# CHISIO WEB: A WEB-BASED FRAMEWORK FOR CUSTOMIZABLE VISUALIZATION OF RELATIONAL INFORMATION

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Selçuk Onur Sümer

March, 2012

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Uğur Doğrusöz(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Ali Aydın Selçuk

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Tolga Can

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

# ABSTRACT

## CHISIO WEB: A WEB-BASED FRAMEWORK FOR CUSTOMIZABLE VISUALIZATION OF RELATIONAL INFORMATION

Selçuk Onur Sümer
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Uğur Doğrusöz
March, 2012

Graphs are widely used to represent complex relational information. Graph visualization is crucial for effective analysis of information. In simple graphs, nodes are generally considered as uniform-sized components and they cannot be nested. This is often not sufficient to visualize complex relationships, because relational information is often clustered or hierarchically organized into groups or nested structures.

There exist many free, open source software in the field of web-based graph visualization. However, none fully supports compound or clustered graphs. Moreover, customization provided by such software is often limited to the basic visual properties of nodes and edges. It requires a lot of effort to build an advanced customization of visual properties and interactive functionality with these software.

In this thesis, we introduce a free, open source, general-purpose, web-based graph visualization framework, named Chisio Web (ChiWeb). ChiWeb supports visualization, interactive editing and layout of both simple and compound graphs. ChiWeb is implemented in ActionScript language and based on Flare, which is an open source ActionScript library designed for data visualization.

ChiWeb is specifically designed for easy customization with respect to visualization and functionality. ChiWeb can be used as a library to create a custom graph visualization with an advanced application behavior for particular needs of a specific domain. The elements and functionality that can be easily customized with ChiWeb are: visual styles, controls for interactive events such as node creation, key and mouse functionality, context menus, toolbars, and inspector windows. Furthermore, ChiWeb's architecture allows easy integration of new graph layout algorithms.

*Keywords:* graph visualization, interactive graph editing, customizable software, compound graphs, relational information.

# ÖZET

# CHISIO WEB: İLİŞKİSEL BİLGİNİN UYARLANABİLİR GÖRSELLEŞTİRİLMESİ İÇİN WEB-TABANLI BİR ÇERÇEVE

Selçuk Onur Sümer
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Doçent Dr. Uğur Doğrusöz
Mart, 2012

Çizgeler, karmaşık ilişkisel bilgilerin gösteriminde sıklıkla kullanılmaktadır. Çizge görselleştirmesi, bilginin etkin bir şekilde analizinde oldukça önemlidir. Basit çizgelerde, köşeler genelde aynı büyüklükte kabul edilir ve iç içe geçemezler. Bu durum, genellikle karmaşık ilişkilerin görselleştirilmesinde yeterli değildir, çünkü ilişkisel bilgi genelde kümelenmiş ya da gruplar veya iç içe geçmiş yapılar şeklinde hiyerarşik olarak düzenlenmiştir.

Web-tabanlı çizge görselleştirme alanında, ücretsiz, açık kaynak bir çok yazılım bulunmaktadır. Ancak, bunların hiçbiri bileşik ya da kümelenmiş çizgeleri tam olarak desteklememektedir. Üstelik, bu yazılımların uyarlanabilirliği, genelde köşe ve kenarların temel görsel özellikleriyle sınırlıdır. Bu yazılımlarla, görsel özellikler ve etkileşimsel işlevsellik açısından gelişmiş bir uyarlama yapmak çok fazla emek istemektedir.

Bu tezde, Chisio Web (ChiWeb) adında, genel amaçlı, web-tabanlı, ücretsiz ve açık kaynak bir çizge görselleştirme uygulama çatısı sunuyoruz. ChiWeb, hem basit hem de bileşik çizgelerin görselleştirilmesine, yerleştirilmesine ve etkileşimli olarak düzenlenmesine olanak sağlamaktadır. ChiWeb, ActionScript programlama diliyle gerçekleştirilmiştir ve Flare adında açık kaynak bir ActionScript kütüphanesini temel almıştır.

ChiWeb görselleştirme ve işlevsellik yönünden kolay uyarlanabilir olması için özel olarak tasarlanmıştır. ChiWeb kütüphanesi, belirli bir alana özgü ihtiyaçlar doğrultusunda özelleşmiş bir görselleştirme ve gelişmiş bir uygulama işlevselliği yaratmak için kullanılabilir. Görsel biçimler, köşe yaratmak gibi etkileşimli eylemler için denetimler, klavye ve fare işlevselliği, menüler, araç çubukları ve

denetim pencereleri, ChiWeb ile kolayca özelleştirilebilecek işlevsellik ve öğeler arasındadır. Bunun yanı sıra, ChiWeb'in mimarisi yeni yerleştirme algoritmalarının kolaylıkla eklenmesine izin verecek yapıdadır.

*Anahtar sözcükler*: çizge görselleştirme, etkileşimli çizge düzenleme, uyarlanabilir yazılım, bileşik çizgeler, ilişkisel bilgi.

# Acknowledgement

I would like to express my gratitude to my supervisor Assoc. Prof. Dr. Uğur Doğrusöz for his guidance and advices during the development of this thesis. It was a unique experience to work with him. I have learned so much from him during my graduate study.

I would like to thank to Assist. Prof. Dr. Ali Aydın Selçuk and Assoc. Prof. Dr. Tolga Can for reviewing and commenting on the manuscript of this thesis.

Thanks to Alper Karaçelik and other members of the i-Vis Research Group of Bilkent University. It was a privilege to be part of such a talented team. Also, thanks to Ethan Cerami and the Sander Research Group of MSKCC for giving me a chance to participate in the cancer genomics research.

Special thanks to Salim Arslan, for his contribution to the graphical component design of the ChiWeb project.

Last but not least, I would like to thank my parents Sadiye and Erkan for their endless love and patience. I would not be able to complete this thesis without their support.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

A *graph* is a collection of *nodes* (or *vertices*) and *edges* where a node (or a vertex) is an abstract representation of an object and an edge is a link between a node pair. A *compound graph* is an extended form of a *simple graph*. In a compound graph, a node can contain other nodes and edges, hence it is possible to construct nested structures with a compound graph. However, nodes cannot be nested within each other in simple graphs.

Graphs are widely used to represent complex *relational information*. This allows topological and functional analysis of complex networks by using *graph theory* concepts [6, 24, 11], and helps calculation or prediction of structural and dynamic properties defined by these networks. Since relational information is often clustered or hierarchically organized into nested structures, it is more appropriate to represent relational information with compound graphs.

It is a fast and easy way to represent information textually, but it is often hard to understand and analyze textual information (Figure 1.1). *Information visualization* plays a critical role in understanding of any kind of information. Visualization allows us to explore, observe, and understand large amounts of information at one sight, since it takes advantage of human eye's broad bandwidth pathway [26].

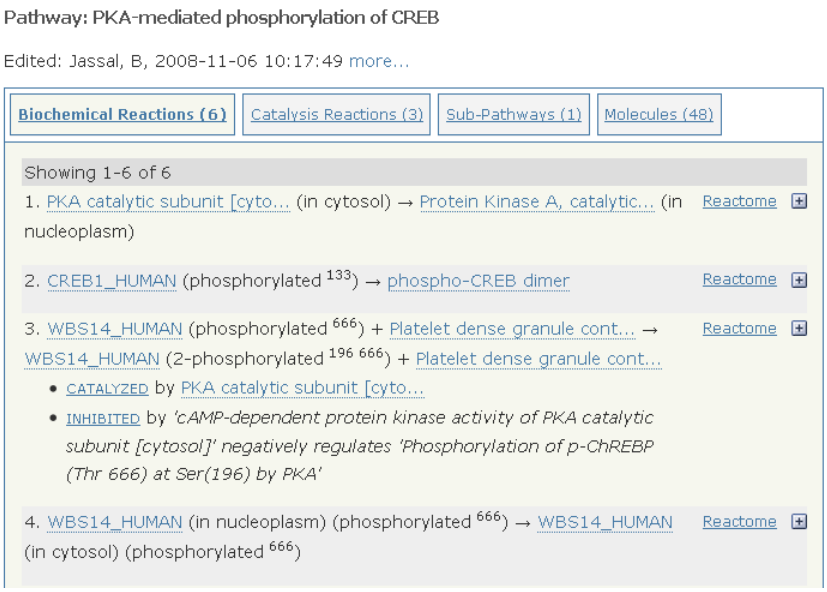


Figure 1.1: Textual representation of a biological pathway (PKA-mediated phosphorylation of CREB) in PathwayCommons [29]

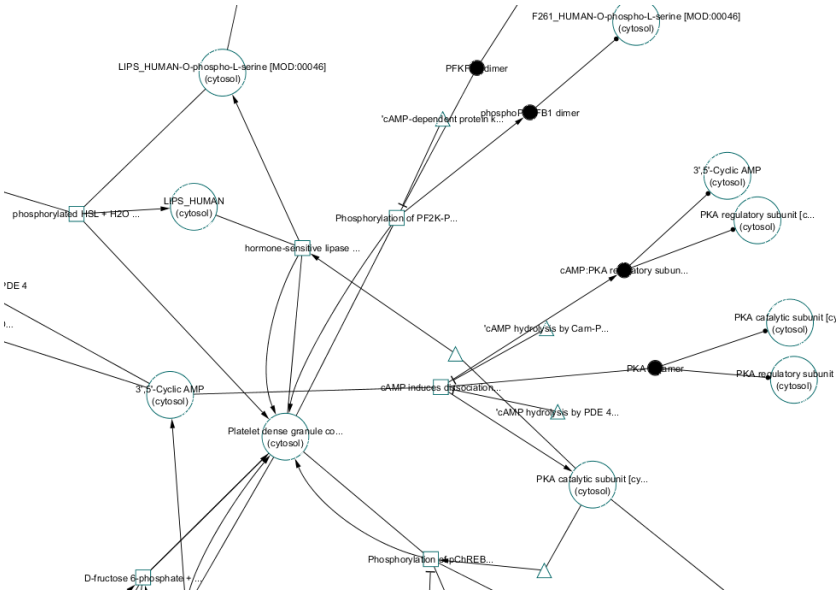


Figure 1.2: Visualization of a biological pathway (PKA-mediated phosphorylation of CREB) with a simple graph structure in Cytoscape [12]

Simple graph visualization is a powerful method that is widely used to represent information (Figure 1.2). However, in most cases, it is inadequate for a full comprehension of complex relational information. Compound graph visualization, on the other hand, is a more capable way of representing complex relational information (Figure 1.3).

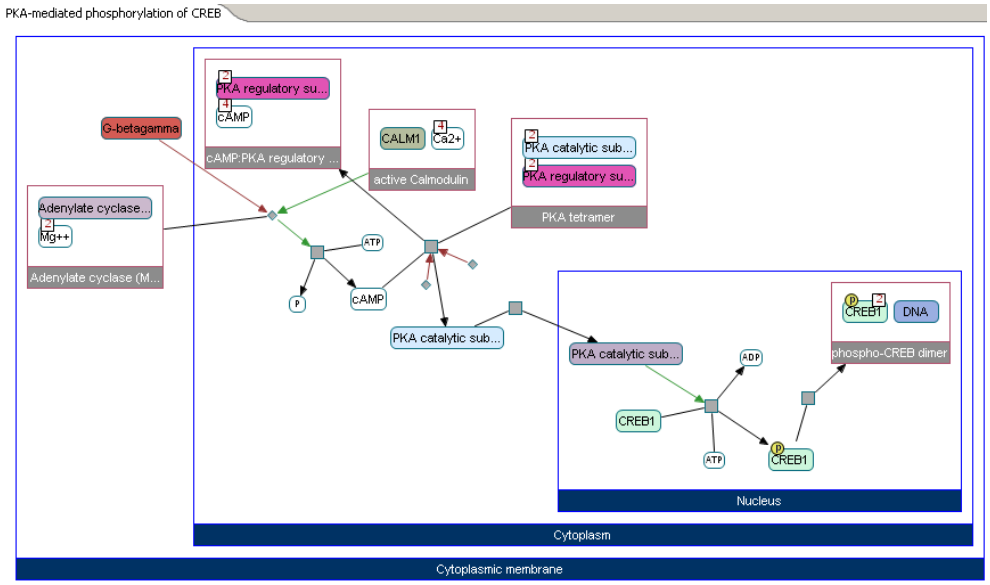


Figure 1.3: Visualization of a biological pathway (PKA-mediated phosphorylation of CREB) with a compound graph structure in CHISIO BioPAX Editor [5]

## 1.1 Motivation

Many graph visualization software have been developed within last two decades. While some of them are domain-specific [14, 25, 28, 35], there are also some general purpose graph visualization tools [2, 9, 12, 13, 20, 22, 23, 32, 33]. However, only a few of them support compound or clustered structures [9, 14, 32, 35], and none of the free, web-based software fully supports visualization and layout of compound graphs.

Web-based software are platform independent, and do not require end-users to install or update any software other than a web browser. Since it is easier to access and use, a web-based software can even reach an end-user with a limited

experience and knowledge in computers. Because of those advantages of web-based software, web-based software development have become more popular.

Easy *customization* is an important aspect of a graph visualization tool. A web-based graph visualization software with an effective customization mechanism should enable defining of arbitrary node and edge user interfaces (UIs). It should provide a way to customize functionality of the software with respect to interactive actions such as click, double-click, and hover. Easy integration of additional layout algorithms is also crucial for a customized graph visualization. Location and content of the drawing canvas, inspectors, and menus should also be customizable.

There are some web-based graph visualization tools [2, 13, 32, 35] that enable customization. However, only a few of them [13, 35] are open source and customization provided by these tools are limited to the basic visual properties of nodes and edges such as shape, size, color, and transparency. Advanced customization of visual properties and interactive functionality require a lot of effort with the existing general purpose, free, web-based software.

## 1.2 Results

Motivated to satisfy the requirement of a sophisticated software in the field of interactive visualization of relational data, a new project, called Chisio Web (ChiWeb), was initiated to develop a free, open source, general purpose, web-based graph visualization and editing framework. ChiWeb is implemented in ActionScript [1] and it requires installation of Flash Player [18], which is a light-weight plug-in available for almost all modern web browsers. ChiWeb is part of the Chisio project and the complete source code of the ChiWeb library together with a sample application demonstrating its usage can be accessed through the project's publicly available source code repository [10].

ChiWeb fills an important gap for compound graphs in the field of web-based graph visualization. With its compound graph support, ChiWeb is capable of

visualization and interactive editing of graphs with compound nodes. ChiWeb's compound graph support is also adapted to Cytoscape Web [13] which is another web-based graph visualization tool that has been previously supporting only simple graphs. Considering that Cytoscape Web is a widely used graph visualization software, this extension is another success for web-based compound graph visualization.

ChiWeb is designed for easy customization and provides various mechanisms to customize the visualization of the graph as well as the interaction control of the application. One can easily customize ChiWeb for particular needs of a domain-specific application. ChiWeb's easy customization support fills another important gap in the field of web-based graph visualization. Some of the elements and functionality that can be easily customized with ChiWeb can be listed as:

- Visual styles: Visual properties of nodes and edges such as shape, width, height, color, and opacity can be customized by defining visual styles. It is possible to define shared visual styles for certain groups of nodes or edges.
- Node and edge user interfaces (UI): By defining custom user interfaces it is possible to render nodes and edges in any desired form.
- Functionality and behavior: With its control customization mechanism, ChiWeb enables introducing advanced control over node and edge creation, and custom behavior for interactive actions such as drag, selection, and zoom. It is also easy to define custom behavior for basic mouse events such as click, double-click, and hover as well as basic key events such as key press, and key release.
- Graphical user interface (GUI) elements: GUI elements such as context menus, tool bars, buttons, input fields, and inspector windows can be customized with respect to their content, appearance, and layout.

The remainder of this thesis is organized as follows: Chapter 2 gives background information on graphs and graph visualization, and also introduces the base software on which ChiWeb is built. Chapter 3 compares ChiWeb to the other

web-based graph visualization software. Chapter 4 explains software architecture of ChiWeb in detail. Chapter 5 illustrates customization of ChiWeb with sample implementations. Chapter 6 provides implementation details of important components. Finally, Chapter 7 concludes the thesis and discusses possible future improvements for ChiWeb.

# Chapter 2

# Background

## 2.1 Graphs

A simple graph $G = (V, E)$ is a pair of sets $V$ and $E$ where $V$ is a set of *nodes* (or *vertices*) and $E$ is a set of *edges*.

An edge $e \in E$ is a node pair $\{u, v\}$ where $u \in V$ and $v \in V$. $u$ and $v$ are *endpoints* of an edge where $u$ is its *source* and $v$ is its *target*. An edge *joins* its endpoints.

A node $v$ is *incident* with an edge $e$, if $v \in e$. A node $u$ is *adjacent* to a node $v$, if there exists an edge $e$, where $u \in e$ and $v \in e$. Two adjacent nodes are called *neighbors*. The *degree* of a node $v$ is the number of its incident edges.

In a *directed graph*, an edge has a direction from its source to its target. In an *undirected graph*, on the other hand, there is no sense of direction for edges.

An edge is called a *loop* (or *self*) edge if both of its endpoints $u$ and $v$ are the same. *Multiple edges* are edges between the same source $u$ and the same target $v$.

In a *compound* (or *nested*) graph, a node $c$ may include other nodes and edges.

Such a node is called a *compound* node. Every node $v$ inside a compound node $c$ is a *child* of $c$, and $c$ is the *parent* of every node inside $c$. A child node inside a compound node $c$ may also be a compound node. Therefore, it is possible to define compound graphs with multiple levels of nesting (Figure 2.1).

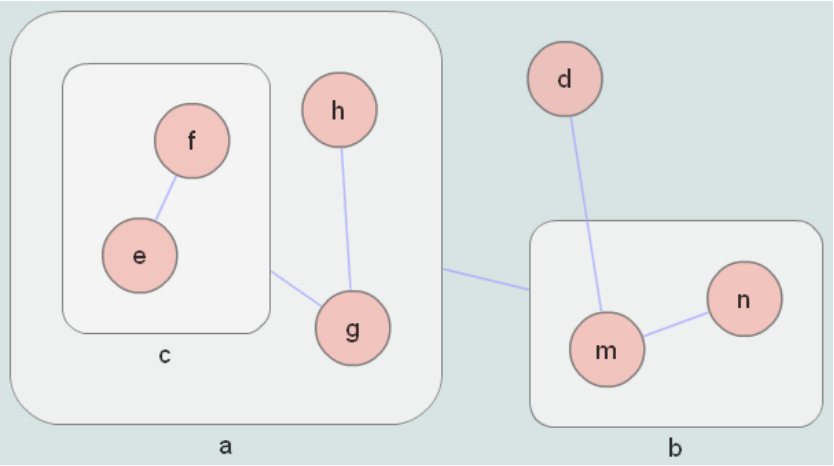


Figure 2.1: A compound graph $C$ with multiple levels of nesting

More formally, a compound graph $C = (V, E, F)$ consists of sets $V$, $E$ and $F$ where $V$ is a set of *nodes*, $E$ is a set of *adjacency edges* and $F$ is a set of *inclusion edges* [16]. While definition of an adjacency edge is the same as definition of an edge in a simple graph, an inclusion edge is to indicate a nesting relationship between two nodes. For the example compound graph $C$ in Figure 2.1:

$$V = \{a,b,c,d,e,f,g,h,m,n\},$$
$$E = \{\{a,b\},\{c,g\},\{d,m\},\{e,f\},\{g,h\},\{m,n\}\}, \text{ and}$$
$$F = \{ac,ag,ah,bm,bn,ce,cf\}.$$

An adjacency edge between two nodes of same nesting level is called an *intra-graph edge*. An edge between two nodes of different nesting level is called an *inter-graph edge*. For the compound graph $C$ in Figure 2.1, the edge $\{a, b\}$ is an intra-graph edge and the edge $\{d, m\}$ is an inter-graph edge.

## 2.2 Graph Visualization

Graph theory, as introduced in Section 2.1, provides *topological* information of the graph but it does not contain any information about graph *geometry*. In addition to topological structure, geometric information is also crucial to *visualize* a graph.

Basically, *graph visualization* is an act of drawing a graph by introducing geometric information for its nodes (such as coordinates, shape and size) and edges (such as edge width) in addition to its topological structure. Although visualization of a graph can be performed in a 3D environment, it is more common to draw graphs in a 2D coordinate system.

Graphical properties, such as color and transparency of a node or an edge, are also considered within graph visualization concept. Labels, which enable attaching additional textual or graphical information to a node or an edge, are also important elements of graph visualization.

*Layout* of a graph plays a critical role for an understandable visualization (Figure 2.2). Since layout is an aesthetic aspect, it is hard to find an optimum layout. For a better layout, there are some general criteria such as *planarity*, *total edge length*, and *symmetry*, but, each graph structure usually requires a different layout approach. Therefore, specific heuristics have been developed for graphs with certain topological and geometrical structures.

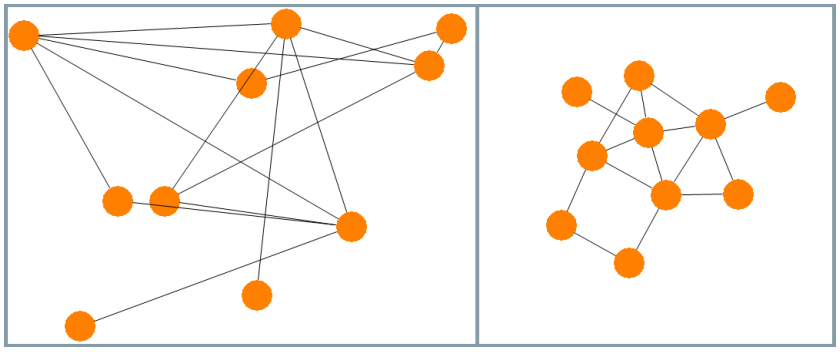

Figure 2.2: A random layout (left) and a force directed layout (right) for the same graph

*Interactive* graph visualization enables interactive editing of a graph in addition to its drawing and layout. An interactive graph editor allows changing both topology and geometry of a graph. While adding a node to or removing an edge from a graph changes its topology, changing coordinates of a node or increasing its size changes a graph's geometry. Improvement in graphical user interfaces (GUIs) and increase in number of software tools providing visual functions have made interactive graph editing an important concept for graph visualization [15].

## 2.3 Flare

Flare [17] is an open source ActionScript [1] library designed for general-purpose web-based data visualization. It has a basic support for visualization of simple graphs. It also supports basic interactive actions such as selecting and dragging nodes. Flare uses lightweight graphical components called *sprites* to display nodes and edges. This results in visualization of large graphs with less performance problems. The rest of this section describes the details of the architecture of Flare and its visualization mechanism.

### 2.3.1 Overview of Flare

In Flare, the data to be visualized consist of two default data groups: *nodes* and *edges*. In addition to these default data groups, it is also possible to define additional data groups (Figure 2.3). Section 2.3.2 and Section 2.3.3 describe the data concept of Flare in detail.

