A web-based, run-time extensible architecture for interactive visualization and exploration of diverse data

Nathan James Conklin
nathan@conklinfamily.net

Thesis submitted to the faculty of
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
December 2002

Advisory Committee:
Chris North, Chair
Stephen Edwards
Naren Ramakrishnan

Keywords: multiple views, visualization coordination, event-based implicit invocation, software architecture

A web-based, run-time extensible architecture for interactive visualization and exploration of diverse data

Nathan James Conklin
nathan@conklinfamily.net

Abstract

Information visualizations must often be custom programmed to support complex user tasks and database schemas. This is an expensive and time consuming effort, even when general-purpose visualizations are utilized within the solution. This research introduces the Snap visualization server and system architecture that addresses limitations of previous Snap-Together Visualization research and satisfies the need for flexibility in information visualizations. An enhanced visualization model is presented that formalizes multiple-view visualization in terms of the relational data model. An extensible architecture is introduced that enables flexible construction and component integration. It allows the integration of diverse data, letting users spend less time massaging the data prior to visualization. The web-based server enables universal access, easy distribution, and the ability to intermix and exploit existing components. This web-based software architecture provides a strong foundation for future multiple-view visualization development.

Acknowledgements

Thanks to Chris North. You have inspired me with creativity and filled me with enthusiasm towards this emerging field of information visualization. I appreciate your advice and support, as well as your motivation to shoot for the stars. Thanks to Stephen Edwards, you helped pushed me to excel both as a student and a teacher. Thanks to Naren Ramakrishnan, you have motivated excellence in my graduate studies and I am very thankful.

Special thanks to those people that were a big part of the Snap team! These contributions are the work of an excellent team. Thanks to those who helped create both the ideas and the software. Varun Saini, you have been wonderful to work with. I wish you the very best. Thanks to Kiran Indukuri and to my VTURCs companions: Amit Nithian, Craig Sinning, Chris Shirk, and Scott Walker. Thanks to my design patterns teammates: Umer Farooq and Aniket Prabhune. Thanks to those who helped create early UI concepts: Matt Clement, Rohit Kelapur, Jeevak Kasarkod, Atul Shenoy.

Many thanks to those who participated with me on the InfoVis team. Thanks to Sandeep Prabhakar and Muthukumar Thirunavukkarasu for your help in discovering new ideas. Sanjini Jayaram, it has been a real treat to share experiences as we struggled together. Thanks to Qing Li.

Most importantly, thanks to the next generation of innovators who will help to achieve our vision while creating a new vision for things to come.

I especially want to thank my wife, Tanya Lynn Conklin. You have provided me with unbounded support and I cannot describe how often I lean on you for that support. You are the reason behind my goals to affect change.

Table of Contents

Table of Figures

Chapter 1 Introduction and Motivation

Research has produced many tools that are effective information visualizations for specific targeted data and tasks [CMS99]. However, most of these tools are not broadly applicable to different situations. For example, Windows Explorer is a common way for users to navigate their file system, whereas Treemaps provide an overview of the directory structure along with visual mappings for attributes (Figure 1).

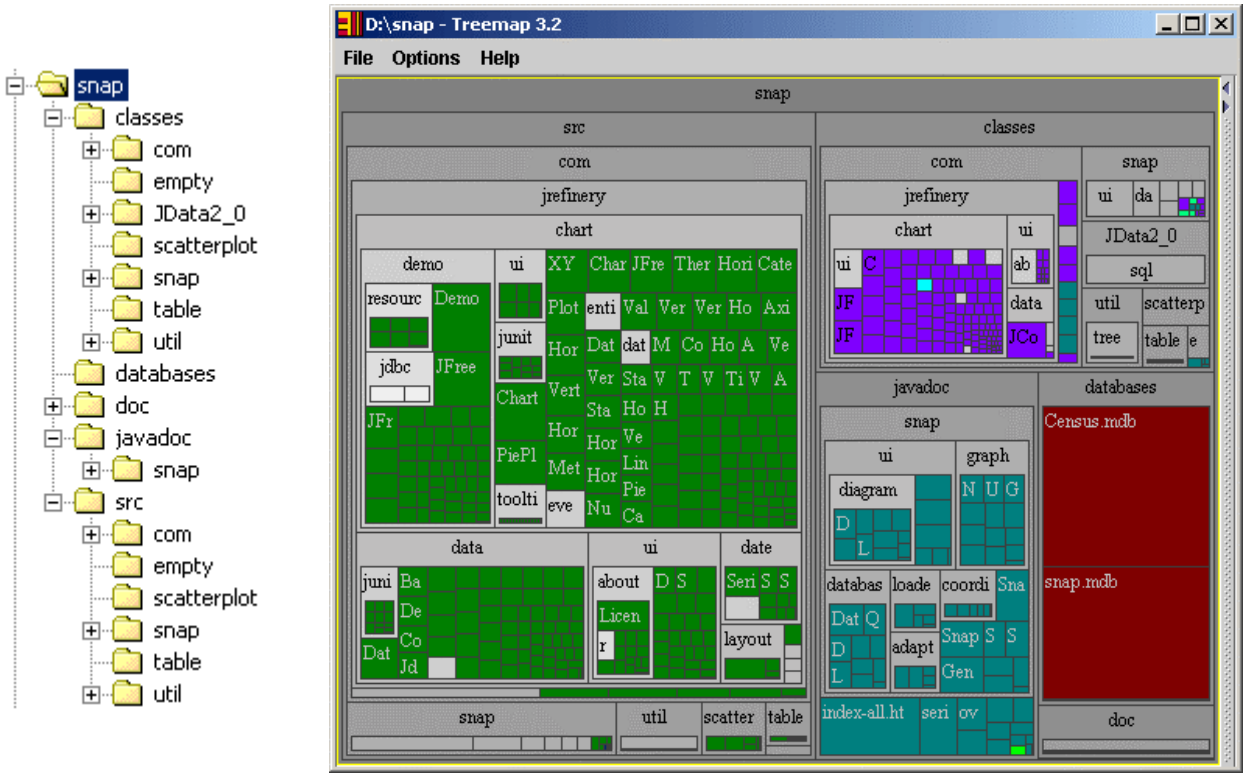


Figure 1 – Exploring a file system using Windows Explorer and Treemaps

Multiple-view visualizations allow user interface designers to take advantage of the strengths of these different tools. Designers can integrate and coordinate these visualizations into a single, more powerful visualization. These coordinated multiple-view visualizations are then able to support a broader range of tasks [BWK00].

However, the design of an appropriate visualization for a given database depends greatly on the data schema and user tasks. Because each data schema is unique, each database requires a unique solution. General-purpose visualization tools (such as Spotfire [AW95]) can be applied, but often are only a partial solution. Custom visualizations are often needed, but they require custom programming. This is both expensive and time consuming, even when general-purpose visualizations are utilized within the solution. There is a need for flexibility in the design and implementation of information visualizations. North and Shneiderman have presented Snap-Together Visualization (Snap) as a potential solution to this problem [NS00a].

1.1 Initial Solution and Benefits

Initial research introduced Snap-Together Visualization as a conceptual model and user interface that allows data users to rapidly construct coordinated multiple-view visualizations without programming [Nor00]. This research provided several contributions:

- **Conceptual model**: an initial model of visualization coordination based on the relational data model.
- **User interface**: a user interface for constructing coordinated-visualization interfaces without programming.
- **Implementation**: an implemented system that integrates the model and user interface, providing software that demonstrates the Snap concept (Figure 2).

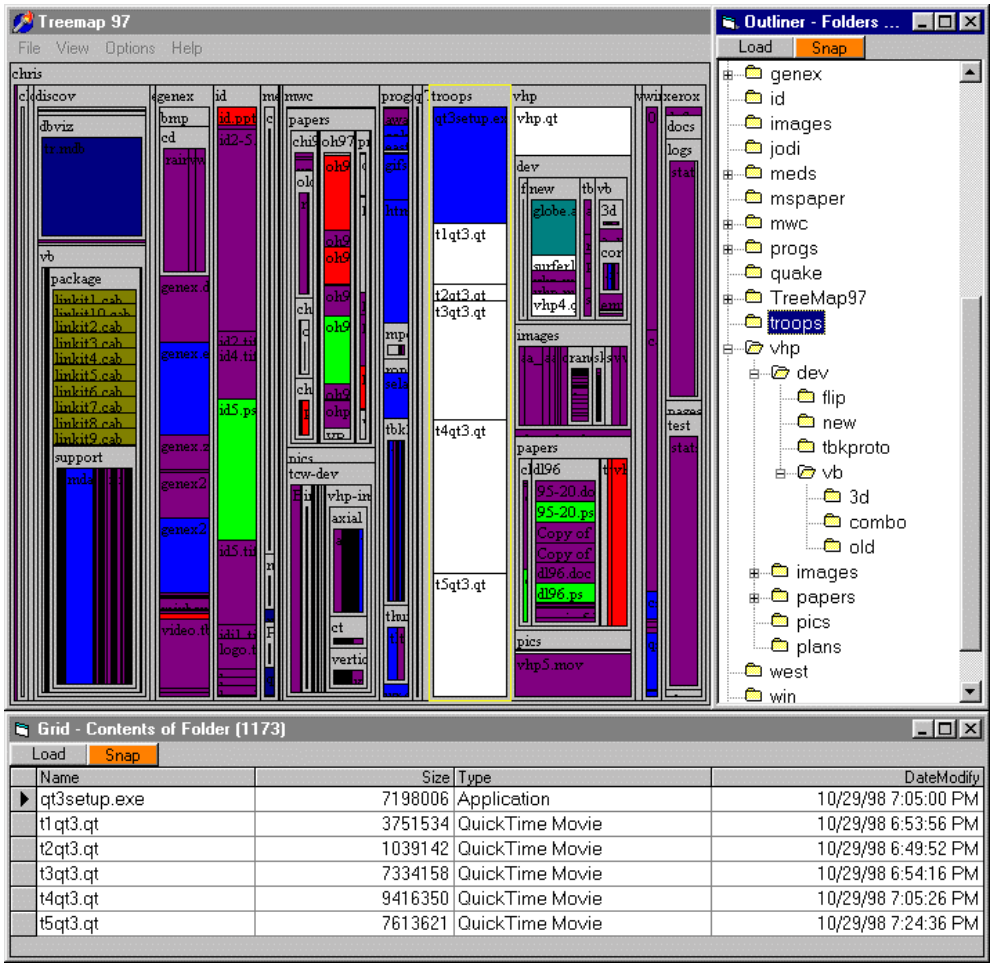


Figure 2 – Snap software enables user construction of a coordinated visualization.

