings of the Annual Meeting of the Association for Information Science and Technology (ASIS&T)*, Copenhagen, Denmark, October 2016.

[3] B. Asokarajan. A Pixel-Based Focus+Context Technique for Visualizing Variation in Classical Text. Master's thesis, University of Oklahoma, May 2016.

[4] B. Asokarajan, R. Etemadpour, J. Abbas, S. Huskey, and C. Weaver. Visualization of Latin Textual Variants using a Pixel-Based Text Analysis Tool. In *Proceedings of the Eurographics / IEEE-VGTC Conference on Visualization (EuroVis)*, Barcelona, ES, June 2017. IEEE.

[5] R. Borgo, J. Kehrer, D. H. Chung, E. Maguire, R. S. Laramee, H. Hauser, M. Ward, and M. Chen. Glyph-based Visualization: Foundations, Design Guidelines, Techniques and Applications. *Eurographics State of the Art Reports*, pages 39–63, May 2013. `http://diglib.eg.org/EG/DL/conf/EG2013/stars/039-063.pdf`.

[6] D. H. Chung, P. A. Legg, M. L. Parry, R. Bown, I. W. Griffiths, R. S. Laramee, and M. Chen. Glyph Sorting: Interactive Visualization for Multi-dimensional Data. *arXiv preprint arXiv:1304.2889*, 2013.

[7] J. H. Coombs, A. H. Renear, and S. J. DeRose. Markup Systems and the Future of Scholarly Text Processing. *Commun. ACM*, 30(11):933–947, Nov. 1987.

[8] A. Dillon. Reading from paper versus screens: a critical review of the empirical literature. *Ergonomics*, 35(10):1297–1326, 1992.

[9] J. Dimarco. *Computer Graphics and Multimedia: Applications, Problems and Solutions*. IGI Global, Hershey, PA, USA, 2004.

[10] J. Felici. *The Complete Manual of Typography*. Peachpit Press (Pearson Education), Berkeley, CA, 2003.

[11] P. Goffin, J. Boy, W. Willett, and P. Isenberg. An Exploratory Study of Word-Scale Graphics in Data-Rich Text Documents. *IEEE Transactions on Visualization and Computer Graphics*, 2017.

[12] N. Ide and J. Véronis. *Text encoding initiative: Background and contexts*, volume 29. Springer Science & Business Media, 1995.

[13] W. Javed and N. Elmqvist. Exploring the Design Space of Composite Visualization. *Visualization Symposium, IEEE Pacific*, 00:1–8, 2012.

[14] K. Koffka. *Principles of Gestalt Psychology*. Routledge & Kegan Paul, 1955.

[15] M. Krstajia, M. Najm-Araghi, F. Mansmann, and D. A. Keim. Story Tracker: Incremental visual text analytics of news story development. *Information Visualization*, 12(3-4):308–323, 2013.

[16] N. McCurdy, J. Lein, K. Coles, and M. Meyer. Poemage: Visualizing the Sonic Topology of a Poem. In *IEEE Transactions on Visualization and Computer Graphics (Proceedings of InfoVis 2015)*, pages 439–448, Jan. 2015.

[17] R. Pettersson. Information Design Principles and Guidelines. *Journal of Visual Literacy*, 29(2):167–182, 2010.

[18] P. Pirolli and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of International Conference on Intelligence Analysis*, volume 5, 2005.

[19] J. C. Roberts. State of the Art: Coordinated Multiple Views in Exploratory Visualization. In *Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization (CMV 2007)*, pages 61–71, July 2007.

[20] T. C. Siculus and C. Giarratano. Calpurnii Et Nemesiani Bucolica., 1910.

[21] S. Silvia. VariantFlow: Interactive Storyline Visualization Using Force Directed Layout. Master's thesis, University of Oklahoma, August 2016.

[22] S. Silvia, R. Etemadpour, J. Abbas, S. Huskey, and C. Weaver. Visualizing Variation in Classical Text with Force Directed Storylines. In *Proceedings of the Workshop on Visualization for the Digital Humanities*, Baltimore, MD, October 2016. IEEE.

[23] Viegas, F. B., Wattenberg, and Martin. TIMELINES: Tag Clouds and the Case for Vernacular Visualization. *interactions*, 15(4):49–52, July 2008.

[24] W3C. XPath. https://www.w3.org/TR/xpath/, 1999. Accessed: 2017-07-31.

[25] M. Wattenberg and F. Viegas. The Word Tree, an Interactive Visual Concordance. *Visualization and Computer Graphics, IEEE Transactions on*, 14(6):1221–1228, Nov 2008.

[26] C. E. Weaver. *Improvise: A User Interface for Interactive Construction of Highly-Coordinated Visualizations*. PhD thesis, University of Wisconsin–Madison, Madison, WI, June 2006.