Flare visualizes the data by rendering the nodes with a shape renderer and the edges with an edge renderer. Section 2.3.4 explains the rendering process of Flare. In addition to basic rendering support, Flare also provides controls to handle actions such as selection and node dragging, and operators to perform tasks such as graph layout and labeling of nodes and edges (Figure 2.3). Section 2.3.5 gives the details of Flare's visualization support.

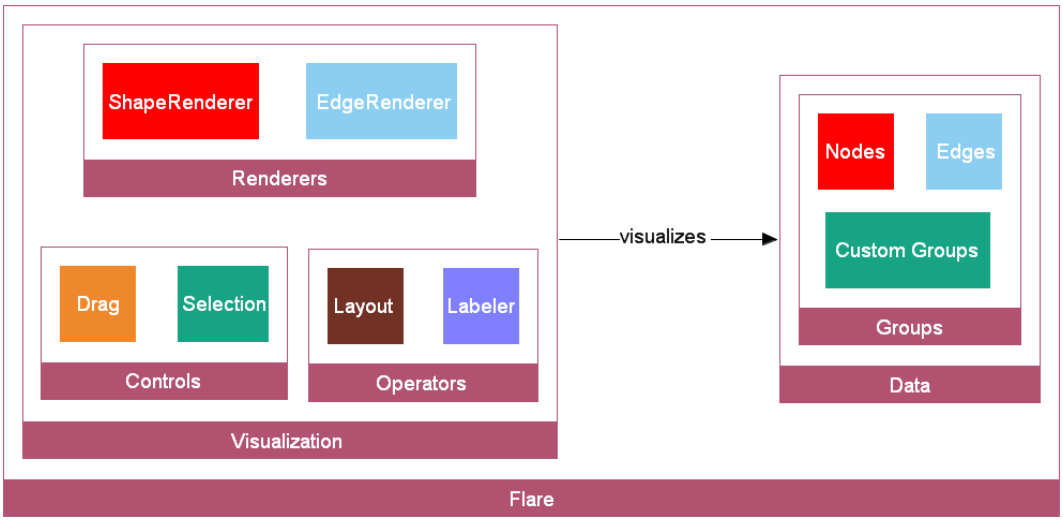


Figure 2.3: Overview of data visualization in Flare

### 2.3.2 Data Representation

Data to be visualized is represented by the `Data` class. By default, there are two data groups in `Data`: *nodes* and *edges*. For each data group, there is a `DataList` which contains a list of data elements (Figure 2.4).

Apart from the default data groups, it is also possible to define additional groups by using the `addGroup` function of `Data`. For each new group, a new `DataList` is added to the data. Additional data groups are useful to handle data in specific conditions. For example, creating a data group for hidden edges can help to simplify the graph view when it is desired by hiding certain types of edges without actually deleting them from the data.

### 2.3.3 Data Sprites

A data element in a `DataList` is represented by the `DataSprite` class. `DataSprite` inherits from `Sprite` which is one of the basic graphical components of the ActionScript platform. Therefore, a `DataSprite` represents both the *model* and the *view* of a data element.

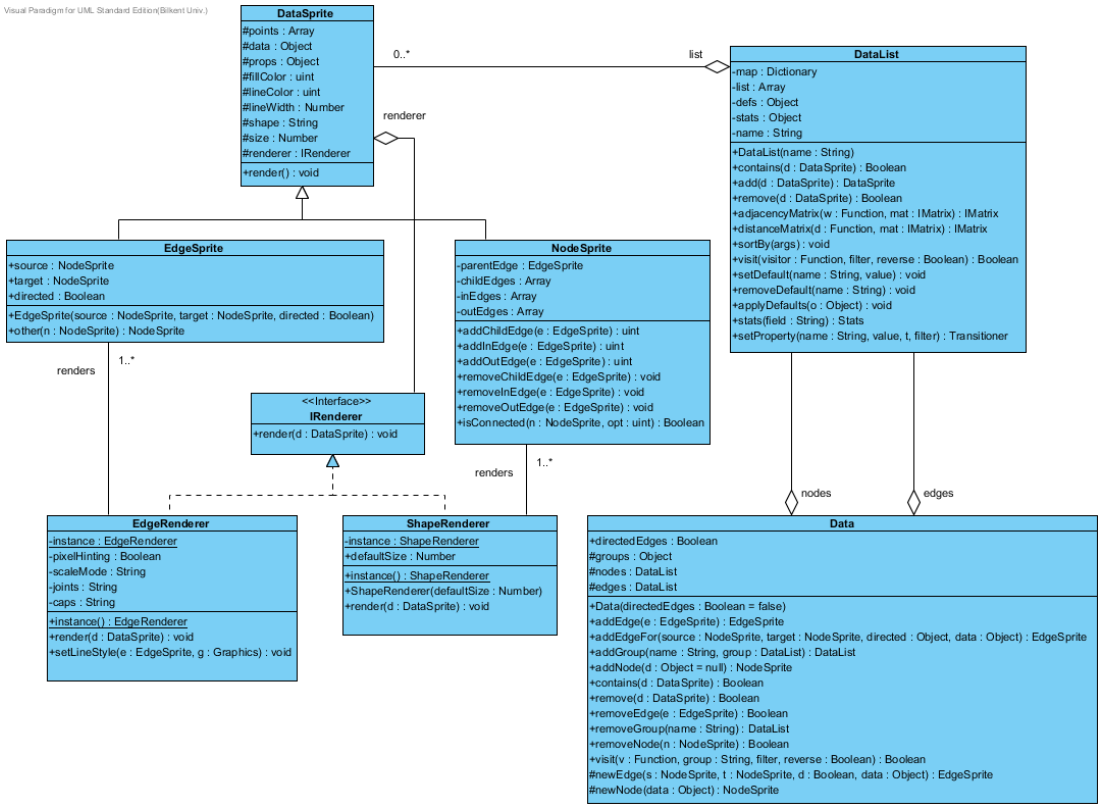


Figure 2.4: Data representation, data sprites and renderers in Flare

While the `data` field of a `DataSprite` is to store the actual data of an element, the field `props` is to attach additional properties which are not directly related to the actual data. Other fields such as `fillColor`, `lineColor`, `shape` and `size` are for the graphical attributes (Figure 2.4).

`NodeSprite` and `EdgeSprite` classes, both of which extend `DataSprite`, represent the node data and the edge data respectively. Both `NodeSprite` and `EdgeSprite` classes contain methods related to the graph topology for an effective traversal of the graph data (Figure 2.4).

### 2.3.4 Renderers

Since a `NodeSprite` and an `EdgeSprite` are both graphical components, it is required to render these sprites. By default, nodes are rendered by `ShapeRenderer` and edges are rendered by `EdgeRenderer`. In Flare architecture, both `ShapeRenderer` and `EdgeRenderer` are designed to be singleton [19] classes (Figure 2.4). As a result, a single `ShapeRenderer` renders all the nodes in the graph, and a single `EdgeRenderer` renders all the edges in the graph.

During the rendering process, a renderer uses the visual properties such as shape, size, and color of a `NodeSprite` or an `EdgeSprite` in order to display a data sprite with an appearance specific to it. `ShapeRenderer` assumes the `shape` property of a `NodeSprite` to be one of the predefined shapes. Therefore, a node having an unknown shape is rendered as it has the default circular shape. `EdgeRenderer` renders an edge by drawing it always from the center of its source node to the center of its target node without using the size or the shape of the source and the target.

### 2.3.5 Visualization

Flare has a visualization mechanism that allows interactive editing and layout of the graph. `Visualization` is the core class of this mechanism (Figure 2.5). It

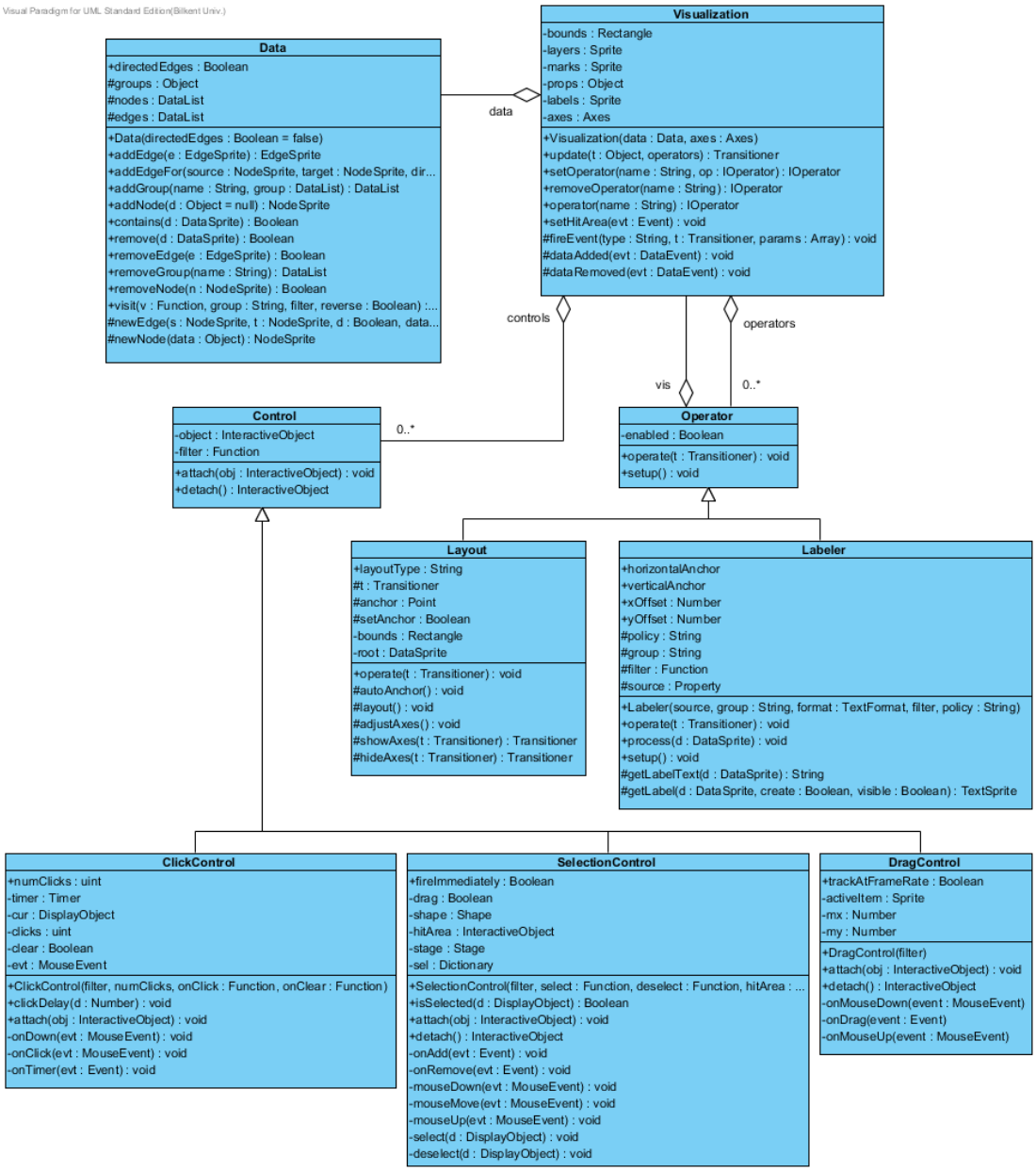


Figure 2.5: Visualization architecture of Flare

observes both topological and geometrical changes in the graph data, and reflects the changes to the graph view.

By the help of the control classes such as `ClickControl`, `SelectionControl` and `DragControl`, it is possible to handle interactive actions. Similarly, by the help of the operator classes such as `Labeler` and `Layout`, it is possible to perform processing tasks on the visualization content. Due to space limitations, Figure 2.5 illustrates only a few of those control and operator classes.

Flare supports defining of additional controls and operators. It is possible to add instances of these user-defined classes to the corresponding lists in `Visualization`. This allows customization of Flare's visualization mechanism.

## 2.4 Chisio

Chisio [9] is a framework that is specifically designed for interactive visualization and layout of compound or clustered graphs. Chisio is composed of two main parts: `ChiLay` (Chisio Layout) [8] and `ChiEd` (Chisio Editor) [7]. Figure 2.6 illustrates the overview structure of Chisio.

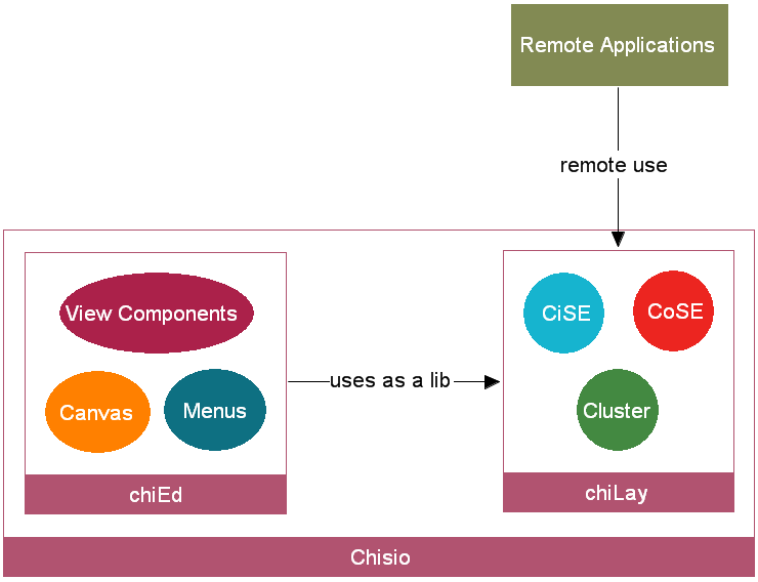


Figure 2.6: Overview of main Chisio parts

ChiLay is a graphic independent Java [27] library that can be used by another application to layout compound or clustered graphs as well as simple graphs. It provides layout algorithms CoSE [16] for compound graphs, and CiSE for clustered graphs. There are also several other layout algorithms, such as circular layout and Sugiyama [31] layout, available in ChiLay.

ChiLay can be used both as a local development library and as a remote layout service. In order to use the provided layout algorithms as a local library for a graphical application, it is required to construct a *layout level* topology from a *view level* topology [30]. In order to use ChiLay remotely, it is required to save the geometry of a graph in XML format that conforms to an XML schema specific to ChiLay (Figure 2.7), and to send the saved XML to a server, where ChiLay is deployed, as an HTTP request [30].