The Snap visualization model is based on composition of multiple-views. In the multiple-view visualization approach, data is displayed in several different views (called *visualization components*). Different components may display the same or different portions of the data. These components can then be tightly coupled (or *coordinated*) in a variety of ways [Nor01] such that interacting with one component causes meaningful effects in others. This produces an integrated composite or *multiple-view visualization* [BWK00]. Figure 2 shows an example of a multiple-view visualization for the exploration of a file system database.

Snap's use of the multiple-views approach has several major motivations:

- Enables flexibility through the composition of multiple views.
- Mimics the way visualization designers often build custom visualization solutions [BWK00][Nor01].
- Enables the use of diverse visualization tools, and supports diverse and complex data.
- Enables reuse of the plethora of visualization tools implemented in the field, as well as automated techniques for constructing individual views such as APT [Mac86].

These contributions have provided many benefits to visualization researchers and developers. Users can dynamically coordinate visualizations to construct a custom user interface without the need for programming. Snap introduces a system that provides flexibility in data, visualizations, and coordinations.

1.1.1 Initial Snap Model

The previous model introduced the notion that a visualization encapsulates a relation [NS00a]. As shown in Figure 3, the Plot encapsulates the Folders relation and a Tabular visualization encapsulates the Files. A Select event would occur in the Plot with the primary key of the folder selected. That key would be passed to the Table and it would load the associated files.

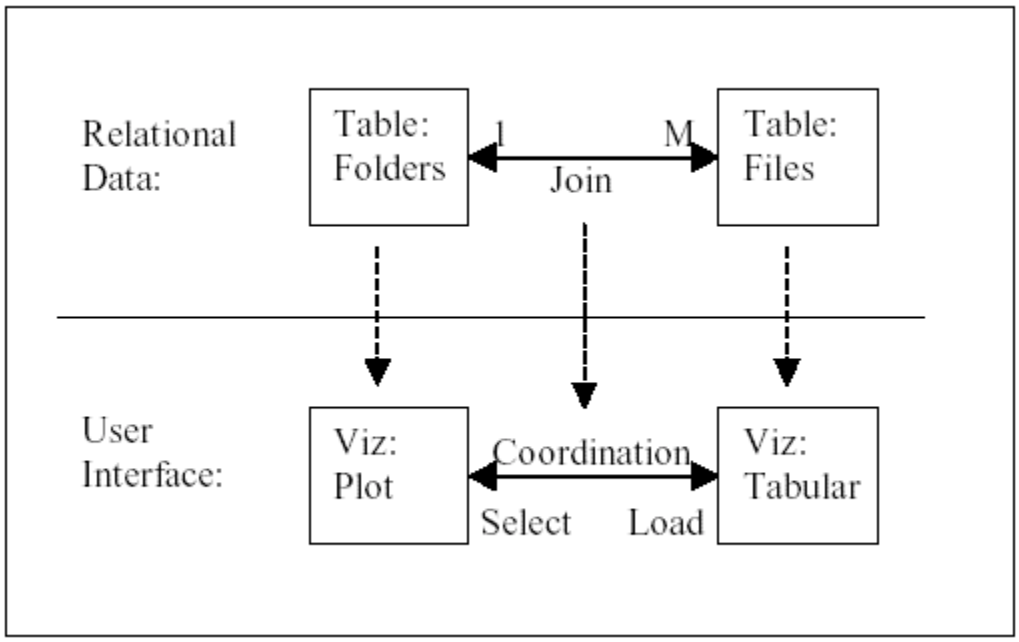


Figure 3 – Overview of the Snap Coordination Model.

1.1.2 Initial Snap User Interface

The initial Snap user interface utilized dialogs for visualization construction. These dialogs allowed a user to specify queries and load the results into a visualization (Figure 4). This is a bottom-up approach for constructing a visualization. The data is selected first and then it is loaded into a visualization.

Snap also provided a direct manipulation user interface for constructing coordinations. A user could drag-and-drop the visualization's Snap button onto a second visualization's Snap button to construct a coordination. The coordination properties dialog (Figure 5) was then used to specify details for the coordination.

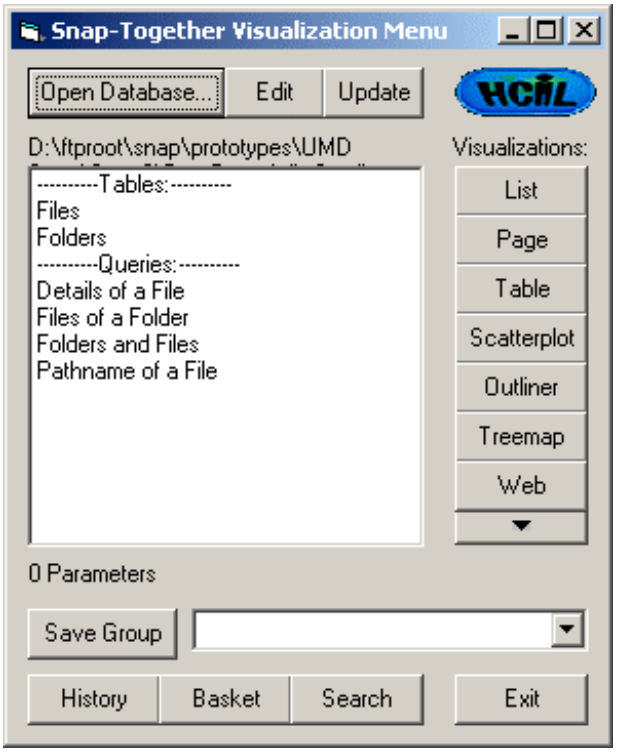


Figure 4 – Previous Snap Visualization Menu used to choose a relation to load into a visualization.

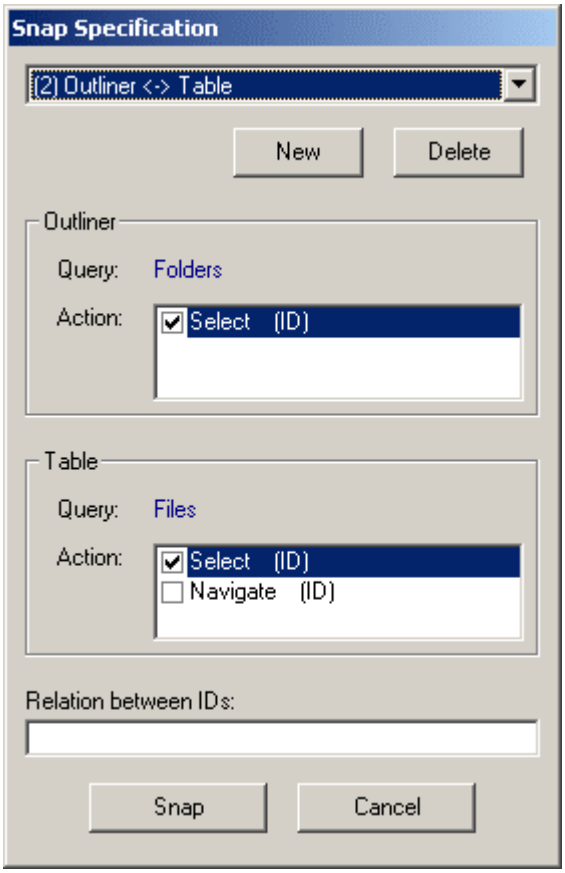


Figure 5 – Previous Snap Specification Dialog used to edit the list of coordinations.

1.1.3 Initial Snap Architecture and Implementation

The initial implementation of Snap was completed utilizing Microsoft COM in the Windows platform. The implementation supported a range of Windows-based visualization components and supported ODBC database connectivity using Microsoft DAO. Snap required a lightweight API for components. The previous implementation provided strong supporting components and fast database connectivity.

1.2 Limitations

Previous work demonstrated that coordinating visualization was conflict free, meaning that cycles in the coordination graph would not cause a conflict in determining which event to fire [Nor00]. However, there were the following limitations to the initial Snap model:

- Snap only supported the coordination of single-tuple actions. This limited visualizations to fire events that occur on a single item. For instance, users could only coordinate brushing of two views for single selected items. There was not support for multiple selection or other multiple-tuple actions, which are a common need in multiple-view visualizations.
- Snap did not utilize the data associations when propagating events. Visualizations that encapsulate different but associated data were required to understand how they were associated. The user was forced to construct a parameterized query that specified the association between the two views. This coupling between views limited the types of coordinations that a user could create and required the user to be very data savvy.

The initial user interface provided many benefits; however, it did not match the Snap model well. Therefore, users had a hard time learning and understanding how the model applied to the system [NS00b]. The initial user interface also had the following limitations:

- Coordinations were hidden from the user. Once views were coordinated the user had no visual cue of the coordination. They would have to interact with a view or edit the coordination list properties in order to see how two views were coordinated.
- No overview is provided to the user. The coordination list properties dialog (Figure 5) was the only presentation of how views were connected. This was a details-only interface for

viewing the coordinations. The coordinations between views in a multiple-view visualization can be complex and a visual overview would help users to understand the constructed visualization [NS00b].

The initial architecture for Snap provided many supporting visualizations and rapid database connectivity. However, the architecture had the following limitations:

- Snap provided a static group of visualization components. Adding new visualization components required some authoring and recompiling of the Snap system. This limited the developer's ability to create visualization components for Snap.
- Snap required a large install base. Each component had to be installed on the machine before Snap could utilize it. This installation requirement, along with the previous limitation, greatly limited the ability to distribute new visualization components.
- The Snap architecture provided no ability for sharing of custom-built visualizations. Once a user built a custom visualization, they were unable to distribute the visualization to others. This limited the power of the multiple-view visualization because a user had to rebuild the visualization in order to reuse it.
- The Snap architecture provided limited database access. Users had limited access to network databases and had no ability to connect to multiple databases. This limited the users ability to integrate data into a single visualization.

1.3 Scenario

This scenario depicts the Snap architecture and visualization server. It demonstrates how Snap can be used to construct a multiple-view visualization to explore US Census data. A user starts by pointing their browser to the Snap-Together Visualization web server. When the browser starts Snap, it loads a frame-based interface with the Snap configuration panel in the left frame and database connectivity documentation in the right frame (Figure 6). The user will start by connecting Snap to their database.