ChiEd is an interactive graph visualization tool developed in Java. It allows creating, editing, saving, and layout of both compound and simple graphs [34]. ChiEd locally uses ChiLay library for layout services.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<view>
  <node id="ID_0" clusterID="0">
    <bounds height="40" width="40" x="0" y="0"/>
  </node>
  <node id="ID_1" clusterID="0">
    <bounds height="40" width="40" x="0" y="0"/>
  </node>
  <node id="ID_2" clusterID="0">
    <bounds height="280" width="240" x="0" y="0"/>
    <children>
      <node id="ID_3" clusterID="0">
        <bounds height="40" width="40" x="0" y="0"/>
      </node>
      <node id="ID_4" clusterID="0">
        <bounds height="40" width="40" x="0" y="0"/>
      </node>
    </children>
  </node>
  <edge id="ID_0">
    <sourceNode id="ID_0"/>
    <targetNode id="ID_1"/>
  </edge>
  <edge id="ID_1">
    <sourceNode id="ID_1"/>
    <targetNode id="ID_2"/>
  </edge>
  <edge id="ID_2">
    <sourceNode id="ID_3"/>
    <targetNode id="ID_4"/>
  </edge>
  <edge id="ID_3">
    <sourceNode id="ID_0"/>
    <targetNode id="ID_4"/>
  </edge>
</view>
```

Figure 2.7: A sample XML that conforms to ChiLay's XML schema

# Chapter 3

# Related Work

Among the graph visualization software, there are also some tools developed for the web other than ChiWeb. While some of them are only to visualize the data, some of them enable interactive editing of the graph content. Table 3.1 compares ChiWeb to the popular web-based graph visualization software. The rest of this chapter discusses some of the notable work done in the field of web-based graph visualization.

| | Graph Editing | Compound Support | Layout | Customizability | Availability | Domain |
|---|---|---|---|---|---|---|
| TSV [32] | Yes | Yes | Predefined[1] | High | Commercial | Generic |
| Asterisq [2] | No | No | Predefined[1] | Medium[2] | Commercial | Generic |
| Cytoscape Web [13] | Yes | Yes[3] | Predefined[1] | Medium[4] | Open Source[5] | Generic |
| VisANT [35] | Yes | Partial[6] | Extendible | Low[7] | Open Source[5] | Specific[8] |
| WiGis [23] | No | No | Extendible | Low[9] | Open Source[5] | Generic |
| ChiWeb | Yes | Yes | Extendible | High | Open Source[5] | Generic |

[1] Additional layout algorithms cannot be integrated easily
[2] Partial support for custom renderers, limited customization for functionality
[3] An adapted version of ChiWeb's compound graph support
[4] Limited customization for visualization and functionality
[5] Free software and its source code is publicly available
[6] Partial support with grouping; nodes can be grouped, where groups are visually highlighted
[7] Limited to basic visual properties of nodes and edges
[8] Designed for visual analysis of biological networks and pathways
[9] No support for customized visualization and functionality

Table 3.1: ChiWeb compared to the popular web-based graph visualization software

## 3.1 Cytoscape Web

Cytoscape Web [13] is one the most capable graph visualization software among the open source, web-based, general purpose graph visualization tools. Cytoscape Web enables interactive editing of the graph topology and provides various graph layout options. Initially, Cytoscape Web was only supporting simple graphs, but with the adaptation of ChiWeb's compound graph support, now it also supports compound graphs (Figure 3.1).

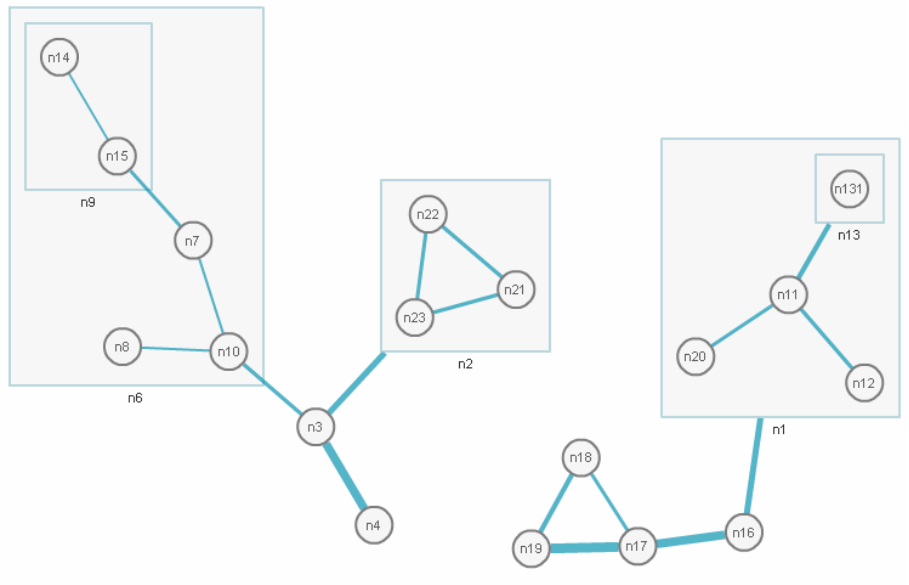


Figure 3.1: Compound Graph Visualization in Cytoscape Web

Cytoscape Web has a layered structure composing of three main components (Figure 3.2). The middle layer (JavaScript API) provides facilities for the upper layer (user-defined JavaScript layer) to create and manage visualization of relational data. By using the JavaScript API, one can create a graph visualization with custom visual styles for nodes and edges, customized menus and custom buttons. It is also possible to define custom behavior for interactive actions such as click, double-click, and hover on nodes and edges.

The advantage of the layered structure of Cytoscape Web is that it creates an abstraction between the top layer and the bottom layer (ActionScript layer). As a result, one can construct a visualization by using only a JavaScript component

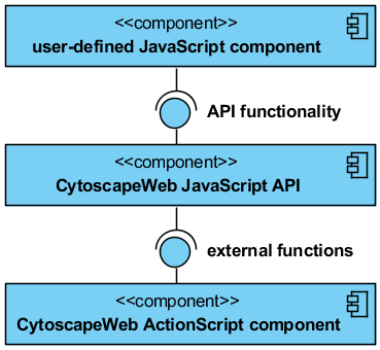


Figure 3.2: Layers of CytoscapeWeb

without dealing with ActionScript. However, this layered structure limits the customizability of Cytoscape Web. For example, rendering a node with multiple colors or introducing a custom graph layout algorithm is not possible through the API. The only way to achieve such an advanced customization is to modify the ActionScript layer. Since Cytoscape Web is not designed for it, this kind of customization requires a lot of effort and a good understanding of the architecture of ActionScript layer. Besides, it may also create backward incompatibility for the API layer and may produce undesired application behavior.

## 3.2 Tom Sawyer Visualization

Tom Sawyer Visualization (TSV) [32] is a commercial software development kit (SDK) for building data visualization applications. Probably, it is the most capable commercial software in terms of visualization and customization. TSV allows interactive editing of the graph content, it supports compound graphs, and by providing API libraries it enables customization of node and edge types as well as the context menu and other graphical user interface elements. TSV also provides various graph layout options, but it does not provide an API to easily integrate custom layout algorithms.

TSV has both desktop and web editions. Figure 3.3 illustrates a customized web application built with TSV SDK. Although TSV has a web development support via JSP and ASP.NET technologies, it is a commercial software and

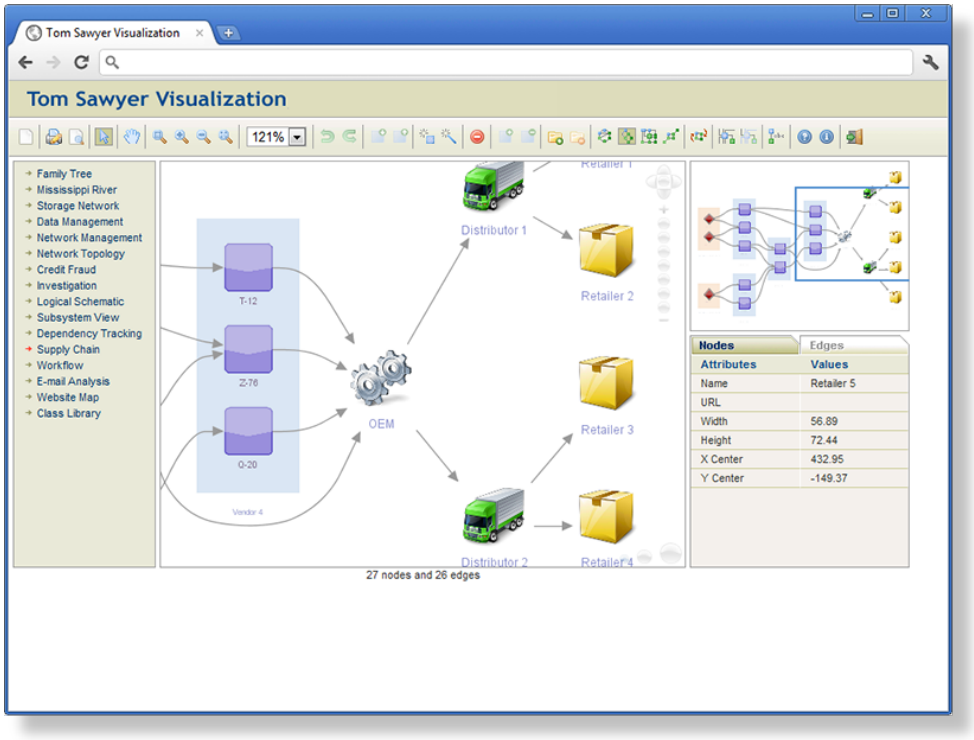


Figure 3.3: A sample application built on the web edition of TSV

designed for development of enterprise applications. Thus, it may not be an affordable solution for small research groups and open source web developers.

## 3.3 Asterisq

Asterisq [2] is another commercial web-based graph visualization software built on ActionScript technology. It consists of two different products: Constellation Custom and Constellation Roamer. For the listed features of Asterisq and for the sample screen shots provided, there is no sign of compound graph support either in Constellation Roamer or in Constellation Custom. There is also no sign of interactive graph editing by adding or removing nodes and edges in Asterisq products.

Since it is a commercial tool, basic customization of Constellation Roamer is only possible through the customization mechanism provided by the software

API [3]. Similar to customization of Cytoscape Web, it is only possible to customize visual properties of nodes and edges, and some configuration parameters through the provided API. It does not seem possible to customize the interactive behavior of the application or to add arbitrary shapes for nodes and edges other than predefined types. Figure 3.4 illustrates a visualization of a simple network in Constellation Roamer.

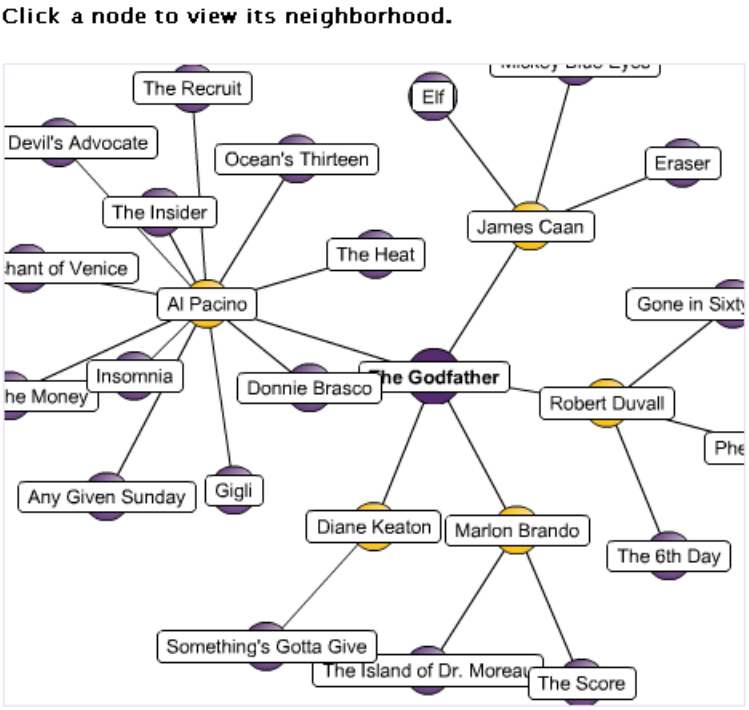


Figure 3.4: A sample view from the demo version of Constellation Roamer (Asterisq)

Constellation Custom, on the other hand, provides additional customization for rendering of nodes. It is possible to define a custom node renderer [4] to display a node with a custom view (Figure 3.5). Since the source code of the application is not publicly available because of commercial reasons, for the documented customization approach, it is not clear whether it is possible to define different renderers for different types of nodes or it is only possible to use a single renderer at a time.

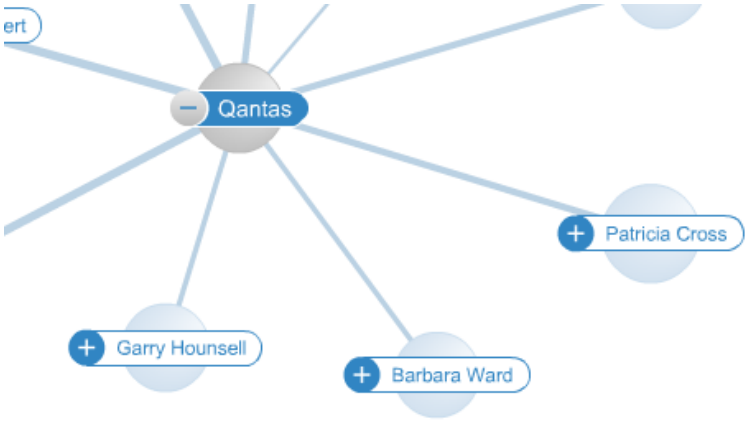


Figure 3.5: A custom node renderer for Constellation Custom that displays a plus sign when it has neighbors that are not being displayed

## 3.4 VisANT

VisANT [35] is another powerful graph visualization and editing software designed for visual analysis of biological networks and pathways. VisANT has both desktop and web-based versions. VisANT has support for grouping nodes. One can select a number of nodes to group as a component, it is also possible to create nested group structures which is similar to nested compound nodes. However, groups are not considered as nodes in VisANT, so it is not possible to define edges for groups. Figure 3.6 illustrates a sample visualization with nested node groups.

Because web-based version of VisANT is built on Java Applet technology, it requires installation of Java Runtime Environment (JRE) which has a longer installation process than Flash Player and can be considered as a heavyweight component compared to Flash Player. Being a domain-specific software for biological networks and pathways is another drawback of VisANT, because it is not easy to customize VisANT for general-purpose graph visualization or for another domain.

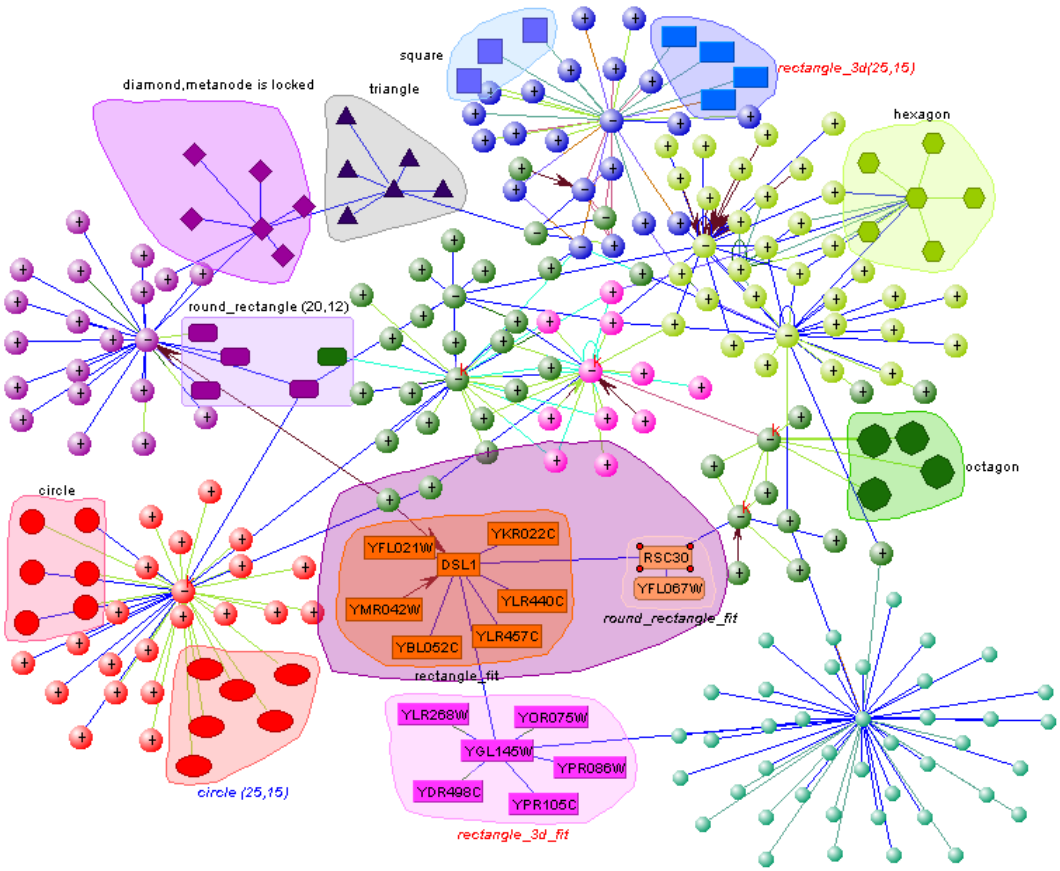


Figure 3.6: Visualization of a sample network in VisANT with nested node groups

## 3.5 WiGis

WiGis [23] is a web-based framework for interactive graph visualization. The lightweight and flexible architecture of WiGis enables it to run on a basic browser without requiring additional plugins. WiGis is specifically desgined to visualize large networks with less scalability limitations.

WiGis enables interaction with the graph for the actions such as node repositioning, zooming, and filtering. WiGis provides various options for complexity management, however, it does not allow to change the graph topology interactively by creating new nodes and edges or by removing existing nodes and edges.

Unfortunately, WiGis does not support compound graphs, and also it does not offer easy customization methods to build domain-specific applications. Although, WiGis provides an interface for plugging in custom layout algorithms, it is not easy to extend WiGis for custom interactive functionality and for custom node and edge views.

# Chapter 4

# ChiWeb Architecture

ChiWeb is designed as an ActionScript [1] component and it is built on Flare library (Section 2.3). Taking advantage of Flare's data management and visualization support, ChiWeb provides a framework which enables easy customization of graph components and interaction handling mechanism. The rest of this chapter gives detailed information on the main components of ChiWeb architecture.

## 4.1 ChiWeb Model

ChiWeb's model structure (Figure 4.2) is based on Flare's data model. Main components of the ChiWeb model are: a graph, nodes, edges, and visual styles.

The `Graph` class represents the graph model with its `Data`. The graph data contains data groups for nodes, edges, compound nodes, bend points, regular edges, selected edges, and selected nodes. These groups help to manage the graph data for specific conditions. For example, when it is desired to perform an operation on all selected nodes, the corresponding data group allows a fast access to the selected nodes.

The `Node` class, which extends Flare's `NodeSprite`, represents simple (regular) nodes, compound nodes and bend nodes (bend points). A simple (regular) node

is the basic node with no nested structure. A compound node has its child nodes, bounds, and padding values. A compound node can contain another node (either a simple node or a compound node) as its child. A bend node, on the other hand, is a special kind of node that is used to create bend points for edges. Padding values represent the distance between the minimum rectangular bounds enclosing the children of a compound and the border of that compound. Bounds of a node represent bounds enclosing the children of a compound node plus compound's padding values.

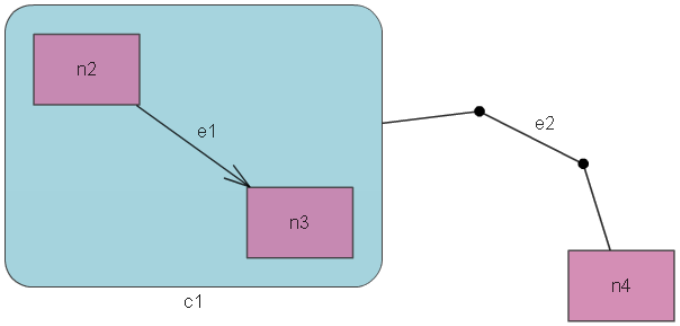


Figure 4.1: An example graph with 3 simple nodes (n2, n3, n4), one compound node (c1), one regular edge (e1) with a simple target arrow, and one edge (e2) with 2 bend points and 3 segments

The `Edge` class, which extends Flare's `EdgeSprite`, represents both regular edges and segment edges. A regular edge is an edge between two nodes each of which is either a regular node or a compound node. A regular edge can contain bend nodes (bend points) and edge segments. On the other hand, a segment edge, which is a child of a regular edge, is an edge between two bend nodes or between one bend node and one actual node where this actual node is either the source or the target node of the parent regular edge.

Both `Node` and `Edge` classes implement the interface `IStyleAttachable`. This interface is designed to help management of visual styles for nodes and edges. Both a node and an edge have a `StyleManager` to keep track of the styles attached to them. Each node and each edge has a default style, and can have an arbitrary number of group styles. It is also possible to attach a style specific to a certain node or edge.

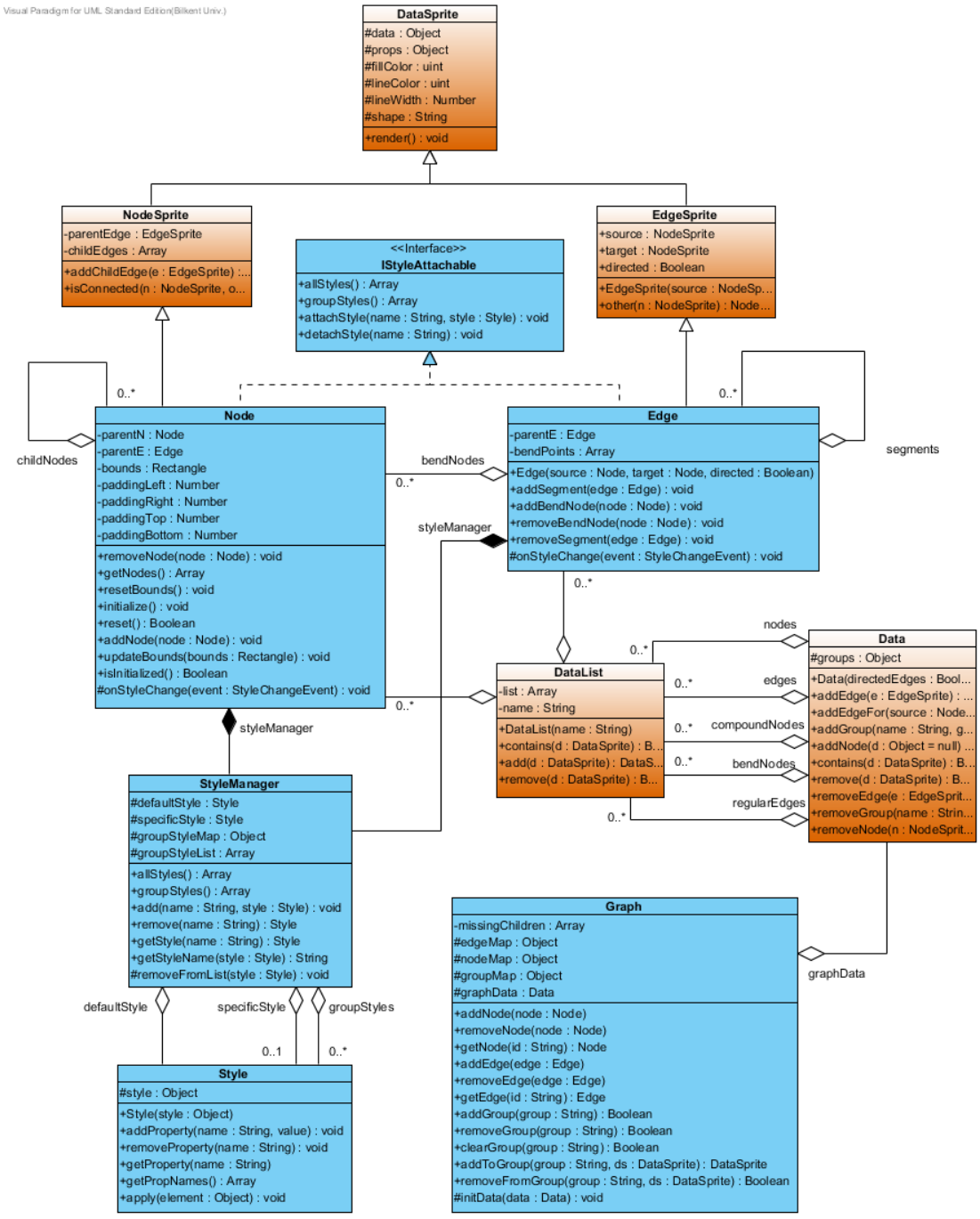


Figure 4.2: Graph model and data representation in ChiWeb

The `Style` class represents a style with visual property and value pairs. It is possible to define an arbitrary number of visual properties within a style. A style is applied on an object with the help of the apply method. More details about visual styles and style management mechanism are given in Section 4.4. Section 5.1, on the other hand, demonstrates how to define custom styles for nodes and edges by providing sample implementations.

In a graph, a unique id is assigned to a node or an edge while adding it to the graph data. Since both `Node` and `Edge` classes inherit from `DataSprite`, this id value is stored as a data attribute within the `data` field of the sprite. In addition to the graph data, a graph also contains two maps, which are keyed on the id attribute, one for nodes and the other for edges. These maps allow fast access to nodes and edges by using their ids.

## 4.2 Renderers and UIs

ChiWeb defines its own renderers for simple nodes, compound nodes, and edges (Figure 4.3). `NodeRenderer`, which extends Flare's `ShapeRender`, is to render simple nodes, and `CompoundNodeRender`, which extends `NodeRenderer`, is to render compound nodes. `EdgeRenderer`, on the other hand, extends Flare's `EdgeRenderer` and it is to render edges. All renderers in ChiWeb are singleton classes. Therefore, a single instance of `NodeRenderer` renders all nodes, and so on.

In contrast to Flare's `ShapeRenderer`, `NodeRenderer` does not rely on predefined shapes. In fact, it does not perform the actual drawing of a simple node. Instead, the graphical component of a node is drawn by an `INodeUI` instance specific to that node's shape property. ChiWeb provides three default node UIs implementing the `INodeUI` interface (Figure 4.3). Any `INodeUI` instance can be registered to the system with a unique UI name by the help of the `registerUI` function of `NodeUIManager`. Then, `NodeRenderer` accesses to a registered `INodeUI` instance by the `getUI` function, and it calls the `draw` method

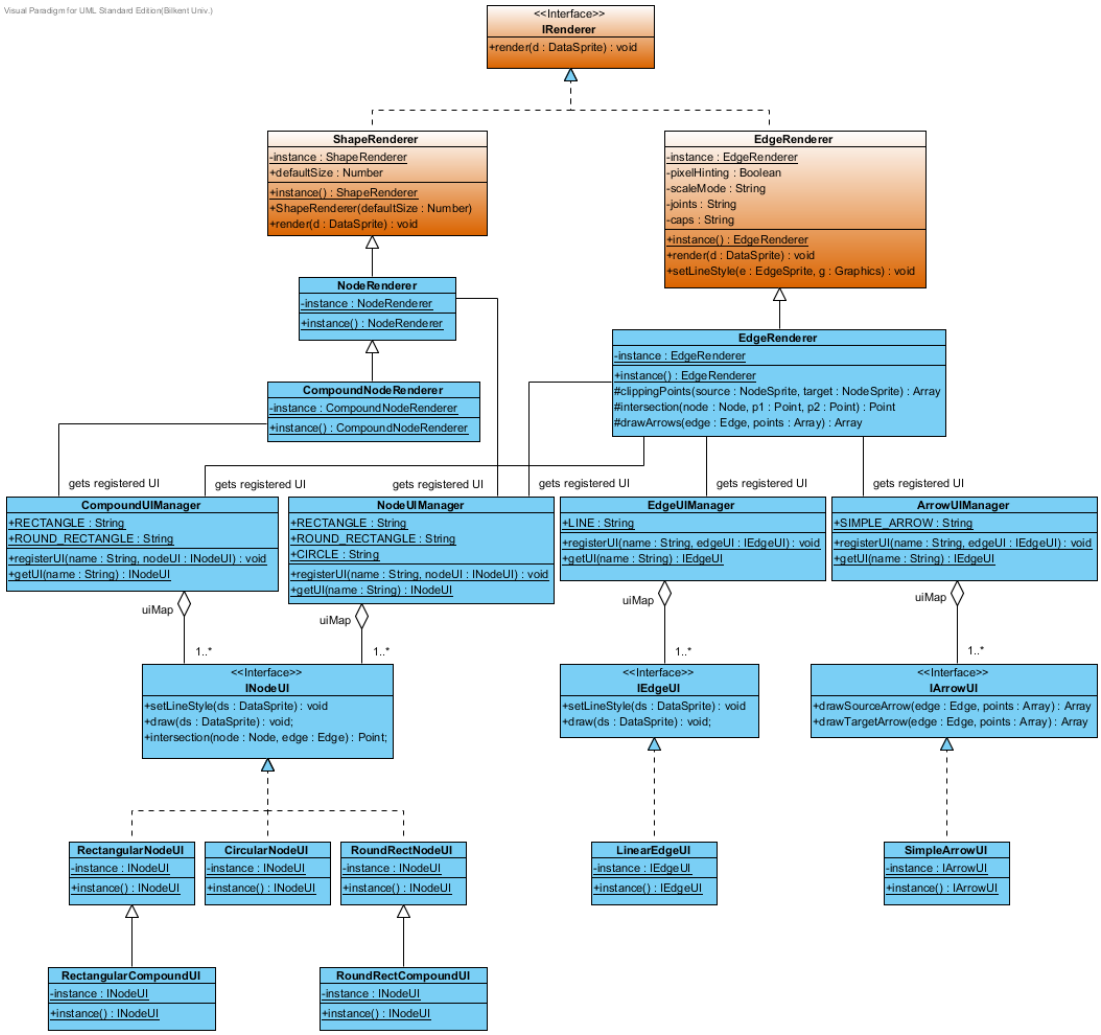


Figure 4.3: Renderer and User Interface (UI) classes for data sprites

of the node UI to render the node. Defining additional classes implementing the `INodeUI` interface enables customization of node UIs. In order to associate a node with a certain node UI, it is enough to set the `shape` property of the node to the registered name of the node UI.

Compound node rendering is similar to simple node rendering. In order to render a compound node, `CompoundNodeRenderer` uses `CompoundUIManager` to access to a registered `INodeUI` instance, and calls its `draw` method. ChiWeb provides two default compound node UIs (Figure 4.3). The main difference between a simple node and a compound node is node bounds. A fixed size or bounds can be defined for a simple node. However, the bounds of a compound node depend on the size and the position of its children. An `INodeUI` instance should always take the bounds enclosing the compound node's children into account when drawing the compound node. For the example graph in Figure 4.1, bounds of the compound node (*c1*) depend on the position and size of its children (*n2* and *n3*) and also depend on the padding values. For this specific example, left padding value is the distance between the left border of the node *n2* and the left border of the compound *c1*.

Similar to `INodeUI`, ChiWeb provides the interface `IEdgeUI` for edge UI customization. In addition to `IEdgeUI`, another interface called `IArrowUI` is provided to enable customization of edge arrows for directed graphs. There is one `IEdgeUI` and one `IArrowUI` implementation by default (Figure 4.3). In order to associate an edge with a certain edge UI, it is required to set the `shape` property of the edge to the registered name of the edge UI. Similarly, to associate an edge with an arrow UI it is enough to set the property `sourceArrowType` or `targetArrowType` to the registered name of the arrow UI. Figure 4.1 illustrates a default edge (*e1*) with a default target arrow.

ChiWeb's default edge UI (`LinearEdgeUI`) draws an edge as a line from its source node to its target node. However, by defining a custom edge UI and implementing the `draw` method accordingly, it is possible to render an edge in any desired form. It is possible to achieve an arbitrary drawing, such as a custom shape in the middle of an edge, with the custom implementation of the `draw`

method.

Flare's default `EdgeRenderer` draws an edge from the center of the source node to the center of the target node. However, for a more aesthetically pleasing view, edge rendering should start from a point on the bounds of the source node and should end on a point on the bounds of a target node. Those points are called *clipping points*. In order to provide clipping point support for edge rendering, it is required to calculate clipping points for both the source and the target. Since a node can have an arbitrary shape, a function calculating the clipping point is defined as a part of the `INodeUI` interface instead of the `IEdgeUI` interface. `EdgeRenderer` gets the corresponding node UIs and calls the method `intersection` for both the source and the target nodes to find two clipping points. After calculating clipping points, an edge is rendered by calling the `draw` method of the corresponding edge UI.

Default node and edge UIs render sprites by using their graphical properties encoded directly as field values of sprites, which are inherited from `DataSprite`, such as `fillColor`, `fillAlpha`, `lineColor`, `lineWidth`, and `size`. Apart from these default properties, an edge UI or a node UI may also use custom properties attached as fields to the `props` object of the associated sprite.

## 4.3 Visualization and Management of the Graph

`GraphManager` (Figure 4.4) is designed to manage both the topology and the view of the graph. It contains sophisticated methods to add a node, to add an edge, to add a bend point for an edge, to remove an existing element, etc. These methods enable modification of the graph topology and use the basic functionality provided by `Graph` to perform advanced tasks on the graph. After modifying the topology, topology related methods also update the view to keep consistency between the view and the topology of the graph.

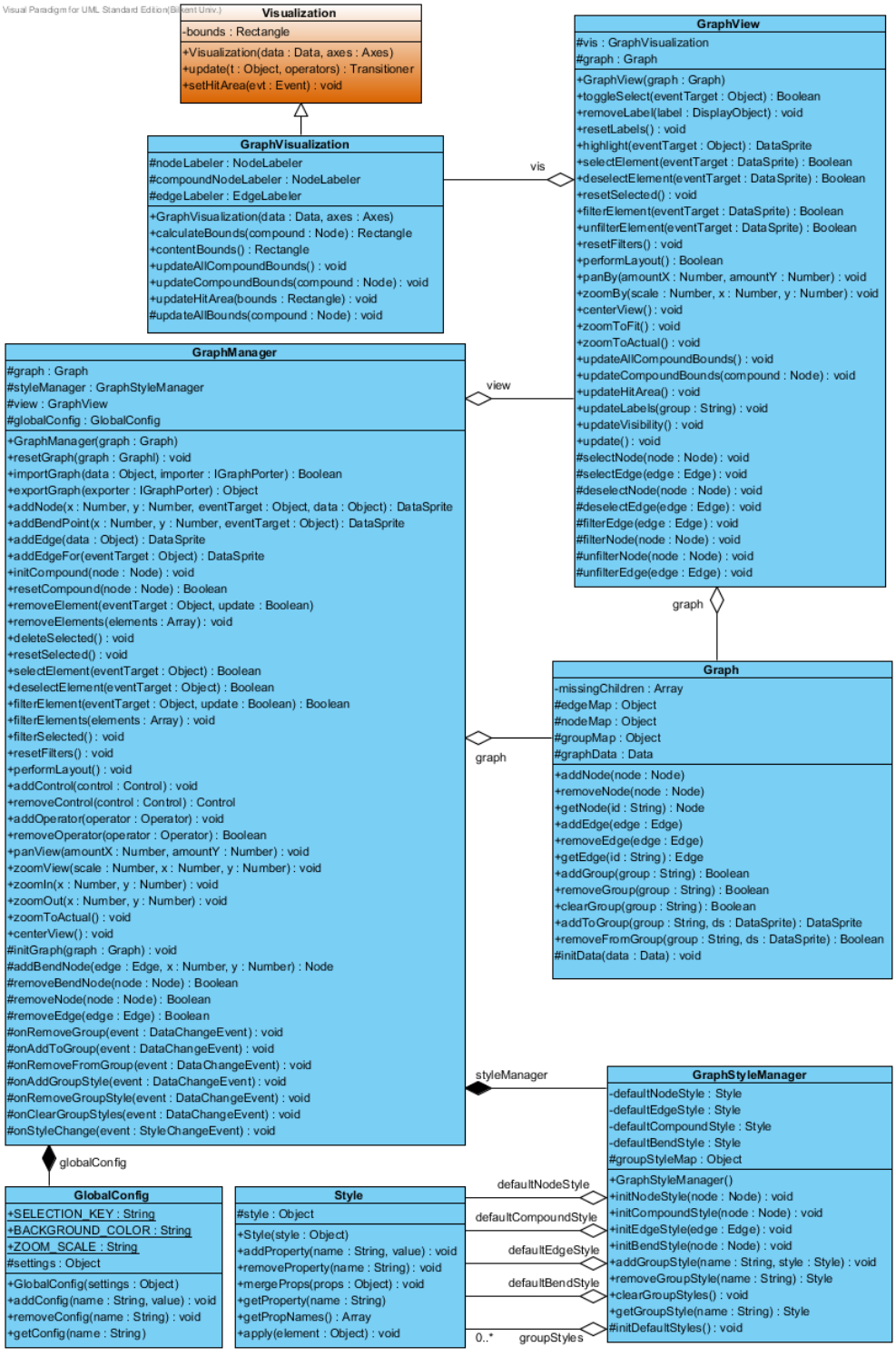


Figure 4.4: Graph management and visualization

`GraphManager` contains methods to select and deselect a graph element, to hide and show an element, to pan and zoom the graph view, etc. These methods enable modification of the graph view and use the functionality provided by `GraphView` (Figure 4.4) to perform required tasks. Functionality provided by `GraphManager` is mainly used by `EventControl` classes (Section 4.5) within ChiWeb.

`GraphManager` also contains listener methods to reflect the changes in the visual styles to the graph elements. Visual style management is described in Section 4.4 in detail.

`GraphView` is the main class providing functionality to highlight a graph element when it is selected, to update visibility of elements for filtering purposes, to update labels, to pan or zoom, to center the view, etc. `GraphView` also handles the visualization of the graph by using the functionality provided by `GraphVisualization` (Figure 4.4).

`GraphVisualization` extends Flare's `Visualization` by adding extra functionality for the hit area of the canvas, compound bounds, bounds of the visible elements, and the labels. Functionality provided by `GraphVisualization` is crucial to keep the graph elements consistent with each other. It is critical to update the bounds of a compound node, after a drag event for instance, to keep it consistent with its children. It is also critical to update edge and node labels, after a layout operation for instance, to keep them consistent with their owners.

## 4.4 Visual Style Management

As described in Section 4.2, renderers and default UIs render data sprites (nodes or edges) by using both the default graphical properties and the additional attributes attached to the `props` object. In order to change the appearance of the sprite, it is required to modify the values of these properties.

As introduced in Section 4.1, it is possible to attach one or more *visual styles* to

a node or an edge. When these visual styles are applied on the sprite, graphical properties of the sprite are updated with the values obtained from the visual styles.

`GraphStyleManager` is to define visual styles for the data groups. `GraphStyleManager` defines 4 default styles (Figure 4.4) for the default data groups. It is possible to add custom styles for both the default and the custom data groups by using the `addGroupStyle` method. `GraphManager` attaches the corresponding default style to the data sprite after creation of a node, a compound node, an edge or a bend point. This ensures that a data sprite has at least one style, which is the default style.

Adding a style for a specific data group allows defining a style for each member in that group. In other words, a group style is shared among all sprites in that data group, and a change in a group style is reflected to all sprites in that specific data group. It is also possible to define a specific style per node or per edge. This allows attaching a unique style for a certain data sprite.

Attaching a visual style to a data sprite does not change its graphical properties immediately. In order to reflect the style to the sprite, it should be applied on the sprite by invoking the `apply` method. ChiWeb provides a utility class `Styles` (Figure 4.18) to perform necessary updates when it is required to re-apply styles to the sprites. The function `reApplyStyles` always applies the default style first and element-specific style last. Group styles are applied after the default style, and before the element-specific style. This order gives priority to the element-specific style over group styles, and priority to group styles over the default style. This mechanism prevents general styles to overwrite more specific styles.

In order to preserve synchronization between the visualization of the sprite and the styles attached to it, ChiWeb takes advantage of the event dispatching facility of Flash. `DataChangeEvent` and `StyleChangeEvent` (Figure 4.17) are specifically designed to help visual style synchronization.

When a new sprite added to a data group, a `DataChangeEvent` specific to this action is dispatched by `Graph`, and the associated listener in `GraphManager`

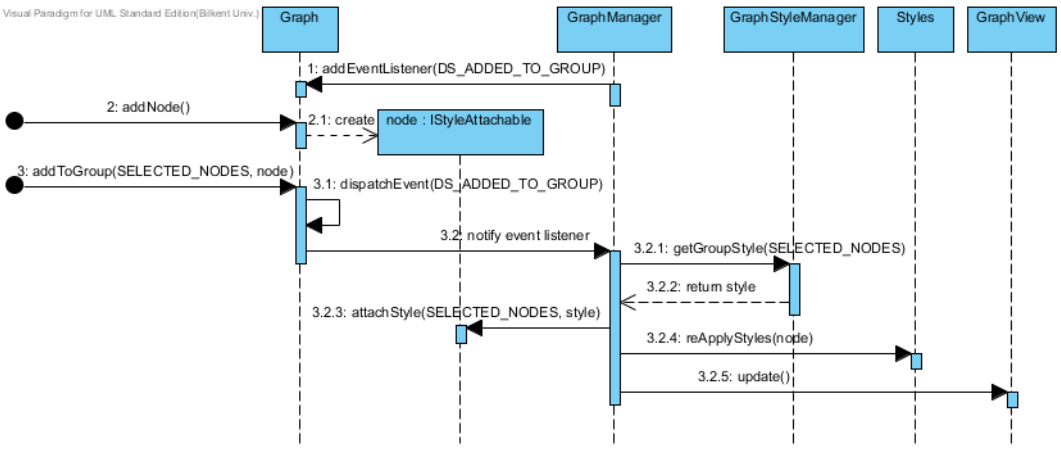


Figure 4.5: Adding a node into a data group of selected nodes

performs the tasks required to update the visualization. Figure 4.5 illustrates the sequence of operations related to data groups for selection of a node. This sequence diagram does not show the sources calling the methods `addNode` and `addToGroup` for the sake of simplicity. Details of node creation via user interaction are given in Figure 4.10.

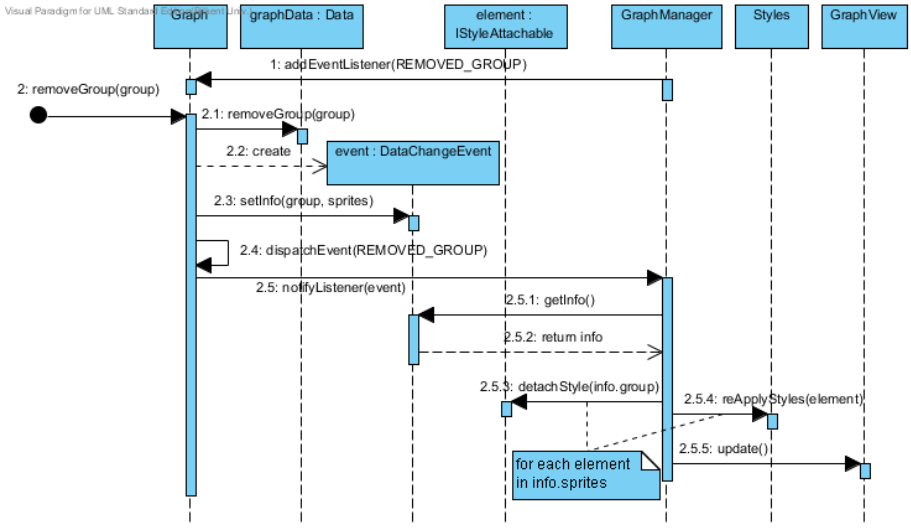


Figure 4.6: Removing an existing group from graph data

Similarly, specific events are dispatched by `Graph` when a sprite is removed from a data group, and when an existing group is removed from the graph data (Figure 4.6).

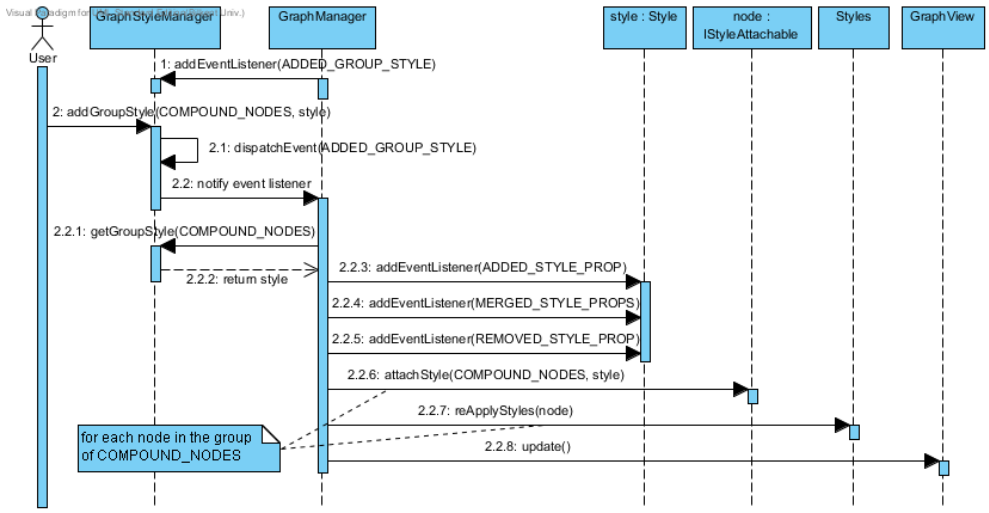


Figure 4.7: Adding a style specific to the data group of compound nodes

`GraphStyleManager` dispatches a `DataChangeEvent` when a new style is added for a data group (Figure 4.7), and dispatches another `DataChangeEvent` when a group specific style is removed.

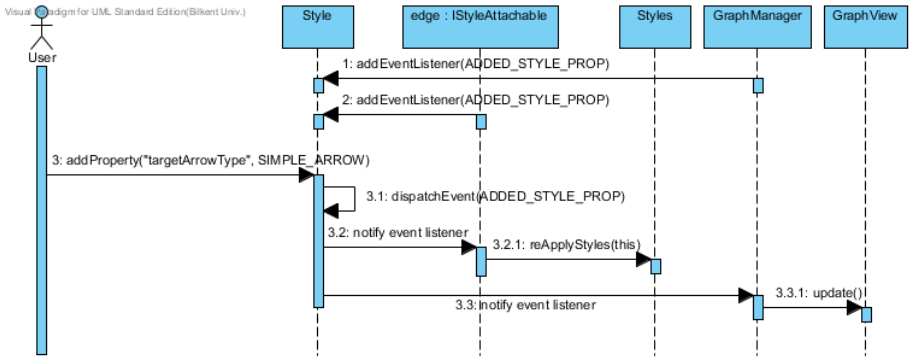


Figure 4.8: Changing a visual style specified for edges by adding a new property

On the other hand, each data sprite (a node or an edge) listens to all styles attached to it for a property change. When a style property changes, or a new property is added to a style, a `StyleChangeEvent` is dispatched by that style. Then, the associated listener of the sprite performs required updates (Figure 4.8).

## 4.5 Controls

ChiWeb provides a control mechanism for various interactive actions such as node and edge selection, node dragging, panning and zooming, clicking on the canvas or on a graph element, and pressing a keyboard key.

`EventControl`, which extends Flare's `Control`, is designed as a base class for the other control classes in ChiWeb. There are 7 default controls provided by ChiWeb (Figure 4.5). Each control class listens to specific events and performs required tasks on the event target or on the whole graph via the methods provided by `GraphManager`.

Apart from the default controls, it is possible to define custom listener functions for specific events and activate them via the `addCustomListener` method of `ControlCenter`. For an advanced control mechanism, it is also allowed to add or remove controls using the `addControl` and the `removeControl` methods of `ControlCenter`. A predefined default control cannot be removed from `ControlCenter`. However, it is possible to disable a default control with the method `disableDefaultControl` for an advanced customization. A default control can be enabled again at any time by invoking the method `enableDefaultControl`.

Every control class has access to a single `StateManager` instance to check or set flags for specific states. `StateManager` is specifically designed for event control mechanism. A state manager can be considered as a shared communication platform among all control classes. A state manager contains 11 predefined flags by default (Figure 4.5). Apart from these default flags, it is also possible to attach custom state flags to a state manager. These custom flags, then, can be used by the custom controls.

Some of the default control classes dispatch a `ControlEvent` (Figure 4.17) when a specific task starts or ends. For example, `PanControl` dispatches events when panning starts and ends. A `ControlEvent` is dispatched on the interactive object to which the control is attached, and this interactive object

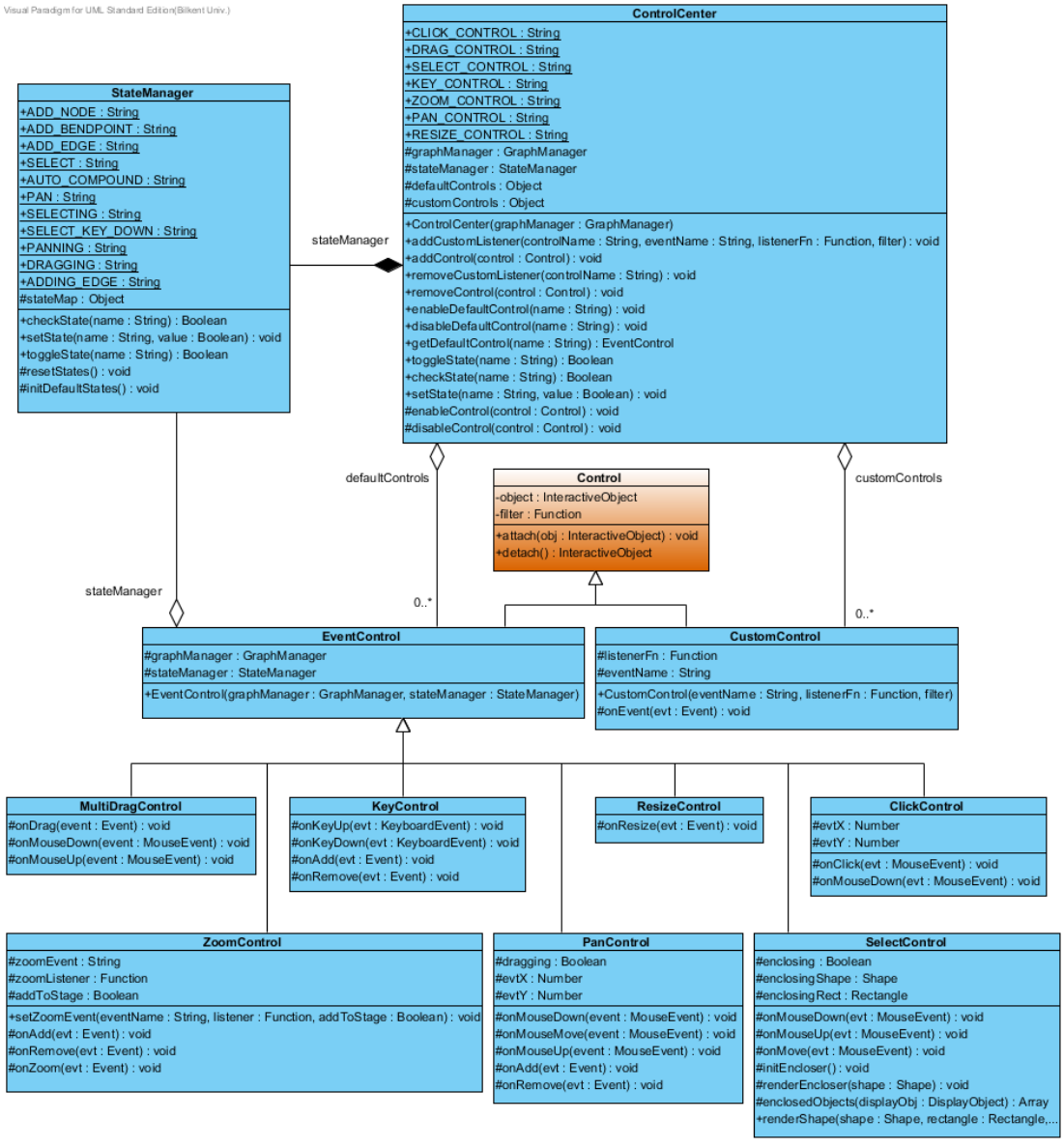


Figure 4.9: Event control classes

is the `GraphVisualization` instance for the default controls. Dispatching a `ControlEvent` lets a function, which listens to this `GraphVisualization` instance, perform additional tasks after a specific event.

Examples of control customization are given in Section 5.2 with sample implementations. The rest of this section explains each default control class in detail.

### 4.5.1 ClickControl

`ClickControl` is designed to handle single click events on the graph canvas or on node and edge sprites. Upon a click event, `ClickControl` performs a task according to active states of `StateManager`.

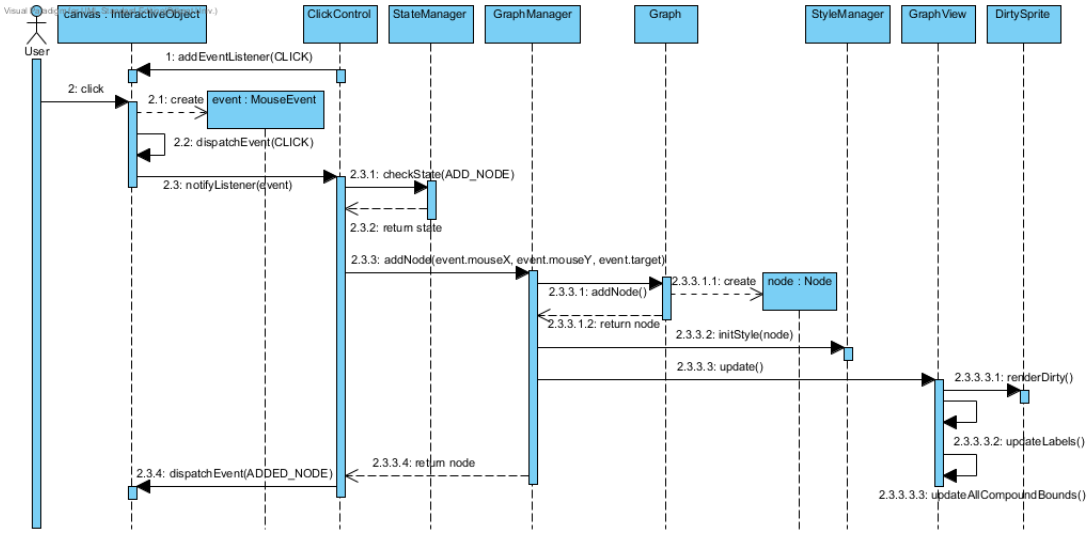


Figure 4.10: Adding a new node into the graph canvas

When the `SELECT` state is active, a single click on a sprite (a node or an edge) results in selection of that sprite. If the `SELECT_KEY_DOWN` state is activated by holding down the selection key (which is the CTRL key by default), then previously selected sprites remain selected. On the other hand, if the selection key is inactive, a click on a sprite or on the canvas resets all previous selections.

When the `ADD_NODE` state is active, a click on the canvas adds a new node to

the graph (Figure 4.10), and a click on another node adds a new node as a child into the target node. The node is added to the location of the click event by invoking the `addNode` method of `GraphManager`. `GraphManager` adds the node to the graph, initializes its default style, and updates the graph view. `ClickControl` dispatches a specific `ControlEvent` (`ADDED_NODE`) after node creation.

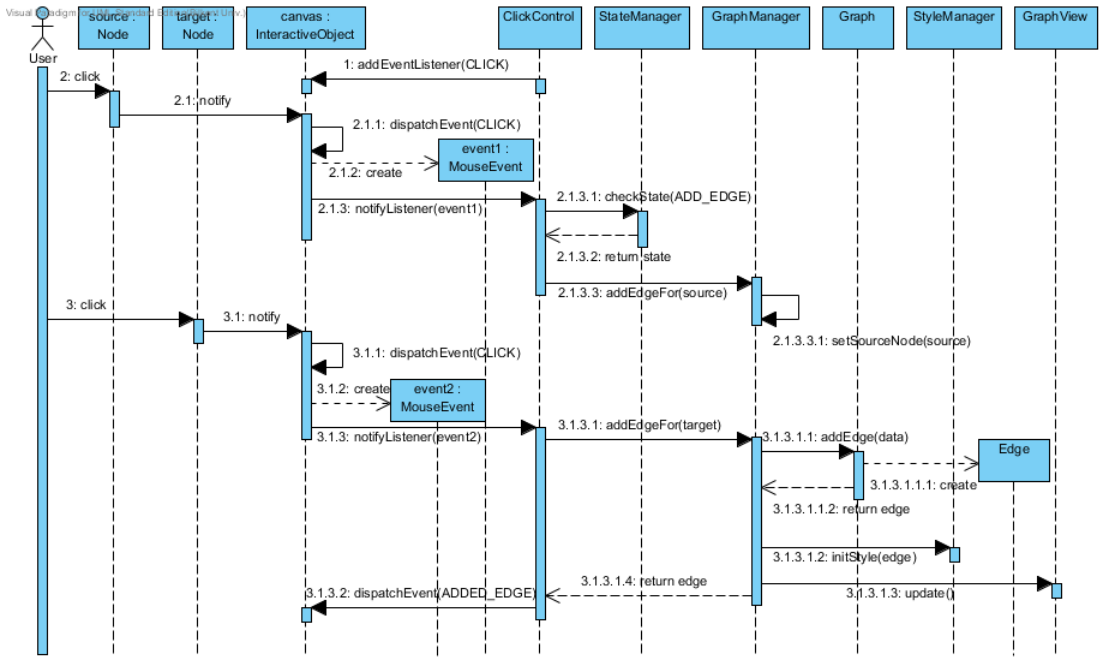


Figure 4.11: Adding an edge between two existing nodes

When the `ADD_EDGE` state is active, successive clicks on two different nodes add a new edge between those nodes (Figure 4.11). First clicked node becomes the source of the added edge, and second clicked node becomes the target of the edge. `ClickControl` dispatches a specific `ControlEvent` (`ADDED_EDGE`) after adding the edge. It also dispatches another `ControlEvent` (`ADDING_EDGE`) after the first click of the edge adding process.

When the `ADD_BENDPOINT` state is active, a click on an edge creates a new bend point on the edge (Figure 4.12). The bend point is created at the location of the click event. A specific `ControlEvent` (`ADDED_BEND`) is dispatched after a bend point is added.

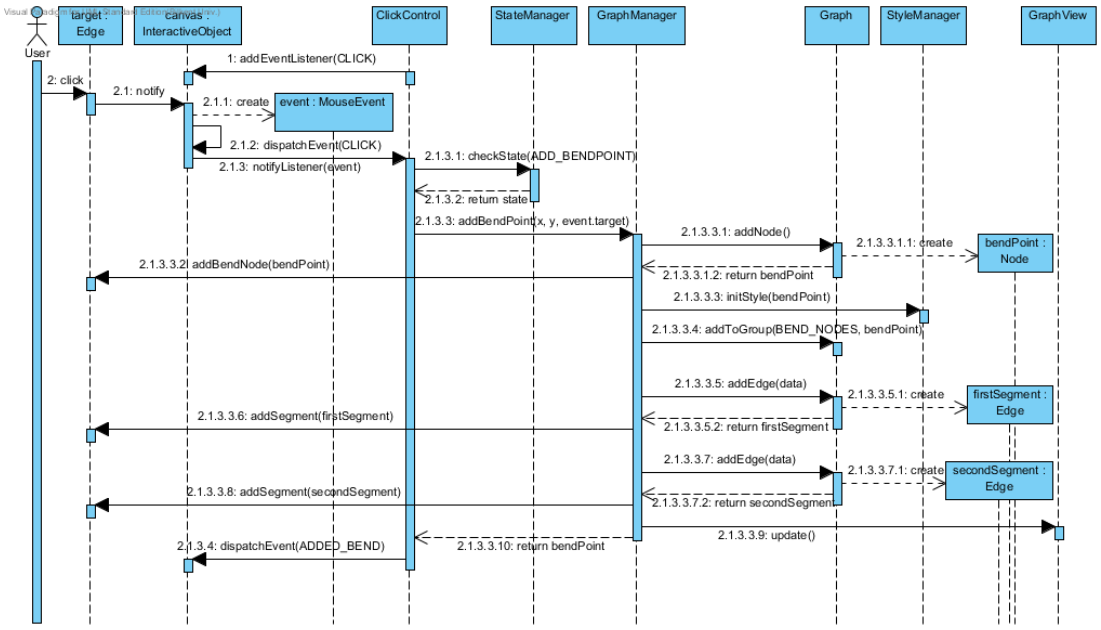


Figure 4.12: Adding a bend point for a regular edge

### 4.5.2 SelectControl

While selection by single click is controlled by `ClickControl`, selection of multiple graph elements is controlled by `SelectControl`. It allows selection of multiple elements by dragging mouse while mouse key is down (Figure 4.13).

When the `SELECT` state is active, holding the mouse down on the graph canvas activates the selection rectangle. The selection rectangle, which is temporarily added to the canvas during selection, is a `Shape` instance, and its bounds are updated upon each mouse movement as long as the mouse button is kept down. Graph elements enclosed by the selection rectangle are selected as soon as the mouse button is released.

Similar to selection by single click, if the `SELECT_KEY_DOWN` state is active, then previously selected sprites remain selected. If it is inactive, `SelectControl` resets previous selections.

`SelectControl` dispatches two control events specific to selection. One is the `SELECT_START` event which is dispatched when selection starts with a `MOUSE_DOWN`

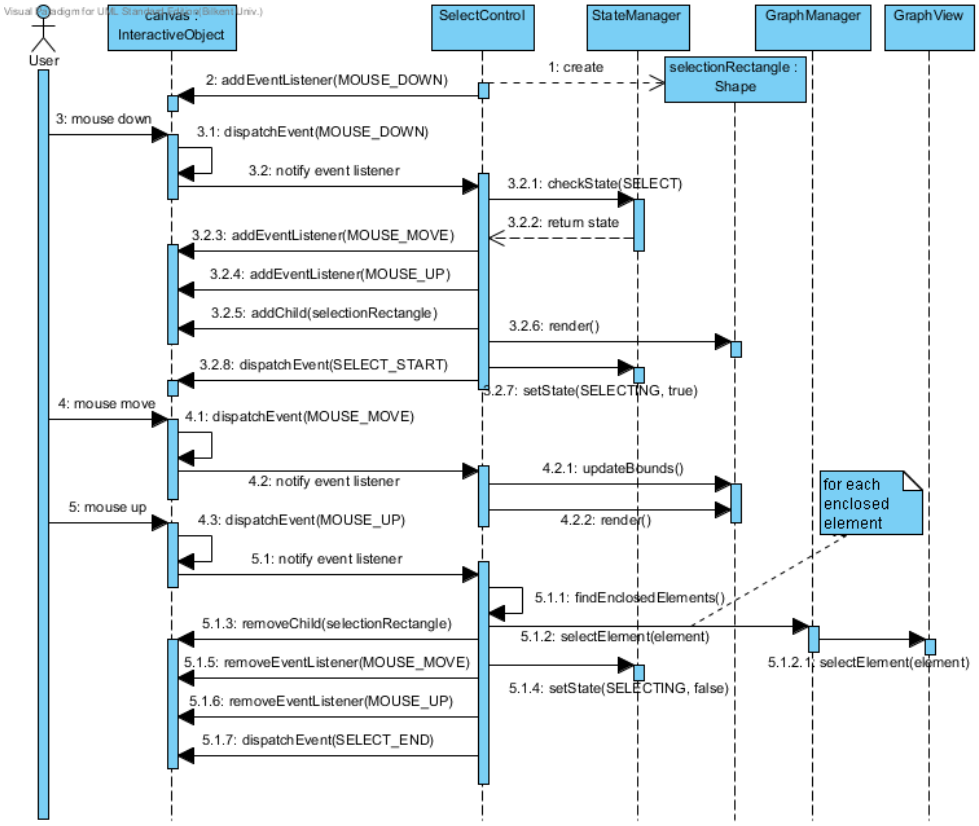


Figure 4.13: Selecting multiple graph elements

event, and the other is the SELECT_END event which is dispatched when selection ends with a MOUSE_UP event.

Multiple element selection is initiated only if the target of the MOUSE_DOWN event is the graph canvas. Holding the mouse down on a sprite (a node or an edge) is considered as a drag event and handled by MultiDragControl (Section 4.5.3).

### 4.5.3 MultiDragControl

MultiDragControl is to drag multiple nodes with a single drag event. MultiDragControl captures a MOUSE_DOWN event where the event target is a node sprite, and activates the DRAGGING state of StateManager (Figure 4.14). If a MOUSE_MOVE event is dispatched while the DRAGGING state is active, target node is dragged to the new location of the mouse.

If the target node is a compound, its children are also dragged together with the compound. If the target node is selected and there are also other selected nodes, MultiDragControl drags all the selected nodes as well as the target node.

In the case when a node inside a compound node is dragged, but the compound itself is not dragged, it is required to update the bounds of the compound node to keep its bounds consistent with respect to its children. If the compound node is a child of another compound node, its parent should also be updated. Therefore, MultiDragControl updates bounds by calling the updateCompoundBounds method of GraphView for all the parent compound nodes up to the root parent.

Dragging ends with a MOUSE_UP event. MultiDragControl performs a final update on the graph view, and deactivates the DRAGGING state.

MultiDragControl dispatches two control events specific to dragging operations. One is the DRAG_START event which is dispatched when dragging starts with a MOUSE_DOWN event, and the other is the DRAG_END event which is dispatched when dragging ends with a MOUSE_UP event.

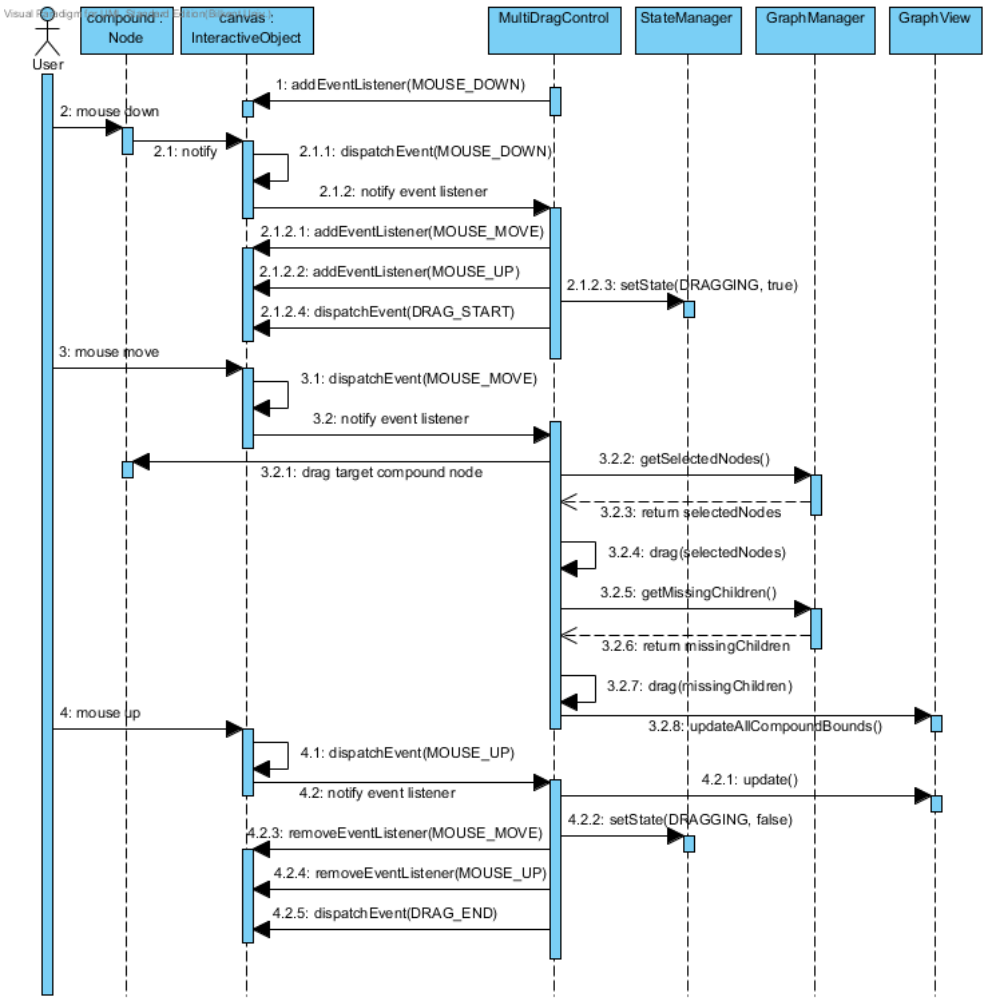


Figure 4.14: Dragging a compound node together with other selected nodes

### 4.5.4 KeyControl

`KeyControl` is specifically designed to control `KEY_DOWN` and `KEY_UP` events for the selection key (the key used for multiple element selection). When a `KEY_DOWN` event is dispatched, `KeyControl` activates the `SELECT_KEY_DOWN` state of `StateManager` if the select key is down. Similarly, when a `KEY_UP` event is dispatched, it deactivates the `SELECT_KEY_DOWN` state.

The selection key is the CTRL key by default, but it is possible to specify a custom key via global configuration (Figure 4.4).

### 4.5.5 ResizeControl

`ResizeControl` is designed to control the `RESIZE` event dispatched by the graph canvas. Whenever the graph canvas is resized, `ResizeControl` updates the hit area of the graph canvas to ensure that visible area of the canvas is always interactive.

### 4.5.6 PanControl

`PanControl` is designed to enable panning of the graph canvas by clicking on and dragging it. Like `MultiDragControl`, it listens to `MOUSE_DOWN` events. If the event target is the graph canvas, `PanControl` activates the `PANNING` state of `StateManager`. If a `MOUSE_MOVE` event is dispatched while the `PANNING` state is active, graph canvas is panned by the amount of the mouse movement.

Panning ends with a `MOUSE_UP` event. When panning ends, `PanControl` updates the hit area of the graph canvas, and deactivates the `PANNING` state.

In order to enable panning it is required to activate the `PAN` state of `StateManager`. This is the main difference of `PanControl` from `MultiDragControl`, since dragging is enabled in any case by default, unless `MultiDragControl` is explicitly disabled using the `disableDefaultControl`

method of `ControlCenter`. `PanControl`, on the other hand, can be deactivated by toggling the `PAN` state of `StateManager`.

Similar to `MultiDragControl`, `PanControl` dispatches a `PAN_START` event when panning starts with a `MOUSE_DOWN` event, and dispatches a `PAN_END` event when panning ends with a `MOUSE_UP` event.

### 4.5.7 ZoomControl

`ZoomControl` is designed to enable zoom operations on the graph view. By default, `ZoomControl` listens to `MOUSE_WHEEL` events, and zooms in or zooms out according to the direction of the wheel movement. `ZoomControl` is always active by default, but it can be disabled using the `disableDefaultControl` method of `ControlCenter`.

It is possible to customize the event type and the event listener function by changing corresponding attributes of this control. When it is customized, `ZoomControl` listens to a custom event instead of a `MOUSE_WHEEL` event, and invokes the provided listener function instead of its default listener.

## 4.6 Operators

In addition to Flare's default operators, ChiWeb provides operators for node and edge labels (Figure 4.15). `NodeLabeler` extends Flare's `Labeler` to provide additional functionality for simple and compound node labels. `EdgeLabeler`, which extends `NodeLabeler`, helps to position and display edge labels in a desired way.

ChiWeb also provides another operator named `LayoutOperator`, which is designed as a base class for layouts to be implemented by the applications using ChiWeb as a library. `LayoutOperator` extends Flare's `Layout` class (Figure 4.15).

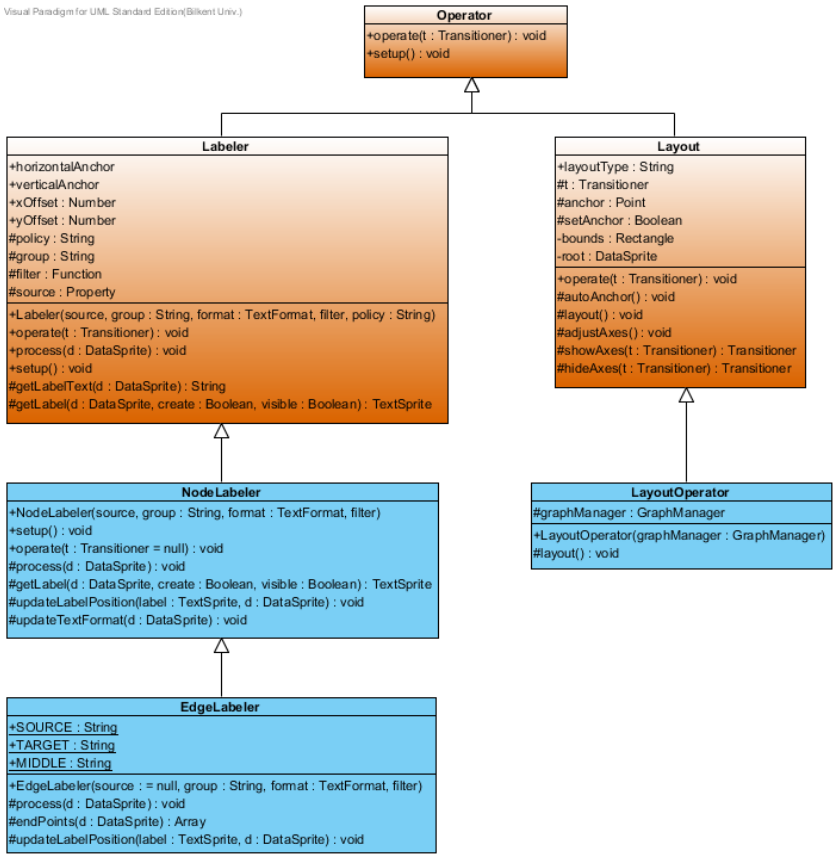


Figure 4.15: Operators introduced by ChiWeb

## 4.7 Persistency

ChiWeb defines an interface `IGraphPorter` (Figure 4.16) to provide a basic template to import/export graph information from/into various formats. By default, ChiWeb includes an implementation of this interface for GraphML format [21] to import and export graphs as GraphML files.

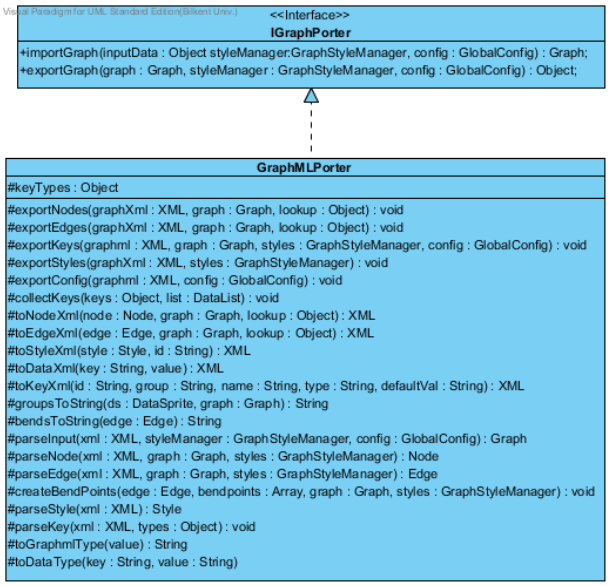


Figure 4.16: Persistency support of ChiWeb

Although it is possible to save and load the graph in standard GraphML format, ChiWeb extends this format to store visual style and global configuration information in a proper way. ChiWeb introduces *style* and *config* elements in addition to GraphML's default elements. While a *config* element can only be a child of the *graphml*, a *style* element can be added as a child to a *graph*, to a *node*, and to an *edge*.

GraphML contents given below partially illustrate the representation of the graph in Figure 5.1 in the extended GraphML format defined by ChiWeb. In addition to default GraphML elements, this example contains a config element to store global configuration parameters, and style elements to store group style properties.