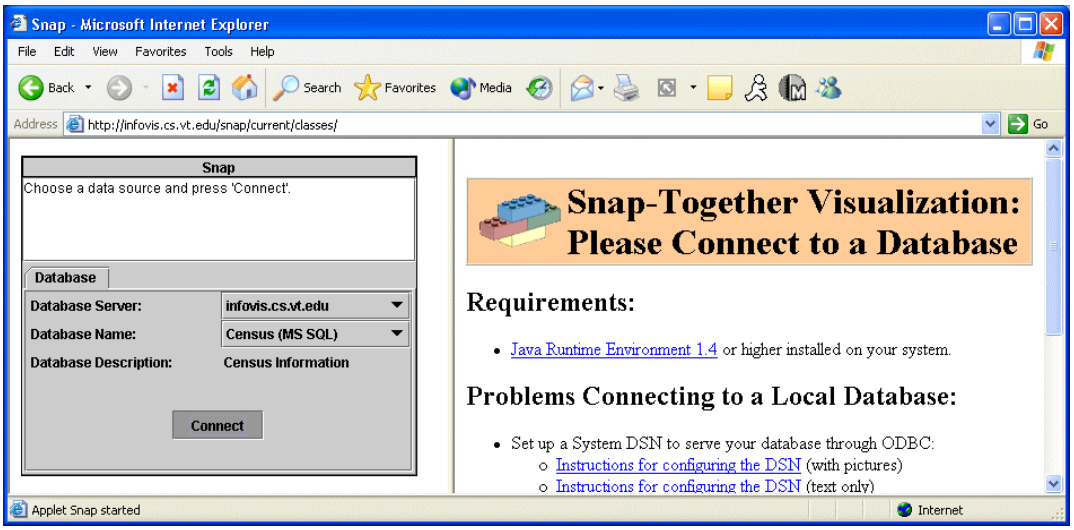


Figure 6 – A user connects Snap to a network database in order to begin visualization construction.

In this example, the user chooses to connect to a network database that publishes US Census statistics about the US states and counties. When a user connects to the database, Snap retrieves the database's schema by querying for both the relations and associations stored within the database. Once connected, the right frame displays the visualization workspace. Users can create their visualization by organizing the frame layout and selecting components to load with a frame (Figure 7).

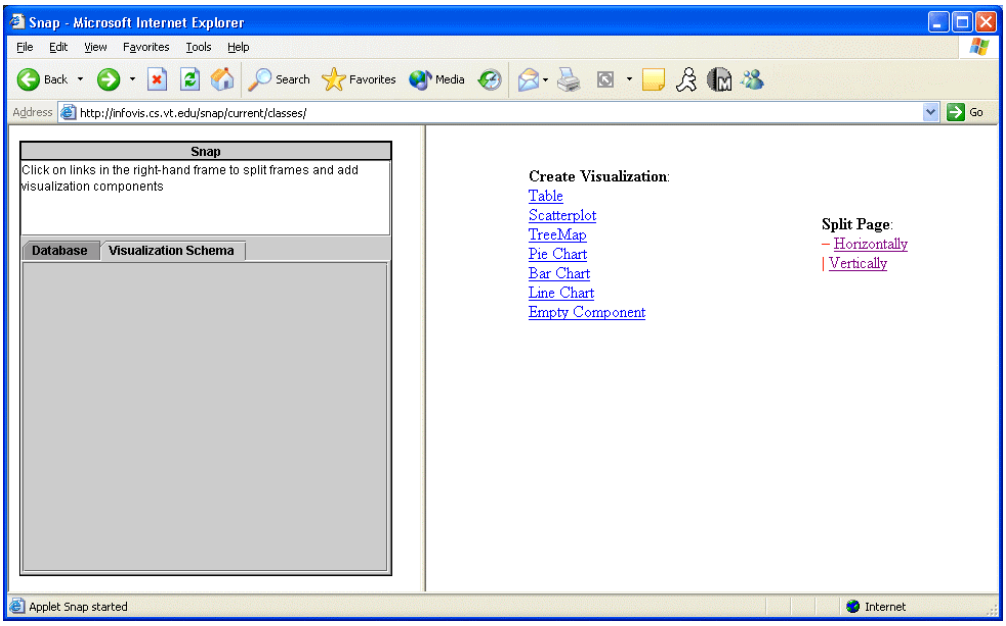


Figure 7 – Users can organize the frame layout and select components once Snap is connected to a database.

The user starts by constructing a simple data explorer. First they split frames in order to build tables for navigating through their data. Within one frame, they create a scatter plot and select the 'States' table to load into the plot (Figure 8). Within a second frame, they create a bar chart and select the 'Counties' table to load into the plot (Figure 9). Icons are shown in the left configuration panel representing each of the visualization components that have been loaded. Users can right-click on an icon to change the properties of visualization.

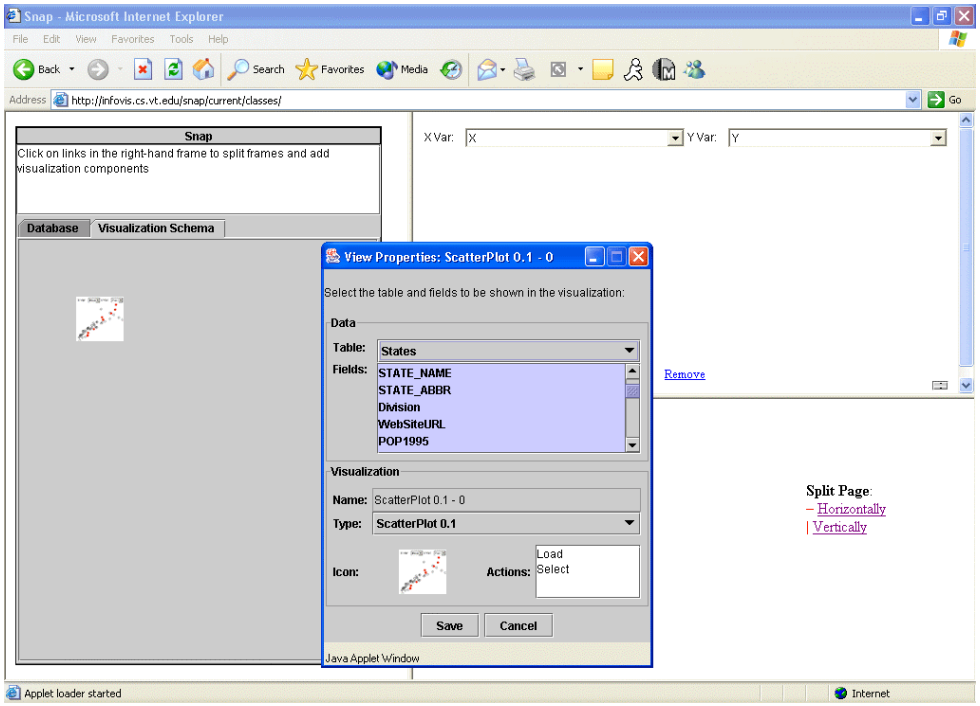


Figure 8 – A user specifies the data to be loaded into a scatter plot visualization component.

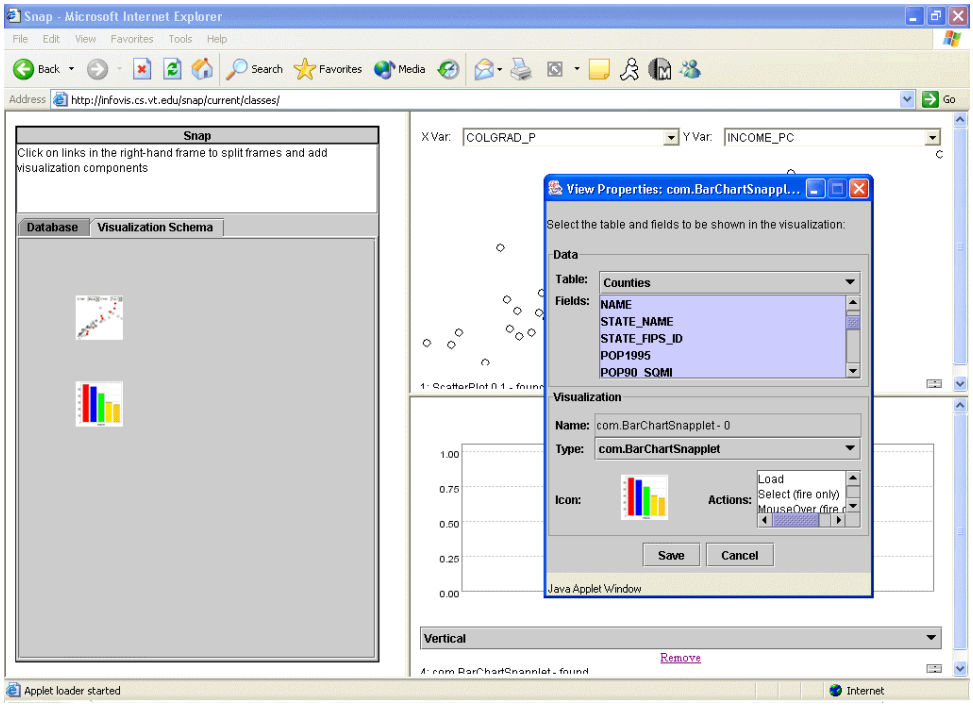


Figure 9 – A user specifies the different data to load into a bar chart component.

In this scenario, the user chooses to coordinate the plot with the bar chart. They right-click on the scatter plot icon and choose 'Add Coordination' (Figure 10). The use then clicks on the bar chart icon to construct a link between the two components. In the coordination properties dialog the user specifies the actions to tightly couple between the two components (Figure 10). This dialog indicates the two components that are being coordinated, the data encapsulated by each view, and the relationship between the views. They choose to couple the plot's 'Select' action with the 'Load' action of the bar chart. This enables the two views to become coordinated so that when states are selected in the scatter plot the corresponding counties are loaded into the bar chart (Figure 11).

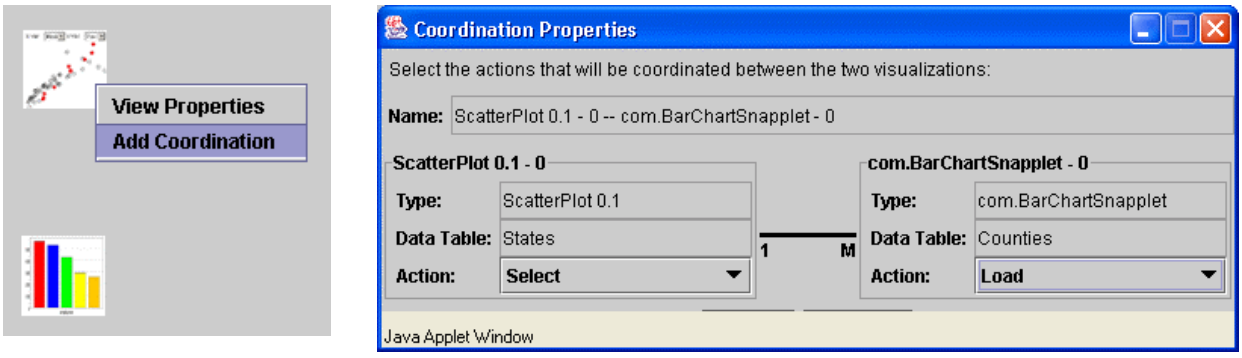


Figure 10 – Users can add a coordination between two components and specify the actions for tight coupling.

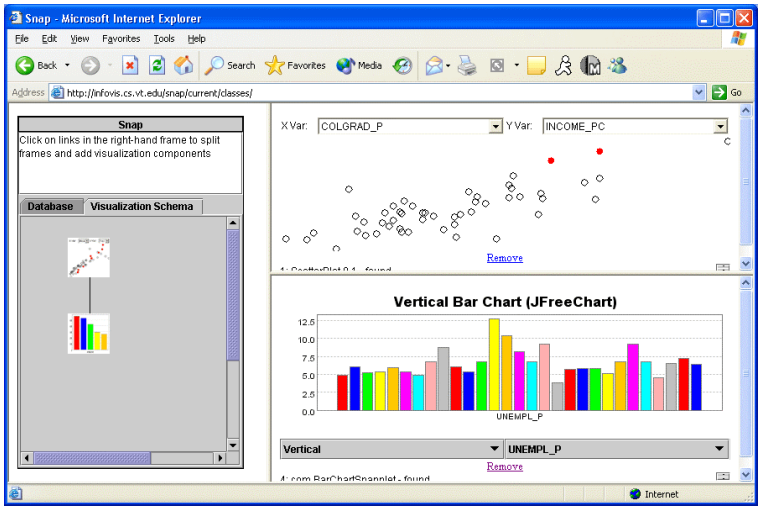


Figure 11 – Snap coordinated visualization with tightly coupled actions between a scatter plot and bar chart.

1.4 Research Questions

As previously noted, there is a need for flexibility in the design and implementation of information visualizations. In improving upon initial Snap research, there are several important questions.

First, the concept of coordination is evolving. The categorization and initial theory of coordination have been introduced. However, there are many limitations as previously mentioned. How can we improve the formalized modeling of visualization coordination to deal these limitations?

Second, how can we enable extensibility and improve flexibility in the construction of multiple-view visualizations? What are the necessary pieces to the architectural solution?

Third, the construction of a multiple-view visualization requires the integration of many visualization components. This requires a large and often inconsistent installation base. How can we improve the accessibility and distribution of multiple-view visualizations?

Finally, it is often difficult to integrate diverse data into a single conceptual picture. Users regularly have to massage and integrate data prior to initial visualization. How can we enable and automate the integration of diverse data?

1.5 Content

This research introduces the Snap *visualization server* and *system architecture* [NCI02] that will support visualization designers' need for flexibility. The Snap system architecture is a combination of several architectural styles. The communication between Snap and the

individual visualization components is an event-based, implicit invocation architecture while Snap itself is built as a layered system [SG96]. The layered architecture for Snap is shown in Figure 12.

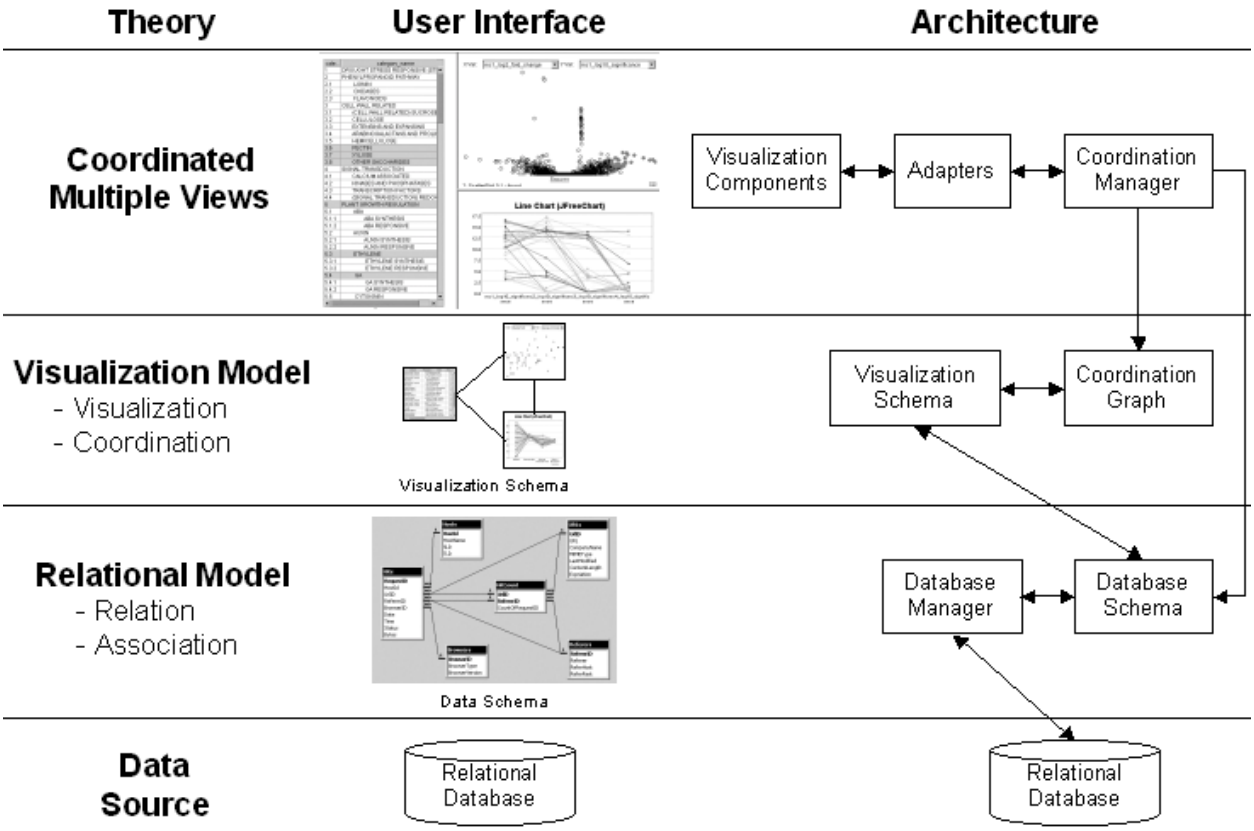


Figure 12 – Web-based Snap Architecture indicating the separate layers needed to implement the Snap system.

The Snap architecture supports an enhanced *visualization model* for coordinating visualization components (Chapter 3). This model allows for the tight coupling of multiple-tuple actions and provides support for event translation using underlying data associations. The new model reduces the coupling of visualization components, allowing a developer to focus on their visualization as a single cohesive component.

The layered architecture separates Snap into a Database Schema (Chapter 4), Visualization Schema (Chapter 5), and Coordination Manager (Chapter 6) with each layer providing additional functionality built on the capabilities of the layer below. The Database Schema supports the automatic retrieval of relations and their associations (schema) from a database. This layer is an abstraction of the database, providing support for querying the database and performing joins based on the database schema. The Visualization Schema provides an overview of visualization coordinations and a user interface for constructing and coordinating the visualizations. This user interface enables a user to specify the coordinations, while the schema is stored within the Coordination Graph. The Coordination Manager supports the coordination of events between the visualization components.

The architecture introduces Adapters to provide a standard interface for the communication between the Coordination Manager and the individual visualization components (Chapter 7). This allows for various technology specific components to communicate with Snap, enabling the support for run-time extensibility. Appendix B demonstrates the capabilities and usefulness of the Snap visualization server.

Chapter 2 Related Work

Visualizations play an important role in understanding a problem domain. They provide the capability for information exploration and cognitive amplification by utilizing human capabilities for visual pre-attentive processing. The primary use for visualization tools is to provide clarification and increased understanding of complex concepts [CMS99]. As visualization tools are more commonly used in scientific and engineering environments, users require powerful visualizations that can provide a deeper understanding of the complex information space. Visualization designers utilize multiple views to provide the needed capabilities. However, there is an increasing need to provide visualization flexibility.

2.1 Multiple-View Visualizations

A multiple-view visualization utilizes two or more distinct views to present a single conceptual entity [BWK00]. Multiple views are able to display different aspects of the data and help to improve understanding through interaction. The HomeFinder tool was one of the first visualization tools to utilize multiple views (Figure 13). The tool utilized tight coupling between the windows to support Dynamic Query Filters and Details on Demand [AS94].

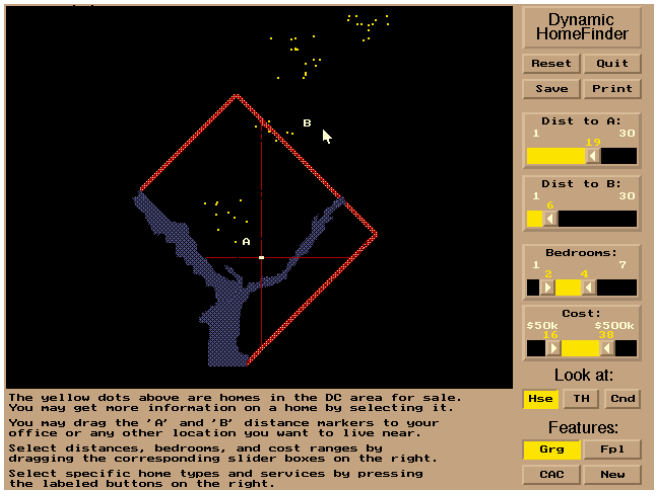
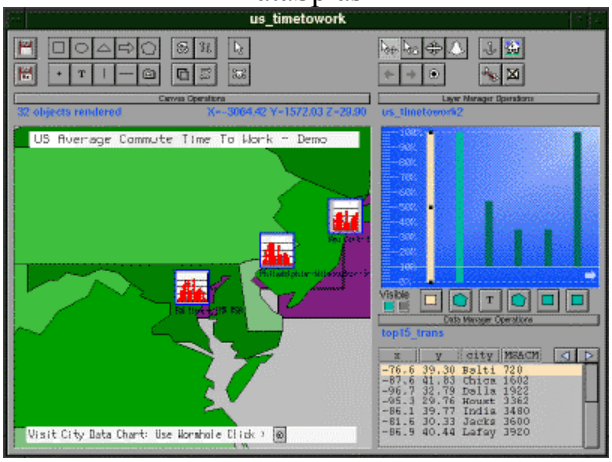


Figure 13 - HomeFinder tightly couples different views.