```
<graphml>
```

```
<key id="x" for="node" attr.name="x" attr.type="double"/>
<key id="y" for="node" attr.name="y" attr.type="double"/>
<key id="bendpoints" for="edge" attr.name="bendpoints" attr.type="string"/>
<key id="groups" for="all" attr.name="groups" attr.type="string"/>
<key id="shape" for="all" attr.name="shape" attr.type="string"/>
...
<config id="globalConfig">
  <data key="enclosingLineWidth">1</data>
  <data key="zoomScale">0.8</data>
  <data key="marqueeFillColor">16299396</data>
  <data key="enclosingFillAlpha">0.2</data>
  <data key="enclosingLineColor">8947967</data>
  <data key="backgroundColor">4294572537</data>
  <data key="marqueeLineColor">16435878</data>
  <data key="enclosingFillColor">8947967</data>
  <data key="marqueeLineWidth">2</data>
  <data key="selectionKey">ctrlKey</data>
  <data key="panAmount">20</data>
  <data key="enclosingLineAlpha">0.4</data>
</config>
<graph>
  <node id="n1">
    <data key="x">-201</data>
    <data key="y">-1.85</data>
    <data key="label">n1</data>
    <data key="groups">compoundNodes</data>
    <graph>
      <node id="n3">
        <data key="x">-256.35</data>
        <data key="y">-29.3</data>
        <data key="label">n3</data>
        <data key="groups">circularNode</data>
      </node>
      <node id="n4">
        <data key="x">-145.7</data>
        <data key="y">25.5</data>
        <data key="label">n4</data>
        <data key="groups">circularNode</data>
      </node>
      <edge id="e1" source="n3" target="n4">
        <data key="label">e1</data>
        <data key="groups">regularEdges</data>
      </edge>
    </graph>
  </node>
  ...
  <edge id="e2" source="n2" target="n5">
    <data key="label">e2</data>
    <data key="groups">dashedEdge;regularEdges</data>
  </edge>
```

```
    ...
    <style id="gradientRect">
      <data key="shape">gradientRect</data>
      <data key="alpha">0.7</data>
      <data key="gradientAngle">autoAngle</data>
      <data key="gradientType">linear</data>
      <data key="w">80</data>
      <data key="fillColor">4282279664</data>
      <data key="spreadMethod">reflect</data>
      <data key="lineWidth">2</data>
      <data key="interpolationMethod">rgb</data>
      <data key="h">60</data>
      <data key="labelFontWeight">bold</data>
      <data key="labelFontStyle">italic</data>
    </style>
    <style id="dashedEdge">
      <data key="shape">dashedEdge</data>
      <data key="alpha">0.8</data>
      <data key="onLengthCoeff">1</data>
      <data key="offLengthCoeff">2</data>
      <data key="lineColor">4284926994</data>
      <data key="targetArrowType">simpleArrow</data>
      <data key="lineAlpha">0.5</data>
      <data key="lineWidth">3</data>
    </style>
    ...
  </graph>
</graphml>
```

## 4.8 Events

ChiWeb provides event classes (Figure 4.17) to enable further customization of the application behavior. `ChiWebEvent`, which extends `Event` class of Flash, is a base class for all other events in ChiWeb. It introduces a new `info` field to attach additional information to an event object before dispatching it.

`DataChangeEvent` and `StyleChangeEvent` are designed to keep consistency of visual styles (Section 4.4) whereas `ControlEvent` is designed to be dispatched within the `EventControl` classes (Section 4.5) and allows performing additional tasks after a specific event.

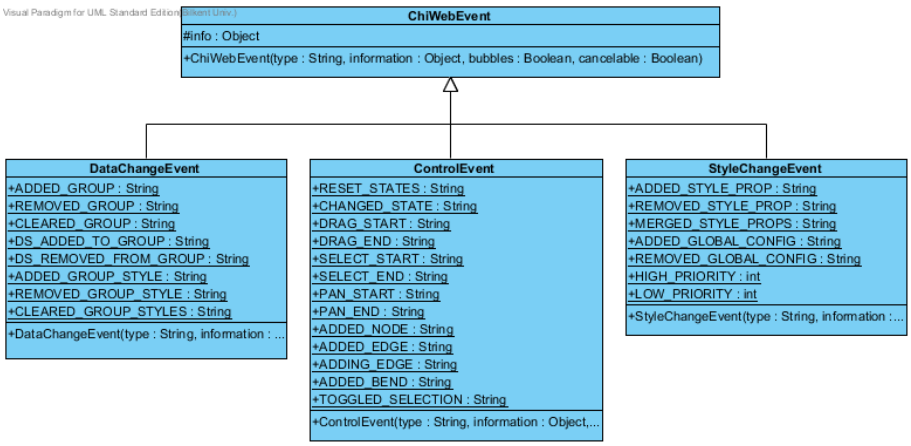


Figure 4.17: Event classes introduced by ChiWeb

## 4.9 Utility Classes

ChiWebprovides utility classes (Figure 4.18) for some specific tasks. Utility functions provided by these classes are used within various components of ChiWeb. `Edges`, `Nodes` and `Styles` contain utilities related to the ChiWeb model (Section 4.1) and mostly used by model and manager classes. `GeometryUtils` contains functions for geometric calculations and used for rendering purposes. `Groups` is mostly composed of group name constants used for data group management. `ImageUtils` is to load and process graphical images. `GeneralUtils` provides functions related to general display and calculation tasks.

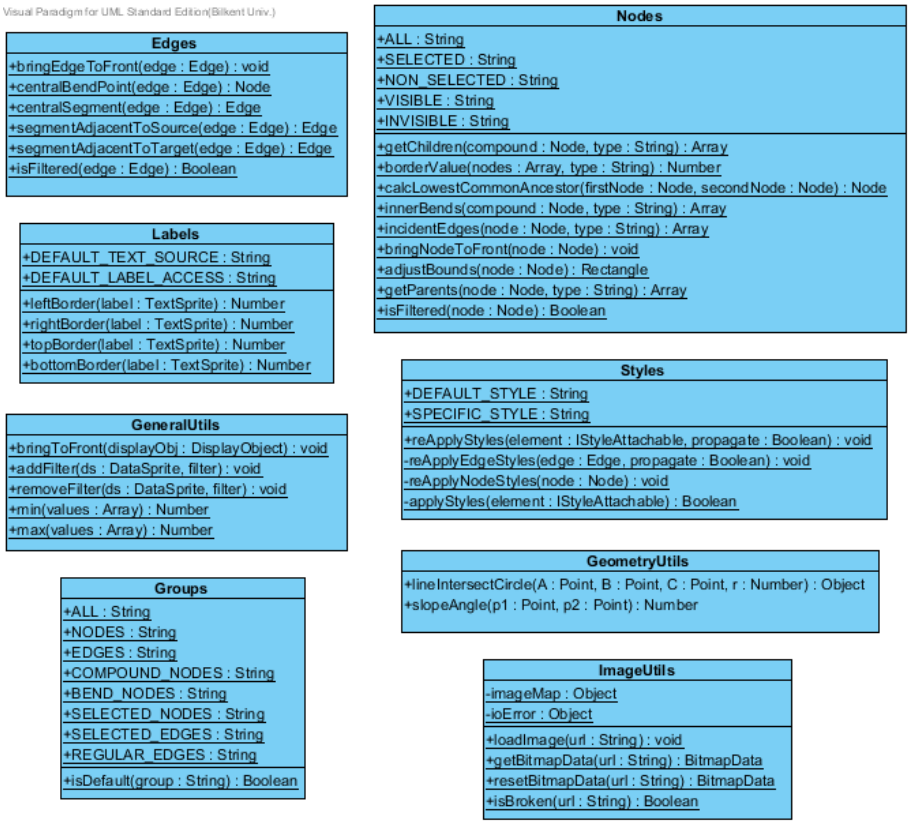


Figure 4.18: Utility classes provided by ChiWeb

# Chapter 5

# Customizing ChiWeb

One of the main design goals of ChiWeb is easy customization. ChiWeb provides various mechanisms to create custom applications for specific domains with minimum effort. The rest of this chapter illustrates customization of ChiWeb in many aspects with implementation examples. All of the example implementations provided in this chapter are taken from a complete sample application that can be accessed through ChiWeb's publicly available source code repository [10].

## 5.1 Introducing New Node and Edge Types

ChiWeb architecture enables easy customization of node and edge user interfaces (UIs). This section describes how to introduce new node and edge types and provides sample implementations to demonstrate the usage of ChiWeb library for UI customization.

### 5.1.1 Custom Nodes

One can introduce new node types by using the customization mechanism provided by ChiWeb. In order to define a custom UI for a new node type, it is

required to introduce a new class by either directly implementing the `INodeUI` interface or extending the default node UI classes provided by ChiWeb such as `RectangularNodeUI` and `CircularNodeUI` (Figure 4.3).

In order to create a new node type, to support gradient colors for example, first it is required to define a custom UI. The code segment below is the implementation of the method `draw` of `GradientRectUI` to enable rendering rectangular nodes with gradient coloring.

```
public override function draw(ds:DataSprite):void
{
  var g:Graphics = ds.graphics;
  var m:Matrix = new Matrix();
  var width:Number = ds.w;
  var height:Number = ds.h;

  if (ds.props.gradientAngle == GradientRectUI.AUTO_ANGLE)
  {
    m.createGradientBox(width, height, Math.atan(width/height));
  }
  else
  {
    m.createGradientBox(width, height, ds.props.gradientAngle);
  }

  g.beginGradientFill(ds.props.gradientType,
    [0xffffff & ds.fillColor, 0xeeeeee],
    [ds.fillAlpha, ds.fillAlpha],
    [32, 255], m,
    ds.props.spreadMethod,
    ds.props.interpolationMethod, 0);

  super.draw(ds);
}
```

Note that, since `GradientRectUI` extends `RectangularNodeUI`, the method above calls the `draw` method of the parent class to draw a rectangular shape. It is also possible to customize the shape of the node by drawing another shape instead of calling the parent's `draw` method. In this case, it may also be required to implement or override the method `intersection` to calculate the clipping points correctly for the incident edges of the node.

After implementing `GradientRectUI`, its instance should be registered to the

system by invoking the `registerUI` method of `NodeUIManager`. For a further customization, a new group style for gradient nodes can also be introduced as it is demonstrated below.

```
// register gradient node UI
NodeUIManager.registerUI(Constants.GRADIENT_RECT, GradientRectUI.instance);

// add a new data group for gradient nodes
graph.addGroup(Constants.GRADIENT_RECT);

// define a group style for gradient nodes
style = {shape: Constants.GRADIENT_RECT,
  w: 80,
  h: 60,
  alpha: 0.7,
  fillColor: 0xff3e66f0,
  lineWidth: 2,
  labelFontWeight: "bold",
  labelFontStyle: "italic",
  gradientType: GradientType.LINEAR,
  gradientAngle: GradientRectUI.AUTO_ANGLE,
  spreadMethod: SpreadMethod.REFLECT,
  interpolationMethod: InterpolationMethod.RGB};

// add new style for the data group
graphStyleManager.addGroupStyle(Constants.GRADIENT_RECT, new Style(style));
```

Note that, style properties such as `gradientType`, `gradientAngle`, `spreadMethod` and `interpolationMethod` are used within the `draw` method of `GradientRectUI`. One can define such custom style properties to increase customizability of a new node UI.

After adding the new UI, the new data group and the new style, to create a gradient node it is sufficient to add a node to the data group defined for gradient nodes by invoking the `addToGroup` method of `Graph`. As soon as a node is added to the corresponding data group, the group style will be attached to that node as described in Section 4.4. Figure 5.1 illustrates a node (*n2*)customized for gradient coloring.

Similar to simple node customization, one can introduce a new compound node UI by implementing `INodeUI` interface or extending default compound node

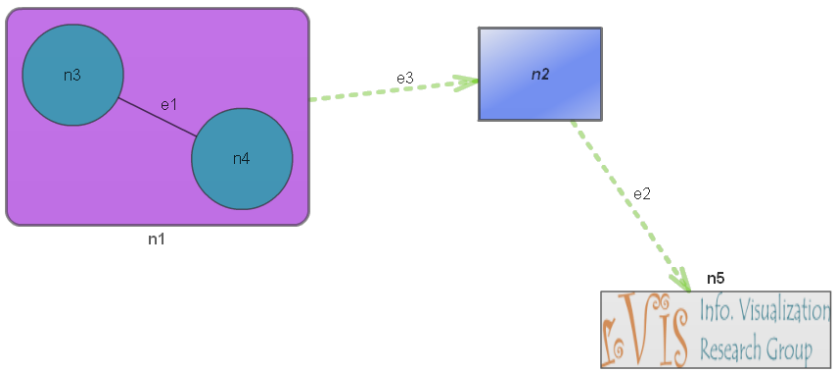


Figure 5.1: An example UI customization with a compound node (n1), a node with gradient coloring (n2), two circular nodes (n3, n4), a node containing an image (n5), and two dashed edges (e2, e3)

UI classes such as `RectangularCompoundUI` and `RoundRectCompoundUI` (Figure 4.3). However, after implementing a custom UI class, its instance should be registered by invoking the `registerUI` method of `CompoundUIManager` instead of `NodeUIManager`.

Node customization can also be achieved by only using custom styles and default UIs provided by ChiWeb without defining a new UI. The code segment below defines a custom style for all compound nodes. Visualization of a compound node (*n1*)having this custom group style is illustrated in Figure 5.1.

```
// define a group style for compound nodes
style = {shape: CompoundUIManager.ROUND_RECTANGLE,
  buttonMode: true, // enables hand cursor when mouse over
  alpha: 0.6,
  fillColor: 0xffa21be0,
  lineWidth: 2,
  labelFontWeight: "bold"};

// add new style for the data group
graphStyleManager.addGroupStyle(Groups.COMPOUND_NODES, new Style(style));
```

### 5.1.2 Custom Edges

It is possible to customize edges in a way similar to node customization. In order to define a custom UI for a new edge type, it is required to introduce a new class

by either directly implementing the `IEddeUI` interface or extending default node UI class `LinearEdgeUI` (Figure 4.3).

In order to introduce a new edge type, to create dashed edges for example, it is required to define a custom UI. The code segment below is the implementation of the methods `setLineStyle` and `draw` of `DashedEdgeUI` to enable rendering edges as dashed lines.

```
public function setLineStyle(ds:DataSprite):void
{
  var onLength:Number = ds.props.onLengthCoeff * ds.lineWidth;
  var offLength:Number = ds.props.offLengthCoeff * ds.lineWidth;

  this.dashedLine = new DashedLine(ds, onLength, offLength);
  this.dashedLine.lineStyle(ds.lineWidth, ds.lineColor, ds.lineAlpha);
}

public function draw(ds:DataSprite):void
{
  var startPoint:Point = ds.props.$startPoint as Point;
  var endPoint:Point = ds.props.$endPoint as Point;

  this.dashedLine.moveTo(startPoint.x, startPoint.y);
  this.dashedLine.lineTo(endPoint.x, endPoint.y);
}
```

After implementing `DashedEdgeUI`, its instance should be registered to the system by invoking `registerUI` method of `EdgeUIManager`. For a further customization, a new group style for dashed edges can also be introduced as it is demonstrated below.

```
// register dashed edge UI
EdgeUIManager.registerUI(Constants.DASHED_EDGE, DashedEdgeUI.instance);

// add a new data group for dashed edges
graph.addGroup(Constants.DASHED_EDGE);

// define a group style for dashed edges
style = {shape: Constants.DASHED_EDGE,
  alpha: 0.8,
  targetArrowType: ArrowUIManager.SIMPLE_ARROW,
  onLengthCoeff: 1,
  offLengthCoeff: 2,
```

```
  lineColor: 0xff66cc12,
  lineAlpha: 0.5,
  lineWidth: 3};

// add new style for the data group
graphStyleManager.addGroupStyle(Constants.DASHED_EDGE, new Style(style));
```

Note that, style properties such as `onLengthCoeff` and `offLengthCoeff` are used within the `setLineStyle` method of `DashedEdgeUI`. Similar to node customizing, defining such style properties increases customizability of a new edge UI.

As soon as an edge is added to the data group defined for dashed edges, the group style will be attached to that edge. This results in visualization of that edge as a dashed edge (Figure 5.1).

## 5.2 Customizing Functionality and Behavior

Easy customization of functionality and behavior with respect to user interaction is crucial for an interactive visualization. By providing sample implementations of custom controls and listeners, this section explains how to extend default control mechanism of ChiWeb (Section 4.5) to customize behavior.

### 5.2.1 Controlling Node and Edge Creation

`ClickControl` allows to add a new node by clicking on the canvas when the `ADD_NODE` flag is active, and dispatches a `ControlEvent` of type `ADDED_NODE` after adding a new node as described in Section 4.5.1. However, `ClickControl` always adds a simple node with a default style. Similarly, it always adds a default edge when the `ADD_EDGE` state is active. In order to customize the behavior of the application to add different types of nodes or edges with mouse clicks, custom controls and flags should be introduced to the system.

One can start by defining a new control class, say `CreationControl`, which extends `EventControl` to customize node and edge creation. The code segment below is the implementation of the method `attach` within `CreationControl`. This function registers listeners for control events of type `ADDED_NODE`, `ADDED_EDGE` and `ADDING_EDGE` to perform additional tasks for customization of the behavior.

```
public override function attach(obj:InteractiveObject):void
{
  super.attach(obj);

  if (obj != null)
  {
    obj.addEventListener(ControlEvent.ADDED_NODE, onAddNode);
    obj.addEventListener(ControlEvent.ADDED_EDGE, onAddEdge);
    obj.addEventListener(ControlEvent.ADDING_EDGE, onAddingEdge);
  }
}
```

By using the information provided by the `ControlEvent`, the listener function `onAddNode` adds the new node to the data group defined for gradient nodes if the `ADD_GRADIENT` flag is set as shown below.

```
protected function onAddNode(evt:ControlEvent):void
{
  var node:Node = evt.info.sprite as Node;

  if (this.stateManager.checkState(Constants.ADD_GRADIENT))
  {
    // adds the node to the corresponding group
    this.graphManager.graph.addToGroup(Constants.GRADIENT_RECT, node);
  }

  // define a node-specific label using its id property
  node.data.label = node.data.id;
}
```

The listener function `onAddingEdge` is to add a simple animation to visualize the state between two clicks of edge adding process. This function listens to the `ADDING_EDGE` event and registers another listener to trace mouse movements for edge animation.

```
protected function onAddingEdge(evt:ControlEvent):void
{
  // update source node
  this._sourceNode = evt.info.sprite;

  // enable preview edge
  (this.object as DisplayObjectContainer).addChild(this._previewEdge);

  // add mouse listener for the animation of the preview edge
  this.object.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
}
```

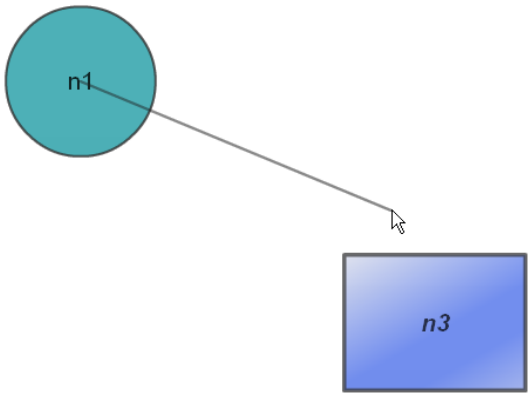


Figure 5.2: Animation of the preview edge during edge creation process

Actual edge animation is performed by the listener function `onMouseMove` as given below. This function renders a preview edge from the center of the source node to the current location of the mouse whenever a `MOUSE_MOVE` event is dispatched (Figure 5.2).

```
protected function onMouseMove(evt:MouseEvent):void
{
  var g:Graphics = this._previewEdge.graphics;

  var startX:Number = this._sourceNode.x;
  var startY:Number = this._sourceNode.y;

  var endX:Number = this.object.mouseX;
  var endY:Number = this.object.mouseY;

  g.clear();
  g.lineStyle(1, 0xff222222, 0.5);
  g.moveTo(startX, startY);
  g.lineTo(endX, endY);
}
```

Edge creation is controlled by the listener function `onAddEdge` with an approach similar to node creation control as shown in the code segment below. In addition to code related to dashed edge creation, this function also stops animation of a preview edge for edge adding process.

```
protected function onAddEdge(evt:ControlEvent):void
{
  var edge:Edge = evt.info.sprite as Edge;

  // remove listeners (for the animation of preview edge)
  this.object.removeEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);

  // disable preview edge
  this._previewEdge.graphics.clear();
  (this.object as DisplayObjectContainer).removeChild(this._previewEdge);

  if (this.stateManager.checkState(Constants.ADD_DASHED_EDGE))
  {
    // adds the edge to the corresponding group
    this.graphManager.graph.addToGroup(Constants.DASHED_EDGE, edge);
  }

  // define an edge-specific label using its id property
  edge.data.label = edge.data.id;
}
```

After implementing the custom control `CreationControl`, its instance should be added to the system by invoking the `addControl` method of `ControlCenter` as shown below. Note that, it is also a required to initialize custom state flags used by `CreationControl`.

```
// initialize creation control
var creationControl:EventControl = new CreationControl();

// add custom control to the system
controlCenter.addControl(creationControl);

// initialize custom flags
controlCenter.setState(Constants.ADD_GRADIENT, false);
controlCenter.setState(Constants.ADD_DASHED_EDGE, false);
```

### 5.2.2 Customizing Key and Mouse Behavior

`KeyControl`, which is the default control for keyboard events, is designed to control selection of multiple nodes (Section 4.5.4). In order to assign other tasks for specific keys, one can define a custom control by extending `KeyControl` and overriding listener functions `onKeyDown` and `onKeyUp`. The code segment below demonstrates implementation of listener functions in a custom control class `KeyPanControl` to enable panning with arrow keys.

```
protected override function onKeyDown(evt:KeyboardEvent):void
{
  var amount:Number = this.graphManager.globalConfig.getConfig(
    Constants.PAN_AMOUNT);

  if (evt.keyCode == Keyboard.RIGHT)
  {
    // pan to right by the amount set in global config
    this.graphManager.panView(-amount, 0);
  }
  else if (evt.keyCode == Keyboard.LEFT)
  {
    // pan to left by the amount set in global config
    this.graphManager.panView(amount, 0);
  }
  else if (evt.keyCode == Keyboard.UP)
  {
    // pan to up by the amount set in global config
    this.graphManager.panView(0, amount);
  }
  else if (evt.keyCode == Keyboard.DOWN)
  {
    // pan to down by the amount set in global config
    this.graphManager.panView(0, -amount);
  }
}

protected override function onKeyUp(evt:KeyboardEvent):void
{
  // do nothing: this function is overridden to disable super.onKeyUp
}
```

Note that the overridden listener function `onKeyDown` accesses a custom configuration parameter `PAN_AMOUNT` to determine the amount of pan for each key

stroke. Required actions to activate `KeyPanControl` after implementing it, including defining a custom global configuration, are demonstrated below.