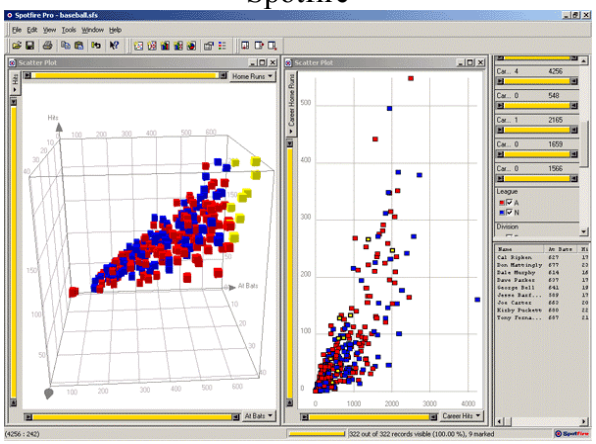
Ward introduced advanced techniques for brushing and linking, building on Becker and Cleveland's concept of data brushing [BC87]. Ward demonstrated the concept of an n-dimensional brush as a technique for interaction between various multiple view visualizations [War97]. Advanced multiple-view systems, such as Wing, have evolved and found use in many problem domains [MMB95]. Tools such as Sage/SageBrush [RCK97], DEVise [LRB97], DataSplash [ACS96], and Spotfire [AW95] have enabled users to construct powerful multiple-view visualizations of a single data relation. Sage uses an automated approach while DEVise, DataSplash, and Spotfire use a form-based dialog to match attributes to visual properties.

Wing

DataSplash



Spotfire



Visage

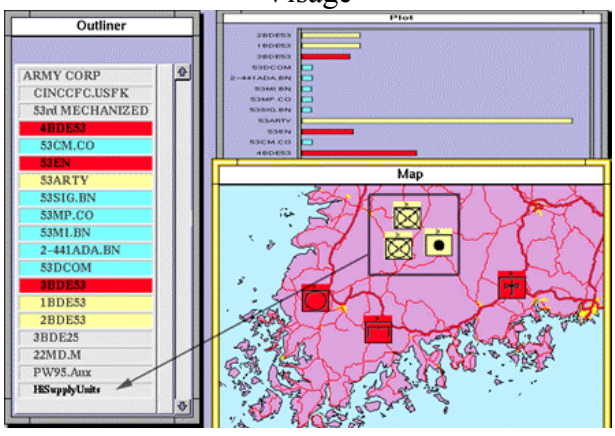


Figure 14 - Wing, DataSplash, Spotfire, and Visage are examples of how multiple-view visualizations support many problem domains.

These systems enable multiple visualizations of a single relation. SageBrush provides a visual language and user interface for connecting views. DEVise and DataSplash enable users to link the visualizations for synchronized pan and zoom. Visage/VQE [DRK97] extends the concepts of attribute mapping, brushing, and dynamic queries to data composed of multiple relations. Users can perform the operations on tuples and attributes in different relations that are associated by a join. DataSplash supports a semantic zooming space that lets users drill down across relations by zooming in.

Visualization spreadsheets emphasize the data operands by displaying them in cells (Figure 15). The coordination is based on data relatedness and operations. It is defined by a command language or by user interface direct manipulation [CBR97]. While visualization spreadsheets provide user definable data manipulation and coordination, they only provide limited support for constructing visualizations.

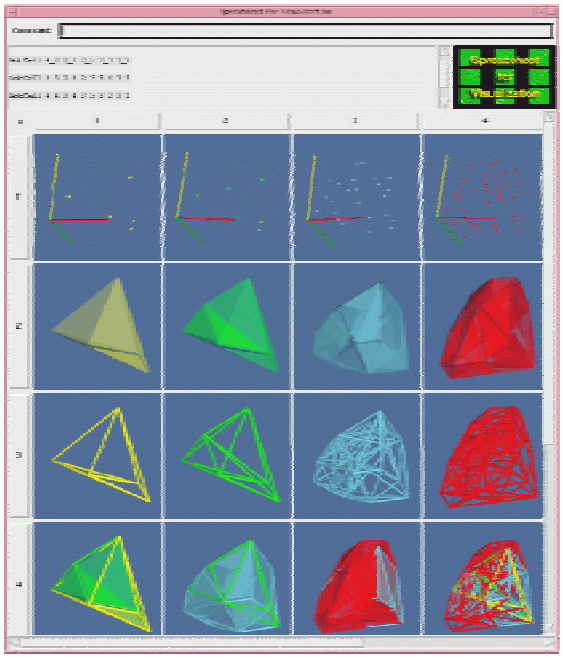


Figure 15 - Visualization Spreadsheets provide coordination based on data operations.

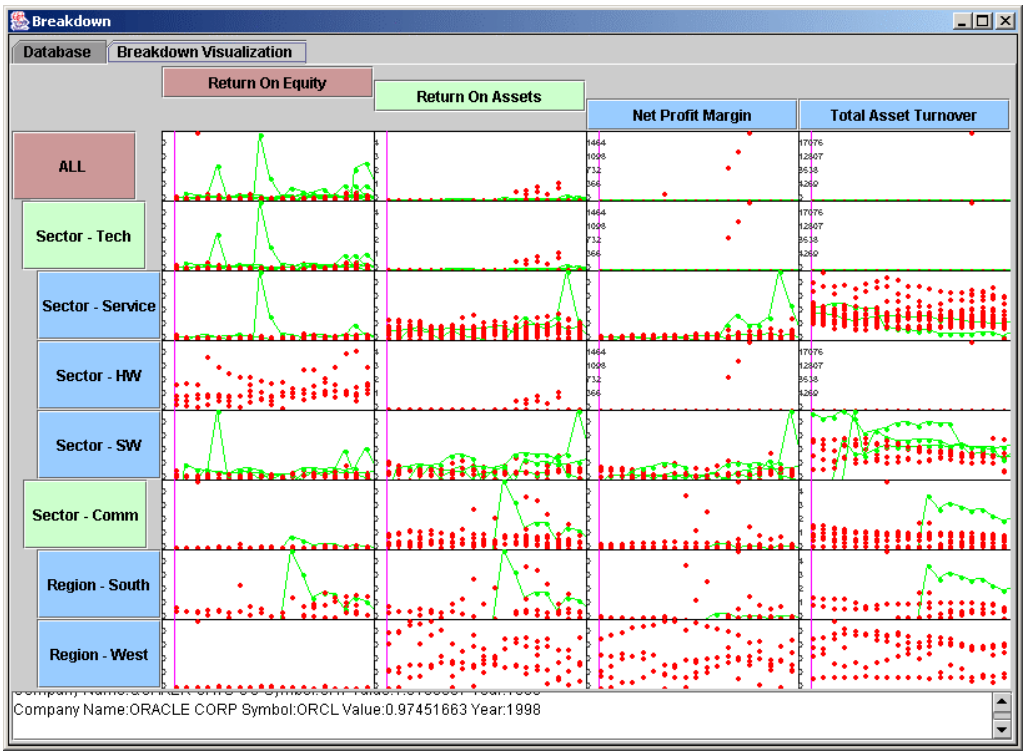


Figure 16 - Breakdown Visualization coordinates multiple views to support drill-down.

Conklin, Prabhakar, and North have demonstrated the need to add related views as a user drills down into their data set [CPN02]. Breakdown Visualization supports coordinated views and drill-down, enabling users to make comparisons of aggregated overviews and detailed subsets [PCN02]. Breakdown Visualization, like many customized multiple-view visualizations, requires specifically organized data and does not have the capability to visualize generalized data sets (Figure 16).

A taxonomy for coupling in visualization illustrates the types of coordinations supported in multiple-view visualizations [Nor01]. The taxonomy includes several tasks and each action may translate to one or more user interface actions. The task of selection includes the following user interface actions: single select, deselect, multiple select, or mouse-over. The task of

navigation includes the following user interface actions: scroll to an item, zoom to a set of items, load a data set, or move focus to an item (such as in Fisheye). This taxonomy includes the following visualization coordinations:

- Select ↔ Select (Brushing and Linking)
- Select ↔ Navigate (Overview + Details, Drill-Down, or Details on Demand)
- Navigate ↔ Navigate (Synchronized Navigation)

Guidelines have been developed for the design of multiple view systems. These guidelines describe when and how to use multiple views. The rules recognize the impact on user learning time, user memory, and machine computation [BWK00]. As the guidelines and use of multiple-view visualizations have developed, the tools have moved away from statically defined coordination towards more dynamic coordination. Developers have recognized the power of building coordinated multiple-view visualizations and have provided the ability to define the coordination.

2.2 User Constructed Visualizations

The dataflow model provides an effective approach for construction and coordinating visualizations. The Application Visualization System (AVS) is one of the first multiple-view visualization systems built on the dataflow model (Figure 17). Within the dataflow model, modules are combined by connecting the output of one module to the input of another (Figure 18). The output ports produce specific data types and the input ports accept only certain data types. The information flows downstream and each module either manipulates and/or displays the data [UFK89].

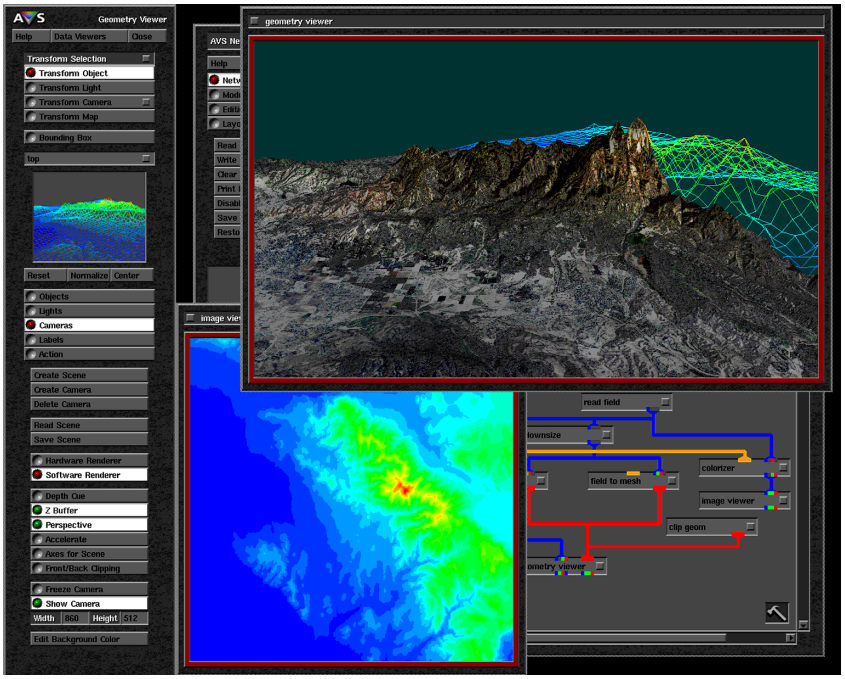


Figure 17 - AVS is a multiple-view visualization system that utilizes the dataflow model.

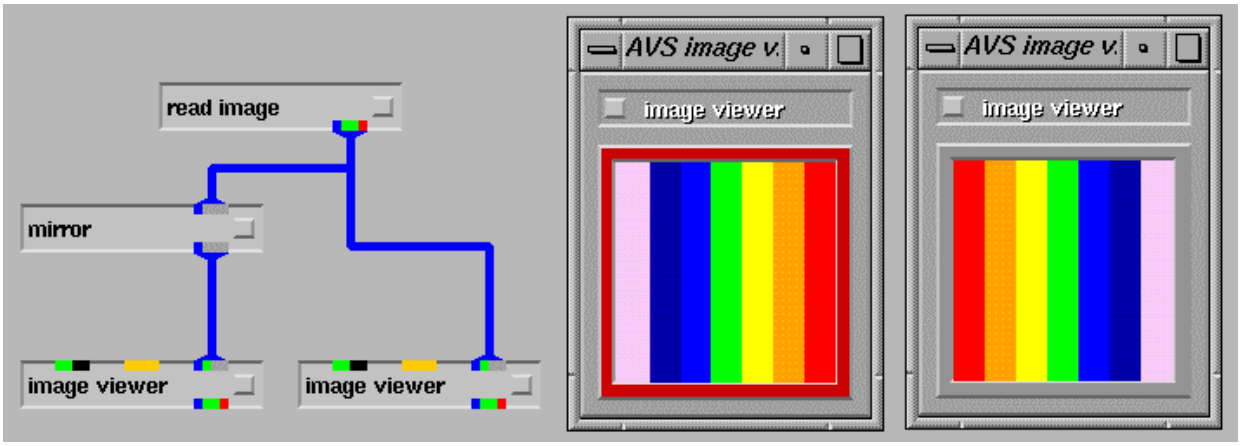


Figure 18 – A dataflow network with computational components as nodes that contain input and/or output ports. Links are used to connect these ports and data flows downstream (top-to-bottom).

Dataflow systems similar to AVS include Khoros [KR93], GeoVista [TG02], apE [D90], OpenDX (IBM's Visualization Data Explorer) [O02], Tioga [ACS96], and others.

North and Shneiderman introduce Snap-Together Visualization, a user interface for coordinating visualizations (Figure 19). As shown in Figure 3, it builds on the relational model of the underlying data [NS00a], [Nor00], [NS01]. In this model, each visualization wraps a relation and each coordination is built on relational joins. Each tuple is usually depicted as a single item in the visualization. The relational model supports many common coordinations: brushing and linking, overview and detail, drill-down, synchronized scrolling, and details on demand.

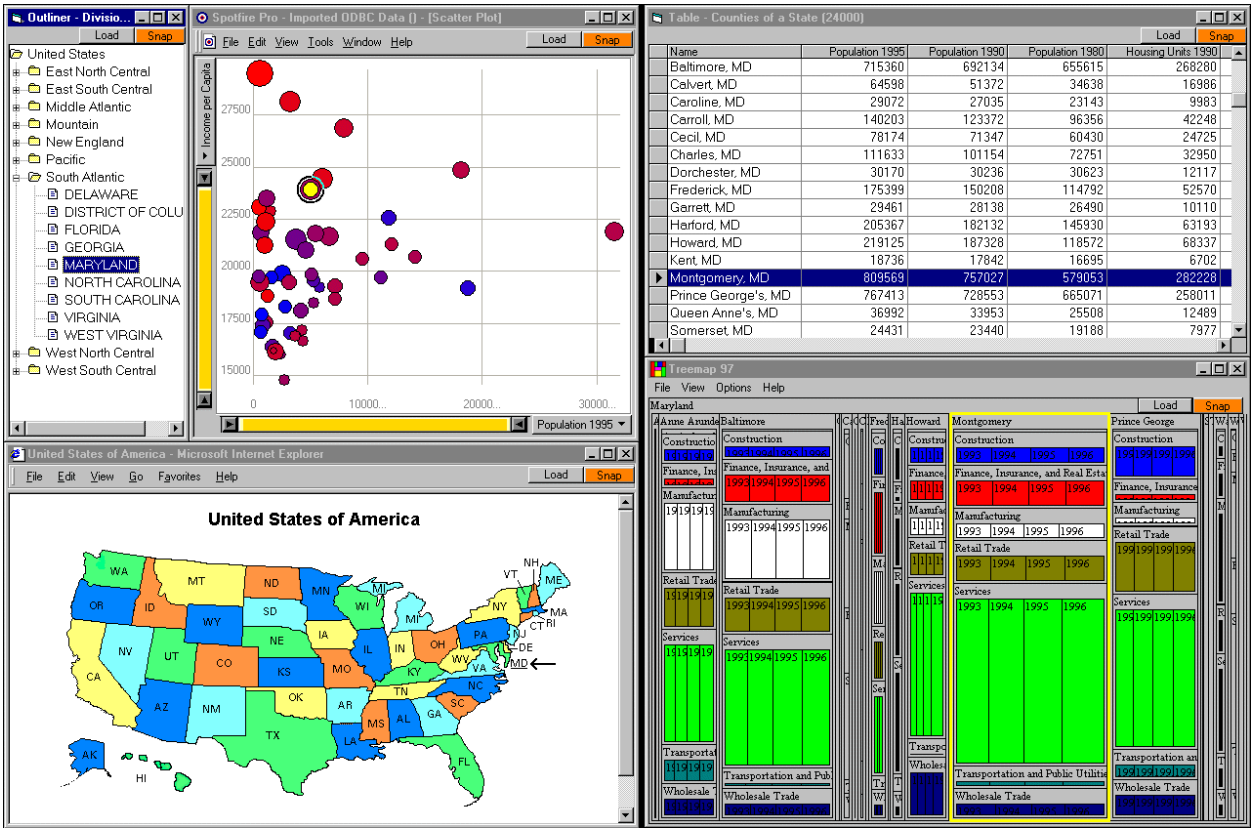


Figure 19 - Snap-Together Visualization allows coordination based on relational joins.

North and Shneiderman identified performance benefits but cognitive and usability issues with relational coordination concepts [NS00b]. They found that users were able to construct coordinated multiple-view visualizations based on relational coordination.

Tools such as APT [Mac86], Sage/SageBrush [RCK97], DEVise [LRB97], Spotfire [AW95], and others enable users to construct visualizations of a single relation by mapping data tuples to visual marks, and then mapping tuple attributes to visual properties of the marks. APT and Sage use an automated approach while DEVise and Spotfire use form-based dialog boxes. The systems enable the simultaneous display of multiple such visualizations of the relation. However, the coordinations between these views are hard-coded by the environment.

2.3 Visualization Architectures

Shaw and Garlan introduce several architectural styles. An architectural style defines a family of systems based on a pattern of structural organization [SG96]. Visualization design typically falls within one of a few architectural styles that include object oriented organization, layered systems, pipes and filters, and event based, implicit invocation. Within these architectures there are more specific models or patterns [GHJ95] which further describe the architectural style.

Bergin et. al. have identified the model-view-controller (MVC) [KP88] and dataflow paradigms as common paradigms for visualization design [BBG96]. However, they did not specifically look into software architectures for coordination. Radestock identifies coordination types available for software architectures [Rad99]. Many design patterns are integrated in the design and implementation of visualizations. Hayden et. al. present a catalog of coordination patterns [HCY99].

Typical visualization design is modeled by the mediator and observer design patterns. The mediator encapsulates the control and coordination of interactions at a single location. This prevents agents from explicitly referring to each other. The observer pattern defines a one-to-many dependency so that when an object's state changes, the dependents are updated [GHJ95].

Included below is an overview of the architectural styles that are used in multiple view visualizations.

Table 1 - Coordinated Visualization Architectures and System Examples

Architectural Style	System Examples
Data Abstraction and Object Oriented	Rivet, Wing
Layered Architectures	J2EE, JMS
Pipes and Filters	AVS, Sieve, Dataflow
Event-Based Implicit Invocation	Snap-Together Visualization

2.3.1 Object Oriented Organization

Architectures based on data abstraction and object-oriented organization are very common for individual visualization components and visualization toolkits. Objects encapsulate the data and associated operations [SG96]. Example visualizations and toolkits using this architecture include TreeMaps [Shn92], Jazz [BMG00], Moosburg [CRI01], Rivet [BST00], VTK [SML97], along with several subsystems of Snap. The Rivet toolkit requires programmed coordination; however, it allows coordination using component listeners [BST00]. Jazz, Moosburg, and Rivet are examples of mixed architectures.

The primary disadvantage to object oriented organization is that an object must know the identity of other objects in order to interact. Methods are explicitly invoked and each object must remain updated when object identities change.