```
// initialize KeyPanControl
var keyPanControl:EventControl = new KeyPanControl();

// define a custom global configuration parameter
graphManager.globalConfig.addConfig(Constants.PAN_AMOUNT, 20);

// add custom control to the system
controlCenter.addControl(keyPanControl);
```

Similar to customization of the keyboard behavior, ChiWeb also enables customization of the mouse behavior. One can define a custom listener for a specific mouse event and add it as a custom listener by invoking the `addCustomListener` method of `ControlCenter` as shown below.

```
controlCenter.addCustomListener("showInspector",
  MouseEvent.DOUBLE_CLICK,
  showInspector,
  NodeSprite);
```

This registers the function `showInspector` as a custom listener with a name *showInspector* for the mouse event `DOUBLE_CLICK`. The `showInspector` function opens an inspector window (Figure 5.3) if the event target is a `NodeSprite`. The implementation of the listener function `showInspector` is given below.

```
protected function showInspector(event:MouseEvent):void
{
  // set target node before creation
  NodeInspector.targetNode = event.target as NodeSprite;

  // create an inspector window
  var inspector:IFlexDisplayObject = this.showWindow(NodeInspector, false);

  // update inspector position
  inspector.x = this.rootContainer.mouseX;
  inspector.y = this.rootContainer.mouseY;
}

protected function showWindow(window:Class,
```

```
  modal:Boolean = true):IFlexDisplayObject
{
  // create and display a pop up window for the given parameters
  var panel:IFlexDisplayObject =
    PopUpManager.createPopUp(this.rootContainer, window, modal);

  return panel;
}
```

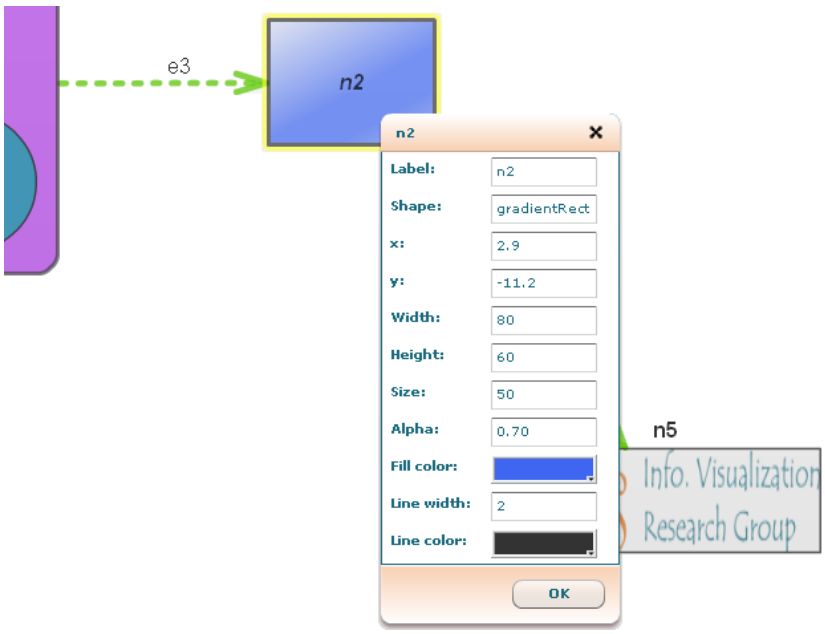


Figure 5.3: An inspector window to display node information

Although in most cases it is enough to implement a single listener function for a simple mouse event such as hover or double-click, one can also define a custom control class for an advanced control over the mouse behavior instead of registering a listener function as demonstrated in the following section.

### 5.2.3 An Advanced Control Over The Mouse Behavior

This section provides a sample implementation of a custom control class `MarqueeZoomControl` that contains an advanced controlling mechanism over several mouse events. Since marquee zoom is performed by *selecting* a specific region on the graph canvas, `MarqueeZoomControl` extends `SelectControl` (Section 4.5.2) and overrides its required methods to enable marquee zoom.

In order to customize mouse behavior, `MarqueeZoomControl` overrides listener functions `onMouseDown`, `onMove` and `onMouseUp` which listen to `MOUSE_DOWN`, `MOUSE_MOVE` and `MOUSE_UP` events respectively.

The function `onMouseDown` initiates the selection process by adding required listeners and initializing the enclosing rectangle as shown below. Note that, this function checks the state of a custom flag `MARQUEE_ZOOM` to initiate the process.

```
protected override function onMouseDown(evt:MouseEvent):void
{
  if (this.object != null &&
      this.stateManager.checkState(Constants.MARQUEE_ZOOM) &&
      !(evt.target is DataSprite))
  {
    this.object.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    this.object.addEventListener(MouseEvent.MOUSE_MOVE, onMove);
    this.initEncloser();
  }
}
```

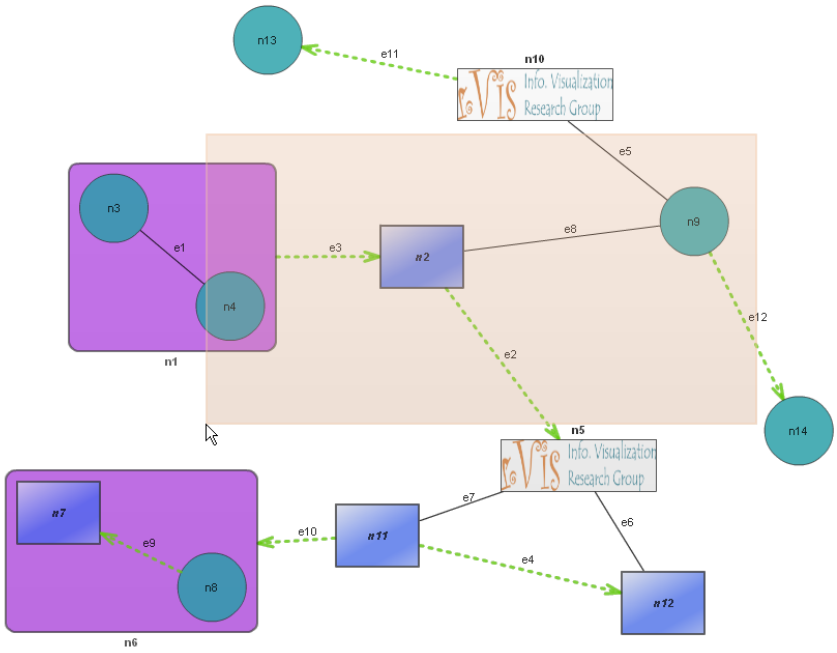


Figure 5.4: Selecting a rectangular area on the canvas to perform marquee zoom

The enclosing rectangle (Figure 5.4) is a visual component to indicate the target area of the zoom operation. This rectangle is updated by `onMove` function whenever a mouse movement is captured as demonstrated below.

```
protected override function onMove(evt:MouseEvent):void
{
  if (this._enclosing &&
      this.stateManager.checkState(Constants.MARQUEE_ZOOM))
  {
    this._enclosingRect.width = this.object.mouseX - this._enclosingRect.x;
    this._enclosingRect.height = this.object.mouseY - this._enclosingRect.y;
    this.renderEncloser();
  }
}
```

The actual zoom operation is performed after the mouse button is released (Figure 5.5). The listener function `onMouseUp` performs the zoom after removing listeners and the enclosing rectangle as shown below.

```
protected override function onMouseUp(evt:MouseEvent):void
{
  // when mouse up, the enclosing rectangle becomes inactive
  this._enclosing = false;

  if (this.object != null)
  {
    // remove the shape
    (this.object as DisplayObjectContainer).removeChild(this._enclosingShape);

    // remove listeners
    this.object.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
    this.object.removeEventListener(MouseEvent.MOUSE_MOVE, onMove);

this.marqueeZoom();
  }
}

protected function marqueeZoom():void
{
  var centerX:Number = _enclosingRect.x + _enclosingRect.width / 2;
  var centerY:Number = _enclosingRect.y + _enclosingRect.height / 2;

  var amountX:Number = -this.object.x - centerX * this.object.scaleX;
  var amountY:Number = -this.object.y - centerY * this.object.scaleY;
```

```
  // center the view to the center of the enclosing rectangle
  this.graphManager.panView(amountX, amountY);

  // zoom to fit to the scale of enclosing rectangle

  var scaleX:Number = this.graphManager.view.parent.width /
  Math.abs(this._enclosingRect.width);

  var scaleY:Number = this.graphManager.view.parent.height /
  Math.abs(this._enclosingRect.height);

  this.graphManager.zoomView(Math.min(scaleX, scaleY) /
  this.object.scaleX);
}
```

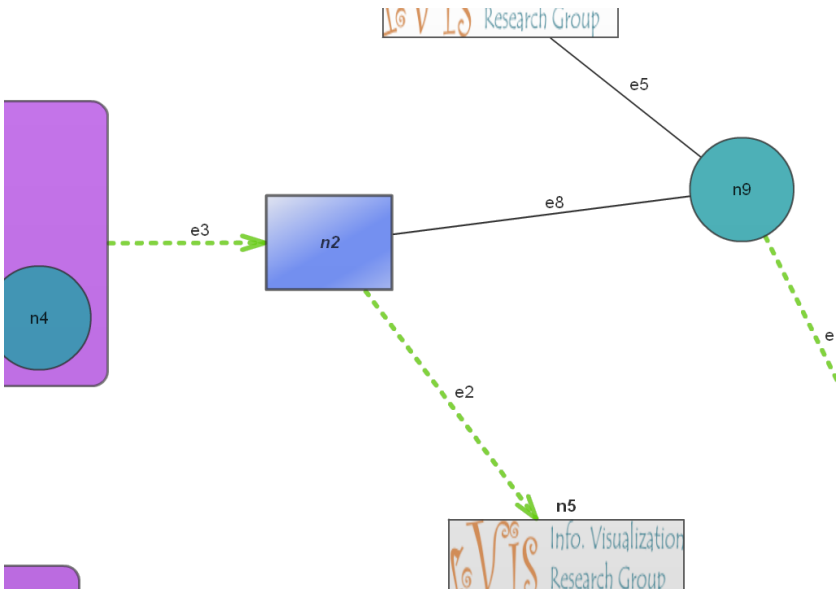


Figure 5.5: Canvas view after performing marquee zoom on a selected area

Similar to the activation of other custom controls, it is required to add an instance of `MarqueeZoomControl` to `ControlCenter` by invoking `addControl` method, and to initialize the custom flag `MARQUEE_ZOOM` as demonstrated below.

```
// init marquee zoom control
var marqueeZoomControl:EventControl = new MarqueeZoomControl();

// add custom control to the system
controlCenter.addControl(marqueeZoomControl);

// initialize custom state flag
controlCenter.setState(Constants.MARQUEE_ZOOM, false);
```

## 5.3 Custom Layout Styles

ChiWeb allows integration of custom layout algorithms for specific needs. One can define a custom layout by extending the base layout class `LayoutOperator` (Section 4.6). This section gives a sample implementation of a custom layout which enables remote usage of `ChiLay` as described in Section 2.4.

`RemoteLayout` (Figure 5.6) is implemented to request `CoSE` and `CiSE` layouts from a remote server on which `ChiLay` is deployed.

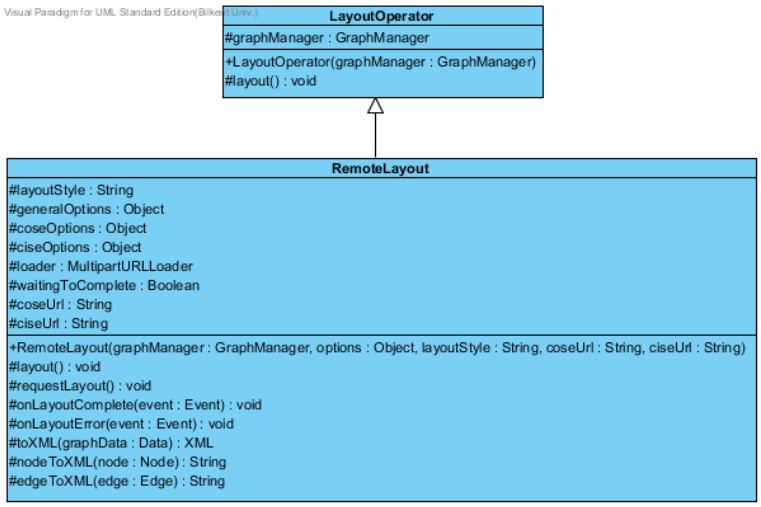


Figure 5.6: Custom layout design for remote use of ChiLay

The constructor of `RemoteLayout` adds listeners to the URL loader for the events `COMPLETE` and `IO_ERROR` as well as initializing the layout parameters as shown below. These event listeners are required to capture the response object returned by the server.

```
public function RemoteLayout(graphManager:GraphManager = null,
  options:Object = null,
  layoutStyle:String = DEFAULT_LAYOUT_STYLE,
  coseUrl:String = DEFAULT_COSE_URL,
  ciseUrl:String = DEFAULT_CISE_URL)
{
  super(graphManager);

  // init loader
this._loader = new MultipartURLLoader();
```

```
  // init layout style
  this._layoutStyle = layoutStyle;

  // init URLs
  this._coseUrl = coseUrl;
  this._ciseUrl = ciseUrl;

  // init options
  if (options == null)
  {
     // initialization of default options is omitted for the sake of simplicity
     ...
  }
  else
  {
    // assuming options object has a correct structure
    this._generalOptions = options.general;
    this._coseOptions = options.cose;
    this._ciseOptions = options.cise;
  }

  // add listeners for the loader
  this._loader.addEventListener(Event.COMPLETE, onLayoutComplete);
  this._loader.addEventListener(IOErrorEvent.IO_ERROR, onLayoutError);

  // init flag
  this._waitingToComplete = false;
}
```

Layout request is sent to the server by the `requestLayout` method, which is invoked by the overridden method `layout` as shown below. For the sake of simplicity, only a part of `requestLayout` that is related to `CoSE` layout is given below.

```
protected override function layout():void
{
  // request layout only if not waiting for a previous layout to complete
  if (!this._waitingToComplete)
  {
    this.requestLayout();
  }
}

protected function requestLayout():void
{
  var bytes:ByteArray = new ByteArray();
```

```
  var xml:XML = this.toXML(this.graphManager.graph.graphData);

  bytes.writeUTFBytes(xml.toXMLString());
  this._loader.addFile(bytes, "graph");

  // append general options
  var go:Object = this._generalOptions;

  this._loader.addVariable("layoutQuality", go.quality);
  this._loader.addVariable("animateOnLayout", go.animateOnLayout);
  this._loader.addVariable("incremental", go.incremental);
  this._loader.addVariable("createBendsAsNeeded", go.createBends);
  this._loader.addVariable("uniformLeafNodeSizes", go.uniformLeafNodeSize);

  if (this._layoutStyle == "CoSE")
  {
    var co:Object = this._coseOptions;

    this._loader.addVariable("springStrength", co.springStrength);
    this._loader.addVariable("repulsionStrength", co.repulsionStrength);
    this._loader.addVariable("gravityStrength", co.gravityStrength);
    this._loader.addVariable("gravityRange", co.gravityRange);
    this._loader.addVariable("compoundGravityStrength", co.compoundGravityStrength);
    this._loader.addVariable("compoundGravityRange", co.compoundGravityRange);
    this._loader.addVariable("idealEdgeLength", co.idealCoSEEdgeLength);
    this._loader.addVariable("smartRepulsionRangeCalc", co.frGridVariant);
    this._loader.addVariable("smartEdgeLengthCalc", co.smartEdgeLengthCalc);
    this._loader.addVariable("multiLevelScaling", co.multiLevelScaling);
    this._loader.addVariable("layoutStyle", "cose");

    this._waitingToComplete = true;
    this._loader.load(this._coseUrl);
  }
}
```

Note that `requestLayout` calls the function `+toXML` which creates an XML representation of the graph that conforms to the XML schema (Section 2.4) defined for `ChiLay`.

After requesting a remote layout from the server, ChiWeb waits for a `COMPLETE` event to be dispatched by the URL loader. Upon completion of the layout on the server side, the listener function `onLayoutComplete` parses the response XML and updates the layout and the view on the client side as demonstrated in the code segment below.

```
protected function onLayoutComplete(event: Event):void
{
  var response:XML = XML(this._loader.getResponse());

  for each(var xmlNode:XML in response..node)
  {
    var node:Node = this.graphManager.graph.getNode(xmlNode.@id);
    node.x = Number(xmlNode.bounds.@x);
    node.y = Number(xmlNode.bounds.@y);
  }

  // reset flag
  this._waitingToComplete = false;

  // since this is a remote layout, it will complete after graph
  // manager updates the view, so view should also be updated here
  this.graphManager.view.update();
}
```

After implementing `RemoteLayout`, its instance can be introduced to the system by invoking the `setLayout` method of `GraphManager` as shown below.

```
// create remote layout operator
var remoteLayout:RemoteLayout = new RemoteLayout();

// set layout operator of the graph manager
graphManager.setLayout(remoteLayout);
```

## 5.4 Customizing GUI

Since ChiWeb is built on ActionScript technology, graphical user interface (GUI) elements such as menus, buttons, combo boxes and panels can be customized directly by using GUI components provided by ActionScript. This section demonstrates customization examples for GUI components created by taking advantage of MXML and CSS support of ActionScript.

### 5.4.1 Customizing Menus

One can define a menu with custom items by using MXML syntax as demonstrated in the code segment below.

```
<mx:MenuBar id="mainMenu" width="100%"
            dataProvider="{menuItems}"
            labelField="label"
            itemClick="handleMenuCommand(event)"/>
```

The property `id` is to give a unique identifier name to the menu, `width` indicates the horizontal space that will be used by the menu bar, and `itemClick` defines the listener function for the mouse click event. `dataProvider` is the source object for the menu content, and `labelField` defines the source of the label text of each menu item within `dataProvider`. A sample definition of a data provider object is given below.

```
private var menuItems: ArrayCollection = new ArrayCollection(
[
  { label: "File", children:
    [
      { label: "New" },
      { label: "Sample"},
      { label: "Load" },
      { label: "Save" }
    ]
  },
  { label: "View", children:
    [
      { label: "Actual Size" },
      { label: "Zoom In" },
      { label: "Zoom Out" },
      { label: "Fit in Canvas" }
    ]
  },
  { label: "Styles", children:
    [
      { label: "Compound Node Style" },
      { label: "Circular Node Style" },
      { label: "Gradient Node Style" },
      { label: "Image Node Style" },
      { label: "Simple Edge Style" },
      { label: "Dashed Edge Style" }
```

```
    ]
  },
  { label: "Layout", children:
    [
      { label: "Remote Layout Properties" },
      { label: "Local Layout Properties"}
    ]
  },
  { label: "Help", children:
    [
      { label: "How to Use" },
      { label: "About" }
    ]
  }
]);
```

Figure 5.7 illustrates an application with a custom menu. Upon clicking on a menu item, the listener function `handleMenuCommand` performs the required action specific to that menu item.


Figure 5.7: A sample application with a customized menu bar

### 5.4.2 A Customized Tool Bar

One can define a tool bar with customized buttons and combo boxes using MXML syntax as shown below. Note that, the code segment below illustrates only a part of the tool bar that enables creation of custom node types.

```
<mx:HBox id="toolBar" width="100%"
         horizontalGap="1" paddingBottom="5" paddingTop="22">

  <mx:VBox id="nodeSelectBox" width="{nodeComboBox.width}">
    <mx:ComboBox id="nodeComboBox" change="setNodeType(event)" >
      <mx:ArrayCollection>
        <mx:Object label="Circle" state="{Constants.ADD_CIRCULAR_NODE}"/>
```

```
        <mx:Object label="Gradient" state="{Constants.ADD_GRADIENT}"/>
        <mx:Object label="Image" state="{Constants.ADD_IMAGE_NODE}"/>
        <mx:Object label="Compound" state="{Constants.ADD_COMPOUND_NODE}"/>
      </mx:ArrayCollection>
    </mx:ComboBox>
  </mx:VBox>

  <mx:Button id="addNode"
             icon="@Embed(source='assets/buttons/create-node.png')"
             width="32"
             toolTip="Create Node (of selected type)"
             click="addNode(event)"/>

</mx:HBox>
```

Defining the tool bar as an `HBox` allows to layout its content horizontally. The combo box `nodeComboBox` is to select the type of the node to be created. It contains a collection of objects to provide a list of nodes to select.

The button `addNode` is to activate the `ADD_NODE` state which enables creation of a node by clicking on the canvas as described in Section 5.2.1. The property `icon` is to use a custom image as a button icon, `toolTip` defines the text to be displayed when hovered on the button, and `click` indicates the listener function to be invoked when the button is clicked. For a further customization, to adjust visual appearance of the buttons one can define a CSS as follows:

```
mx|Button {
  width: 32;
  height: 32;
  cornerRadius: 8;
  paddingLeft: 0;
  paddingRight: 0;
  paddingTop: 0;
  paddingBottom: 0;
}
```

Figure 5.8 illustrates a tool bar with custom buttons and combo boxes. Upon clicking on a button, a specific listener function performs the required action.


Figure 5.8: A sample tool bar with custom buttons and combo boxes

### 5.4.3 Custom Panels

One can create a panel with a combination of various GUI components to display information or to take input from the user. The code segment below gives a portion from the implementation of a node inspector.

```
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="vertical"
                width="180"
                title="Node Inspector"
                showCloseButton="true"
                close="onClose(event)">

  <mx:VBox width="100%" height="100%"
           creationComplete="onCreation(event)">
    <mx:HBox>
      <mx:Label id="labelLabel" width="70" text="Label:"/>
      <mx:TextInput id="labelValue" width="80"/>
    </mx:HBox>
    <mx:HBox>
      <mx:Label id="shapeLabel" width="70" text="Shape:"/>
      <mx:TextInput id="shapeValue" width="80"/>
    </mx:HBox>
    <mx:HBox>
      <mx:Label id="xLabel" width="70" text="x:"/>
      <mx:TextInput id="xValue" width="80"/>
    </mx:HBox>
    <mx:HBox>
      <mx:Label id="yLabel" width="70" text="y:"/>
      <mx:TextInput id="yValue" width="80"/>
    </mx:HBox>
    <mx:HBox>
      <mx:Label id="fillColorLabel" width="70" text="Fill color:"/>
      <mx:ColorPicker id="fillColorValue" width="80"/>
    </mx:HBox>
  </mx:VBox>

  <mx:ControlBar width="100%">
    <mx:Spacer width="100%" />
    <mx:Button id="okButton" label="OK" width="70" click="onClose(event)"/>
```

```
  </mx:ControlBar>

</mx:TitleWindow>
```

The component `TitleWindow` enables creating a panel with a title on it. `VBox` element is a container to display components in a vertical alignment. Each `HBox` element within the `VBox` represents a single row with a `Label` and an input field which is either a `TextInput` or a `ColorPicker`. `ControlBar` is a component to display control interface such as buttons. Figure 5.3 illustrates a node inspector with sample input fields and a single control button. Visual appearance of this inspector panel can be customized with a CSS segment as given below.