2.3.2 Layered Systems

In a layered system, each layer provides services to the layer above and serves as a client to the layer below. Common layered systems include the ISO model, J2EE (Figure 20), and common 3-Tier software architectures. Clients utilize lower layer services for coordination.

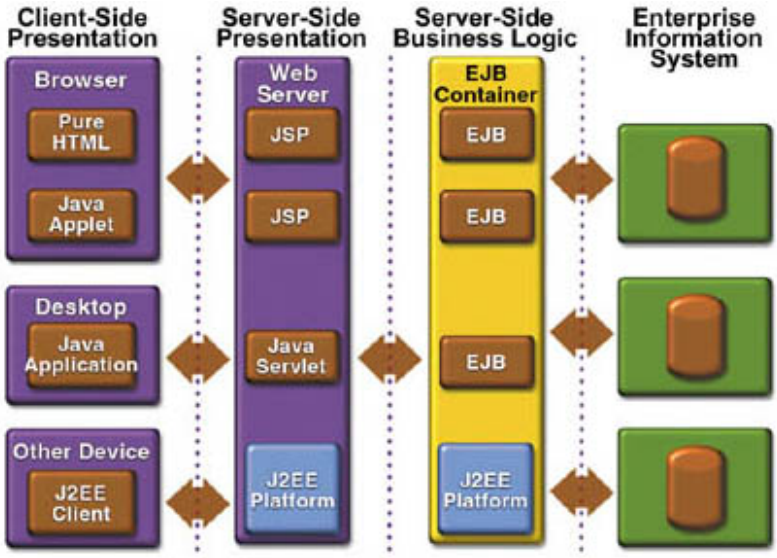


Figure 20 - J2EE Application Model is an example of a layered system as is Snap's architecture shown in Figure 12.

The Snap system uses the layered architecture (Figure 12) within the coordinating component, but does not use this architecture to coordinate visualizations. It is often difficult to organize a system into the layered architecture, and it may result in closer coupling than needed [SG96]. Web-based visualizations are evolving beyond typical n-tier user interface architectures because the client is more than a dumb interface. Web-based visualization clients may include several layers of logic, forcing server-side logic to move to the client. This is the case with the Snap architecture.

2.3.3 Pipes and Filters

The pipe-and-filters architectural style models each component with a set of inputs and outputs. These components are organized into a network, where filters are computational components and pipes are the connections between these components [SG96]. The dataflow model is an example of the pipe-and-filters architecture. Examples include previously

mentioned dataflow systems. A similar system, Sieve, is an example of a collaborative dataflow visualization (Figure 21). However, a Sieve user cannot simply connect visualization components based on data types. Instead, the connections are programmed components [IBH97].

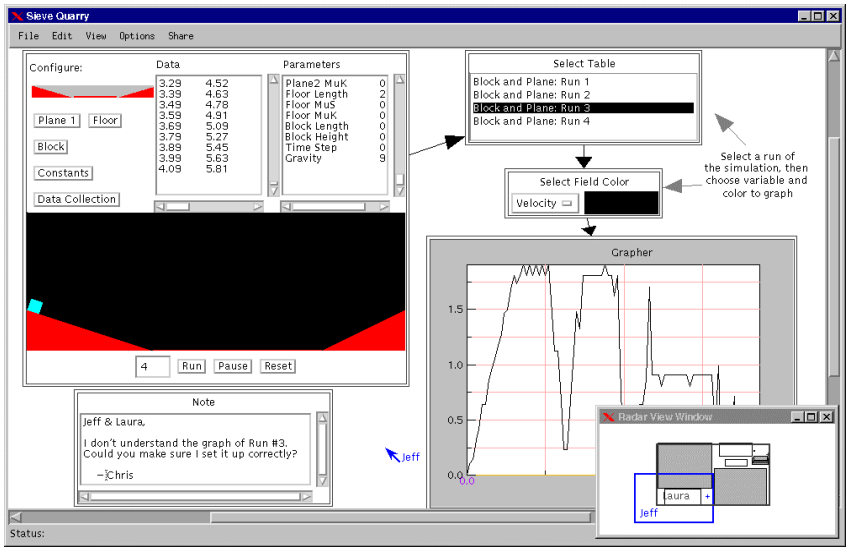


Figure 21 - Sieve's dataflow model requires programmed connections.

Dataflow coordination requires a strong understanding of data types for coordination. Programming the coordination is sufficient for developers but does not support a common user. Another limitation to pipe-and-filter is that a coordination cannot be made bi-directional. This limits the interactions available to the visualization. The Snap model is able to mimic the pipe-and-filter by coordinating components using a sequence of Select → Load coordinations (VisA.Select → VisB.Load, VisB.Select → VisC.Load). However, this provides limited functionality and does not strictly implement the pipe-and-filter architecture.

2.3.4 Event-Based, Implicit Invocation

In this architecture, components do not directly communicate with each other. Instead, a component will announce its events. Other components are registered to an event – associating a procedure with it. Once an event is announced, the system invokes all registered procedures. This procedure invocation occurs "implicitly" as events are announced [SG96]. Examples of this architecture include the Model-View-Controller (MVC) architecture [KP88], various proposed MVC enhancements, and the mediator design pattern [GHJ95]. Similarly, Pattison and Phillips present an architecture that proposes coordination of specification, model, and presentation layers [PP01]. Figure 22 demonstrates the application of MVC to multiple-views.

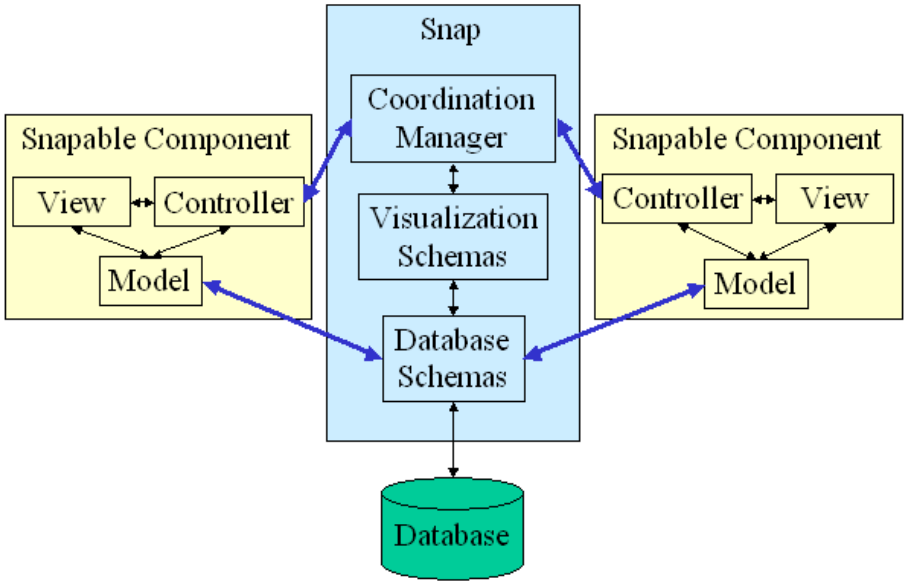


Figure 22 – Snap Visualization Model coordinates models and controllers from multiple views.

The software architecture and design of Snap-Together Visualization is best described as an event-based, implicit invocation architecture. This architecture enables the system to support the goals of flexibility, run-time extensibility, and easy integration of visualization components. The web-based architecture of Snap provides enhanced ability to visualize and explore diverse data.

Chapter 3 Enhanced Visualization Model

3.1 Motivation

Many of the new goals for Snap required an evolution of the Snap visualization model to deal with the limitations of the previous model. An enhanced visualization model was needed to support flexibility in visualization design, de-coupling of visualization components, and provide support for a wider range of actions.

The initial Snap model utilized parameterized queries, along with the concept of primary key actions and foreign key actions. The designer would have to specify the relationship to correctly connect two visualization components. This specification was done by constructing parameterized queries or configuring the association in the coordination properties (Figure 5). While this model allowed for conflict free coordination [Nor00], it limited the types of coordinations that were supported. The model also coupled visualization components, forcing component developers to differentiate between primary key and foreign key actions. The new Snap visualization model should simplify the requirements for coordinating visualizations.

The new visualization model should also provide support for multiple-tuple actions. Components should be able to fire and receive events specifying that an action occurred on several tuples. For instance, a visualization should be able to fire an event indicating that several items were selected or an event indicating that the user zoomed in on a group of items.

An enhanced visualization model was developed to achieve these goals [NCI02]. In the new model, components fire and receive events based on the primary key for the relation that they are encapsulating. If several items have been selected, an event is fired containing the primary key values (IDs) of the selected tuples. The concept of foreign key actions has been removed from the model. Snap is now responsible for translating events as they are propagated

to the individual views. This reduces coupling and simplifies the requirements for a component developer.

3.2 Schema Primitives

The Snap model rigorously defines multiple-view visualization in terms of the relational data model. Like the relational data model, it represents a balance between theory and practice. That is, it is intended to capture not only theoretical design of multiple-view visualization, but also common design practices of typical multiple-view visualizations. This approach enables additional goals of extensibility and applicability to existing components [NCS02].

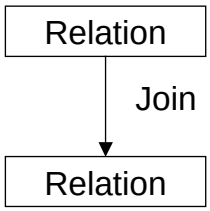
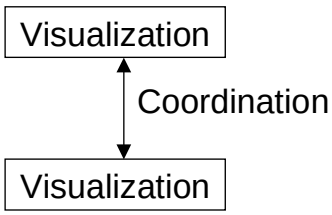
	Relational Databases	**Snap Visualization**
Design goal	Data design	Visualization design
Design method	Data schema	Visualization schema
Designer	Data owner	Data owner
Design change	Rapid, dynamic	Rapid, dynamic
Adaptability	Flexible	Flexible
Perspectives:		
Theory	Relational data model	Snap visualization model
User interface	Relational data schema	Snap visualization schema
Architecture	Relational DBMS	Snap visualization server
Schema Primitives:		
Theory	Relation	Visualization component
	Tuple	Visual item
	Attribute	Visual property
	Selection	User interaction
	Join	Coordination
User interface		

Table 2 – A strong analogy between relational database concepts and Snap visualization concepts enables a matching level of design capability.

The Snap model establishes a direct correspondence between relational data schema primitives and visualization schema primitives: (see also Table 2)

- *Visualization component* = data relation. A visualization component is a view that displays a data relation or query result (we assume that queries or database "views" are integrated into the data schema like relations). An example is a scatter plot component that displays a binary relation (or a binary projection of a relation with larger arity). A visualization component can implement a specific visualization type (e.g. scatter plot), or use automated techniques to dynamically generate visualizations (e.g. APT [Mac86]).