```
mx|Panel {
  borderColor: #1a6b7f;
  borderAlpha: 1;
  borderThicknessLeft: 1;
  borderThicknessRight: 1;
  borderThicknessTop: 1;
  borderThicknessBottom: 1;
  roundedBottomCorners: false;
  cornerRadius: 10;
  backgroundAlpha: 1;
  highlightAlphas: 0, 0;
  headerColors: #FFFFFF, #F8D0B2;
  footerColors: #F8D0B2, #FFFFFF;
  backgroundColor: #FFFFFF;
  shadowDistance: 1;
  dropShadowColor: #330000;
}
```

# Chapter 6

# ChiWeb Implementation

The aim of this chapter is to focus on the implemetation details of some of the important components provided by ChiWeb. While Section 6.1 gives the key concepts for ChiWeb's compound graph support, Section 6.2 explains the bend point management within ChiWeb.

## 6.1 Compound Graph Support

Flare library supports topological management and interactive visualization of simple graphs by default. However, it requires additional effort to fully support compound graphs. Flare is designed to process a single graph `Data` (Figure 2.4), which contains all nodes and edges, to manage and visualize the graph. It is not easy and feasible to change this architecture to introduce child graph and graph manager concepts as implemented by Chisio [30]. Therefore, ChiWeb manages compound graphs without defining additional subgraphs for compound nodes.

As described in Section 4.1, a `Node` represents both a simple and a compound node. A compound node knows its children, and a child node knows its parent compound node. A compound node can also have other compound nodes as its children. Therefore, it is possible to define a graph topology with arbitrary levels

of nesting.

Each `Node` is treated as a simple node until it is initialized by invoking the `initialize` method. Compound node initialization is automatically performed by ChiWeb before adding a child node into a simple node for the first time. Invoking `initialize` without adding a child node, converts the simple node to an empty compound node. It is also possible to revert a compound node to a simple node by invoking the `reset` method. However, a compound node is reset only if it does not have any children.

By taking advantage of data group concept provided by Flare (Section 2.3.2), ChiWeb defines a data group, named `COMPOUND_NODES` (Figure 4.18), that is specific to compound nodes. This data group allows fast access to compound nodes when it is required to perform an operation on all or some of the compound nodes. As soon as a simple node is initialized, `GraphManager` adds it into the data group of compound nodes. Similarly, when a compound node is reset, it is removed from the group of compound nodes.

Concepts discussed so far are related to the topology and structure of the compound graphs. For a complete compound graph support, graph geometry should also be taken into account. For example, invoking the `addNode` method of a `Node` to add a child node updates the topology of the graph, however, it does not add the given `NodeSprite` as a child component to the parent compound node. Hence, while visualizing the graph, Flare treats children of a compound node as they are graphically independent from their parent compound node. Therefore, for most of the interactive actions such as selecting, dragging, deleting and filtering, it is required to consider additional cases to visualize compound graphs correctly.

As described in Section 4.5.3, while dragging a compound node, all its direct and indirect children should be dragged as well. Since each compound node keeps a list of its direct children, it is straightforward to access all children within a compound. Since the `getNodes` method of a `Node` retrieves only direct children of that node, the `getChildren` function within the utility class `Nodes` (Figure 4.18)

is specifically designed to retrieve all direct and indirect children within a compound node. Apart from the children, it is also required to drag all bend points residing inside a compound node. The utility function `innerBends` of `Nodes` is designed for this purpose. It classifies a bend point as an inner bend of a compound node, if both the target and the source nodes of the edge on which that bend point is located are within that compound node.

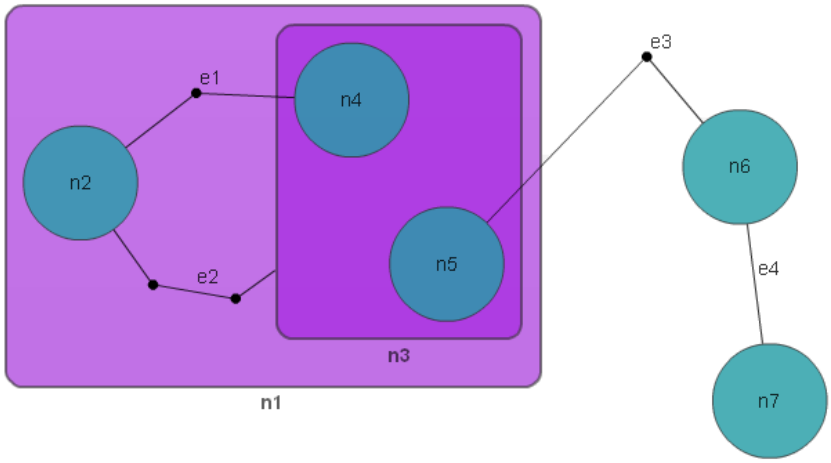


Figure 6.1: A sample graph with a compound node (n1) with 2 direct children (n2 and n3), 2 indirect children (n4 and n5), and 3 inner bends (one on e1, and two on e2)

For the example graph in Figure 6.1, when dragging the compound node *n1*, it requires to drag all its children (*n2*, *n3*, *n4* and *n5*) and inner bends on the edges *e1* and *e2* as well. Note that the bend point on the edge *e3* is not an inner bend for the compound node *n1*, since the target node (*n6*) of *e3* is not a child of *n1*.

When dragging one or more nodes inside a compound node, it is very important to keep compound node bounds consistent with respect to its children. The `updateCompoundBounds` method of `GraphVisualization` (Figure 4.4) helps to keep bounds of compound nodes up to date. This method calculates the bounds enclosing the children of a compound node, and updates the bounds of that compound by also considering the padding values. Figure 6.2 illustrates update of compound bounds after a drag operation. Note that, bounds of both n1 and n3 are affected by this drag operation.

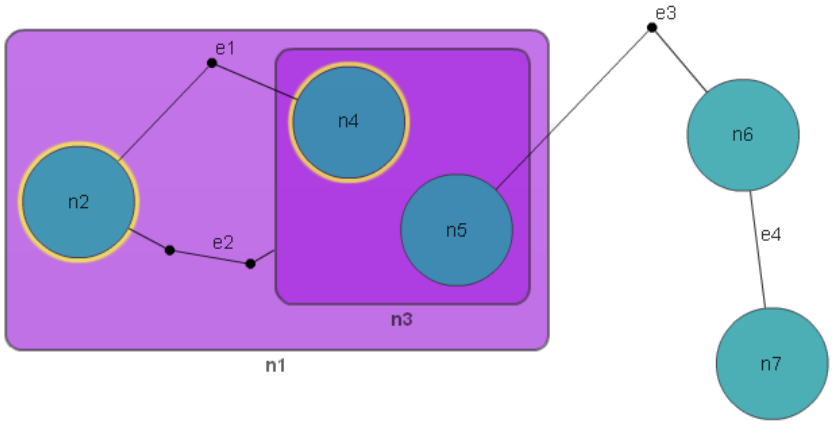


Figure 6.2: Visualization of the sample graph in Figure 6.1 after dragging the selected nodes n2 and n4

Bounds of a compound node should also be updated after interactive actions other than dragging. Deleting or hiding one or more nodes inside a compound, adding a new node into a compound, resizing nodes inside a compound, and performing a layout on the graph are some of those interactive actions that require compound bounds to be updated.

While filtering (hiding) or deleting a compound node, it is also required to consider all children within a compound node. For the example graph in Figure 6.1, when deleting or hiding the compound node *n1*, it is also required to delete or hide the children *n2*, *n3*, *n4* and *n5* as well. When deleting or hiding a node, there is no extra work needed for the regular incident edges of that node, since Flare automatically removes or hides those edges. However, an incident edge with bend points require additional process because of the bend nodes and the segment edges. Section 6.2 explains the management of bend points and segment edges in detail.

## 6.2 Bend Points for Edges

A bend point is a point on an edge which allows linear edges to bend. For each new bend point, a new `Node` is created to represent the bend point. This approach

is preffered to reuse interaction handling and visualization support of Flare as much as possible. Adding a bend point for an edge also introduces a new concept called *segment edge*. As soon as a bend point is added for an edge, the target edge gains a *segmented* structure. As illustrated in Figure 4.2, an edge has a list of bend points and a list of segment edges. On the other hand, a segment edge and a bend point know their parent edges. This structure helps better management of the edges having bend points.

Interactive bend point creation process is handled by the `addBendPoint` method of `GraphManager` (Figure 4.4). If the target of the click event is a regular edge, a new bend node and two new segment edges are added both to the graph and to the corresponding lists of the target edge. The new bend point is always incident with these two new segments. After adding the new bend point and the new segments, the regular edge remains hidden until all bend points are removed from the edge (Figure 6.3). If the target of the click event is a segment edge, then new bend node and the segments are added to the parent of that segment edge instead of the target segment edge, and the target edge is removed from the graph. This ensures new segments and bendpoints to be added only for regular edges. By this way, nested edge structures are avoided to reduce complexity.

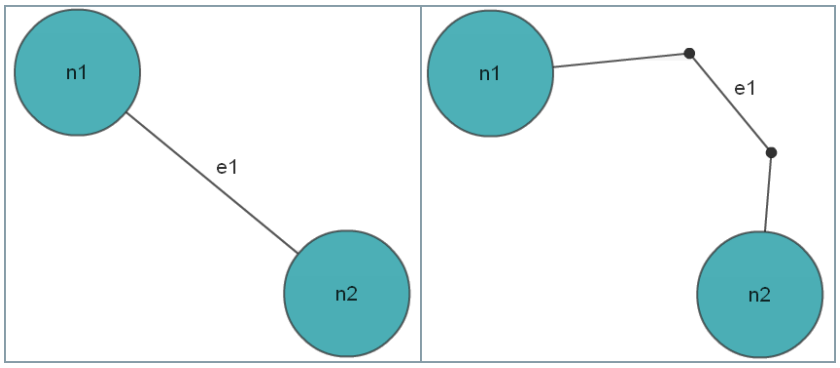


Figure 6.3: An edge with no bend point (left) and the same edge after adding two bend points (right)

While removing a bend point, on the other hand, the bend node and the two incident segments are removed from both the graph and the parent edge. If there is no more bend point remains for the parent edge, then it is displayed as a regular edge again. Otherwise, old two segments are replaced with a new segment. The

`removeBendNode` method of `GraphManager` is implemented to properly remove a bend point from the graph.

While it is possible to remove an arbitrary bend point from an edge, it is not possible to remove a segment edge directly. A segment edge can only be removed when removing a bend point incident with that segment or deleting the parent edge of that segment from the graph. Similarly, it is possible to select or deselect arbitrary bend points on an edge, but an edge segment cannot be selected or deselected separately, since selecting and deselecting a segment edge yields all together selection or deselection of the segments on the corresponding parent edge (Figure 6.4).

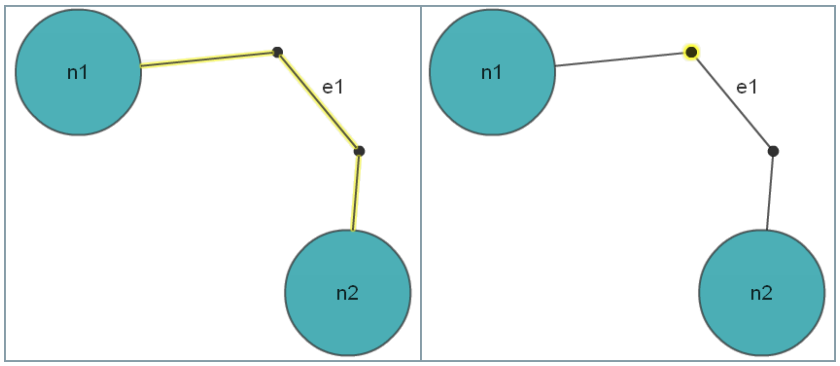


Figure 6.4: While selecting any segment of the segmented edge in Figure 6.3 yields selection of all segments (left), an arbitrary bendpoint can be selected separately (right)

It is required to perform additional cleanup after removing a node that is incident with an edge having bendpoints. Flare normally removes all its incident edges when a node is deleted from the graph, but, since a bend point is a node, only the edge segments which are incident with the deleted node are removed from the graph by Flare. To properly remove an incident edge having bendpoints, all the segments and the bend points on that edge should also be removed to ensure the consistency of the graph topology. `GraphManager` contains a method (`removeNode`) to properly remove a node together with its incident edges, and another method (`removeEdge`) to properly remove an edge together with its segments and bend points.

It is important to keep the visual style of a segment edge consistent with other

segments of its parent edge. Whenever a style is attached to a regular edge by invoking the method `attach` (Figure 4.2), it propagates its style to its segment edges to keep them consistent. It is also possible that a bend point can be added after attaching a style to the regular edge. In this case, the `addSegment` method of the edge attaches its own style to the new segments added during bend point creation.

Since a bend point is also a part of an edge, by default it inherits some of the style properties from its parent such as color and opacity. However, for bend points, it is also possible to define a style completely different from the style of its parent.

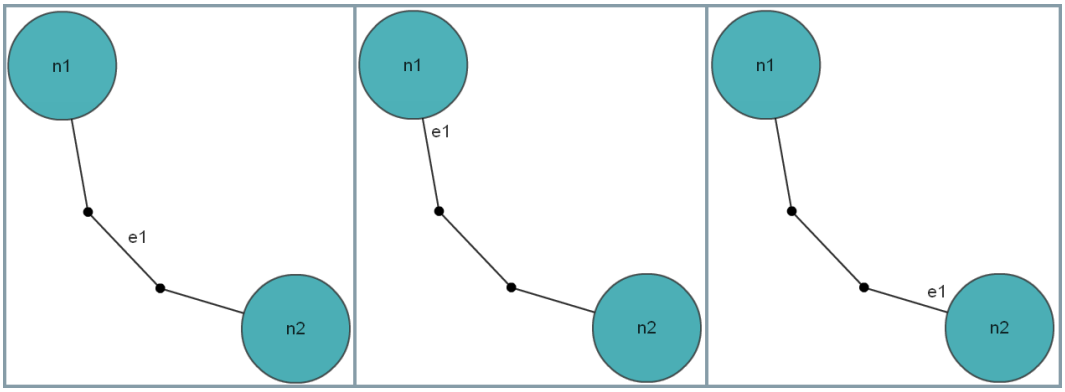


Figure 6.5: Different positions for the same edge label: next to source node (left), in the middle (center), next to target node (right)

Bend points and segment edges should also be taken into account to properly label edges. When there is no bend point, `EdgeLabeler` (Figure 4.15) determines the position of the edge label by only considering the value of the style property `labelPos`. However, when there are bend points on the edge, to position the label according to the value of `labelPos`, `EdgeLabeler` should also determine the middle segment, the middle bend point, the segment adjacent to the source node, or the segment adjacent to the target node. The functions `centralBendPoint`, `centralSegment`, `segmentAdjacentToSource` and `segmentAdjacentToTarget` within the utility class `Edges` (Figure 4.18) are implemented to help positioning of an edge label when there are bend points on an edge. Figure 6.5 illustrates labeling of the same edge with three different values of `labelPos`.

# Chapter 7

# Conclusion

In this thesis, a new, free, open source, and web-based graph visualization framework is introduced for visualization and interactive editing of relational information. This framework, named ChiWeb, is implemented in ActionScript, and it can be run on any web browser supporting Flash Player.

ChiWeb is specifically designed for easy customization of visualization and functionality. This enables building domain-specific applications on ChiWeb framework. In addition to its easy customization mechanism, ChiWeb also provides basic graph visualization and editing functionality such as zooming, panning, node and edge selection, adding/removing nodes and edges, showing/hiding desired elements, and node dragging.

## 7.1 Contribution

Among the non-commercial, web-based graph visualization software, ChiWeb is one of the few tools that fully support compound graphs. Its compound graph support is also implemented for Cytoscape Web, which is a popular graph visualization tool developed by many prestigious universities and research centers. This makes ChiWeb's compound graph support even more valuable contribution

for the field of graph visualization.

Many commercial and non-commercial graph visualization and editing software require a lot of effort to create customized visualization and to provide custom functionality. On the other hand, with its specific design for easy customization, ChiWeb provides a framework for software developers to build domain-specific applications with minimum effort. This makes ChiWeb an important alternative to build a customized graph visualization tool for specific needs.

## 7.2 Future Work

Although ChiWeb provides basic facilities sufficient to create a complete graph visualization tool with many custom components, it can still be improved with additional features. Some of the possible extensions for ChiWeb can be listed as follows:

1. Persistency: ChiWeb currently supports saving and loading a graph in GraphML format only. Support for other formats can be added in order to save the graph information in various forms such as PDF, SVG, XGMML, and PNG.

2. Complexity management: In order to help complexity management, ChiWeb allows node and edge hiding without actually removing them from the graph. One possible extension is to allow merging multiple edges between same source and target nodes into a single edge without removing original edges. Another one is to enable collapsing a compound node into a single node without removing children of that compound. Merging edges and collapsing compounds help to see the big picture more clearly by presenting a less detailed view of the graph. These two extensions will surely increase ChiWeb's complexity management capability.

3. Node resizing: Although it is possible to visualize nodes in various sizes and shapes, ChiWeb does not support interactive node resizing by default.

As a future improvement for user interaction, ChiWeb library can include a default interactive control which enables node resizing by clicking and dragging the bounds of a node.

4. UI annotations: A UI annotation can be defined as an additional UI element attached to a node or an edge. A text label, a small button on the corner of a node, an interactive component attached to a certain location of an edge, and an input text field on a node are some examples of such UI annotations. By default, ChiWeb supports only attaching text labels to the nodes and the edges. It is also possible to attach additional UI annotations by defining custom edge and node UIs. However, as a possible future extension, ChiWeb can provide a better interface for easy integration and management of UI annotations.

5. Algorithms: In addition to basic graph algorithms currently included in ChiWeb or already provided by Flare, as a future improvement, ChiWeb can introduce advanced algorithms for graph-theoretic analysis such as finding shortest paths, neighborhood, and cycles.

6. Rule based graphs: For some specific domains, it may be required to define rules for the graph topology. These rules determine which types of nodes can be connected to which ones. ChiWeb, by default, allows any types of nodes to be connected to each other with any kind of edges. As a future extension, ChiWeb can provide an advanced mechanism for domain-specific topological rule definition in order to create well-defined node and edge types with less effort.

# Bibliography


[1] ActionScript Technology Center. http://www.adobe.com/devnet/actionscript.html, Accessed in March 2012.

[2] Asterisq, Graph Visualization for the Web. http://asterisq.com/, Accessed in March 2012.

[3] Customizing Constellation Roamer. http://asterisq.com/products/constellation/roamer/customization, Accessed in March 2012.

[4] Implementing a custom node renderer for Constellation Custom. http://asterisq.com/products/constellation/framework/tutorials/custom_node_renderer, Accessed in March 2012.

[5] O. Babur, U. Dogrusoz, E. Demir, and C. Sander. Chibe: interactive visualization and manipulation of biopax pathway models. *Bioinformatics*, 26(3):429–431, 2010.

[6] J. Bondy and U. Murty. *Graph Theory with Applications*, volume 290. MacMillan London, 1976.

[7] ChiEd: Chisio Editor. http://www.cs.bilkent.edu.tr/~ivis/chied.html, Accessed in March 2012.

[8] ChiLay: Chisio Layout. http://www.cs.bilkent.edu.tr/~ivis/chilay.html, Accessed in March 2012.

[9] Chisio, Compound or Hierarchical Graph Visualization Tool. http://www.cs.bilkent.edu.tr/~ivis/chisio.html, Accessed in March 2012.

[10] ChisioWeb. `http://sourceforge.net/projects/chisioweb/`, Accessed in March 2012.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT press, 2009.

[12] Cytoscape: An Open Source Platform for Complex Network Analysis and Visualization. `http://www.cytoscape.org/`, Accessed in March 2012.

[13] Cytoscape Web. `http://cytoscapeweb.cytoscape.org/`, Accessed in March 2012.

[14] U. Dogrusoz, E. Erson, E. Giral, E. Demir, O. Babur, A. Cetintas, and R. Colak. Patikaweb: a web interface for analyzing biological pathways through advanced querying and visualization. *Bioinformatics*, 22(3):374–375, 2006.

[15] U. Dogrusoz, Q. Feng, B. Madden, M. Doorley, and A. Frick. Graph visualization toolkits. *Computer Graphics and applications, IEEE*, 22(1):30–37, 2002.

[16] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir. A layout algorithm for undirected compound graphs. *Information Sciences*, 179(7):980–994, 2009.

[17] Flare, Data Visualization for the Web. `http://www.cytoscape.org/`, Accessed in March 2012.

[18] Adobe Flash Player. `http://www.adobe.com/products/flashplayer.html`, Accessed in March 2012.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[20] Gephi: The Open Graph Viz Platform. `http://gephi.org/`, Accessed in March 2012.

[21] The GraphML File Format. `http://graphml.graphdrawing.org/`, Accessed in March 2012.

[22] Graphviz - Graph Visualization Software. `http://www.graphviz.org/`, Accessed in March 2012.

[23] B. Gretarsson, S. Bostandjiev, J. ODonovan, and T. Höllerer. Wigis: A framework for scalable web-based interactive graph visualizations. In *Graph Drawing*, pages 119–134. Springer, 2010.

[24] J. Gross and J. Yellen. *Graph Theory and Its Applications*. CRC press, 2006.

[25] J. Heer and D. Boyd. Vizster: Visualizing online social networks. In *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*, pages 32–39. IEEE, 2005.

[26] Information Visualization, Wikipedia. `http://en.wikipedia.org/wiki/Information_visualization`, Accessed in March 2012.

[27] Oracle Technology Network for Java Developers. `http://www.oracle.com/technetwork/java/index.html`, Accessed in March 2012.

[28] PathVisio. `http://www.pathvisio.org/`, Accessed in March 2012.

[29] Pathway Commons. `http://www.pathwaycommons.org/`, Accessed in March 2012.

[30] Chisio Layout Programmer's Guide. `http://www.cs.bilkent.edu.tr/~ivis/chilay/ChiLay-2.0-PG.pdf`, Accessed in March 2012.

[31] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(2):109–125, 1981.

[32] Tom Sawyer Visualization. `http://www.tomsawyer.com/products/visualization/index.php`, Accessed in March 2012.

[33] uDraw. `http://www.informatik.uni-bremen.de/uDrawGraph/en/home.html`, Accessed in March 2012.

[34] Chisio Editor User's Guide. `http://www.cs.bilkent.edu.tr/~ivis/chilay/ChiLay-2.0-PG.pdf`, Accessed in March 2012.

[35] VisANT: Integrative Visual Analysis Tool for Biological Networks and Pathways. `http://visant.bu.edu/`, Accessed in March 2012.