- *Visual item* = data tuple. Data tuples are displayed as visual items in a visualization component. For example, a tuple is displayed as a dot in the scatter plot.

- *Visual property* = data attribute. Data attributes are used by visualization components to compute graphics. Users map data attributes to component-specific visual properties. For example, a data attribute is mapped to the x axis on the scatter plot, causing tuples to be visually arranged according to their value for that attribute.

- *User interaction* = tuple subset selection. A user interaction in a visualization component selects a subset of tuples from the displayed relation, analogous to performing a selection query, and typically alters the visual display of those tuples. For example, a user highlights a set of outlier tuples in the scatter plot by directly selecting them, or zooms onto a single tuple to reveal more details. Interactions are defined by each component and identified only by name (e.g. "select", or "zoom"). Each interaction defined by a component has a corresponding tuple subset that it controls. A subset consists of zero or more tuples from the relation (an empty subset indicates no tuples were selected). Tuples in a subset can be

identified by their unique primary-key values. Every component also has an inherent "load" action, which contains the entire relation currently loaded and displayed in the component.

- *Visualization coordination* = data join. A coordination links an interaction in one component to an interaction in another component, by equating the corresponding subset selections according to a join between the components' relations. User actions on tuples in one component cause visual actions on join associated tuples in the other component. The tuple subset of the user action in the former component is inner-joined to the latter component's relation, resulting in the new tuple subset to use for the action in the latter component. For example, brushing-and-linking between two scatter plots of the same relation is a coordination of the "select" actions across the implicit one-to-one join association. Then, when users highlight items in one plot, the associated items are automatically highlighted in the other plot. A coordination can link any pair of actions between components. Coordinations, like joins, are bidirectional.

3.3 Coordinations and Joins

An important advancement in this model is the generalization of coordinations and interactions [NCI02]. A coordination is generalized to any single or compound join association, including one-to-one, one-to-many, and many-to-many associations. Join associations for coordinations are automatically derived and executed from data schemas, eliminating the need for user-defined parameterized queries for joins. Furthermore, interactions are generalized to tuple subset selections, enabling them to act on single or multiple tuples. Chained coordinations cascade across arbitrary associations, beyond the previously limited one-to-one cascading.

Four cases demonstrate how generalized coordinations correspond to various joins in the data schema. These are demonstrated using the example data schema and multiple-view visualization for website hits shown in Figure 23 and Figure 24.

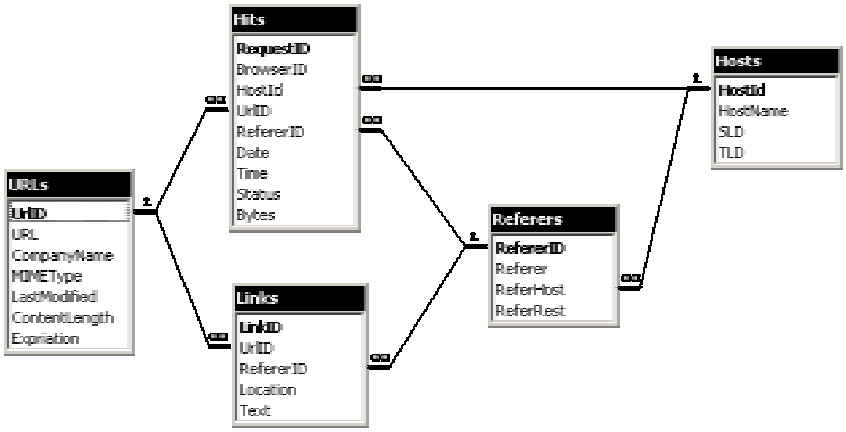


Figure 23 – An example data schema for a database of hits to our website. "URLs" stores information about pages on our website. "Referrers" stores information about external websites that have links to out website. "Hits" stores information about each hit, including a reference to the page requested in "URLs" and the external referring site in "Referrers".

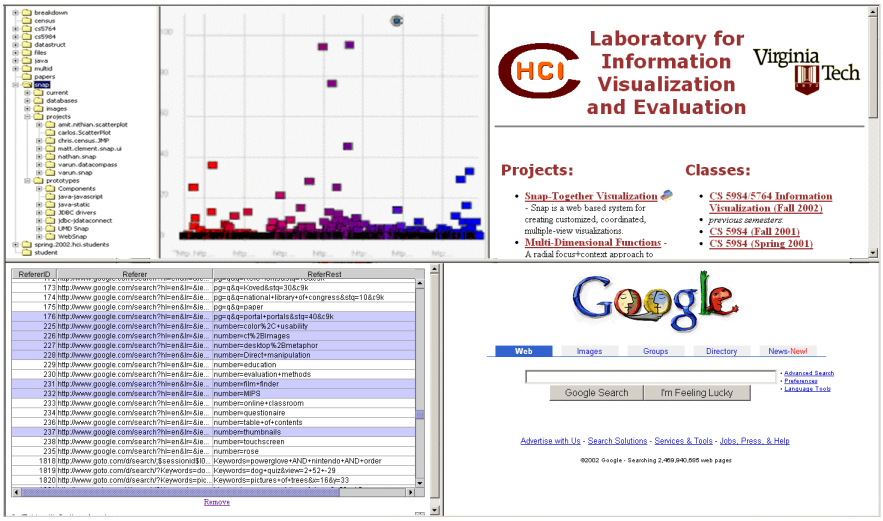


Figure 24 – An example multiple-view visualization constructed with Snap for the database shown in Figure 23. The website map generated from the URLs is shown in the TreeView (top left). Selecting a page in the map displays the page in the web browser (top right), and displays the distribution of hits to that page in the scatter plot (top center). Selecting pages also highlights referring sites listed in the table view (bottom left). Likewise, selecting referring sites highlights pages linked to, and shows their hits in the plot. Clicking a referrer shows its page in the other web browser (bottom right).

- *Self join*: A coordination can be established between two visualization components that display the same relation. In this case, the coordination corresponds to the implicit one-to-one join association that exists between the relation and itself.

  For example, the TreeView visualization component displays the URLs relation, using the URL page pathname attribute to display the tuples as a website tree structure. A web browser component also displays the URLs relation, using the URL attribute to display actual pages represented by the tuples. The "select" action of the TreeView is coordinated to the "navigate" action of the web browser across the self join. When users select a tuple in the TreeView, there is no need to actually perform a join since it is a self join, so the same tuple

is used to navigate the web browser to the selected page.

```
TreeView ← URLs → Web browser
```

- *Single join*: A coordination can be established between two components whose relations have a direct join association in the data schema. The relational model provides two primitive types of direct join associations: one-to-one and one-to-many.

  For example, in addition to the TreeView of the URLs relation, the scatter plot displays the Hits relation, showing all hits to the website by date and time. The "select" action of the TreeView is coordinated to the "load" action of the plot, using the direct join association between the URLs and Hits relations. The data schema indicates this is a one-to-many association. Selecting tuples in the TreeView joins those tuples to the Hits relation to find all the hits to the selected pages. The resulting Hits subset is then loaded and displayed in the plot, essentially filtering out hits to any other pages. This enables users to drill down from pages to hits.

  ```
  TreeView ← URLs ↔ Hits → Scatter plot
  ```

- *Compound join*: A coordination can be established between two components whose relations have an indirect association via one or more intermediate relations in the data schema. This requires a compound join, concatenating each of the direct joins along the indirect association path. Compound joins enable more complex associations such as many-to-many.

  For example, in addition to the TreeView of the URLs relation, the TableView displays the Referrers relation, showing an alphabetical list of all the websites that link (refer) readers to the URLs website (alternatively, it might be interesting to show referrers geographically).

The "select" action of the TreeView is coordinated to the "select" action of the TableView, using the compound join from URLs to Hits to Referrers. The concatenation of the one-to-many and many-to-one joins creates a many-to-many join. Selecting tuples in the TreeView joins those tuples to the Hits relation and then to the Referrers relation to identify the websites that actually sent readers to the selected pages, and then selects them in the TableView. Since coordinations are bidirectional, the reverse interaction also occurs. This example illustrates brushing-and-linking across a many-to-many association.

`TreeView ← URLs ↔ Hits ↔ Referrers → TableView`

- *Multiple alternative joins*: A coordination between two components whose relations have multiple alternative join associations connecting them requires the selection of one of the join associations for use in the coordination. In the compound join example above, an alternative is the compound join through the Links relation. This alternative would have a different effect. Selecting pages in the TreeView would indicate all the websites in the TableView that have links to those pages, rather than the websites that actually referred readers and generated hits.

  `TreeView ← URLs ↔ Hits ↔ Referrers → TableView`

  `TreeView ← URLs ↔ Links ↔ Referrers → TableView`

3.4 Data-centric Coordination

The Snap multiple-view coordination model employs a data-centric approach by focusing on tuple-based coordinations. The motivation for this approach is that tuple-based coordination is necessary for multi-table multi-view visualization. While the model focuses on data visualization and not data editing operations, data edits are tuple-based and can easily be added to the coordination model using standard data change events. Major data modifications that alter

entire relations are better implemented directly in the data schema. Snap's coordination model would combine well with other data-centric visualization concepts such as Visage [RCK97].

The Snap model captures the common types of coordinations used for data navigation. See [Nor01] for a detailed taxonomy of coordinations achievable with this model. These common coordinations support scalability of visualization in each aspect of relational data:

- Scalability in number of tuples: Overview+detail strategies support very large numbers of tuples, especially when chained across several views [PCS95].
- Scalability in number of attributes: Many attributes can be partitioned into multiple simpler views, while brushing-and-linking strategies [BC87] enable correlation between them. Systems such as Visage [RCK97] and Spotfire [AW95] demonstrate the value of brushing.
- Scalability in number of relations and associations: Coordinated drill-down strategies enable drill down across one-to-many associations between relations in different views [FNP99].
- Scalability in number of different data types: Data containing multiple distinct types of data requires different types of views and coordinations to navigate between them. Examples include geographic information systems that combine maps and statistics [MWH99], or bioinformatics [KGM02], which can involve numerical data, images, tree structures, and networks.

3.5 Discussion

Other types of multiple-view coordinations are representation-centric and are not explicitly modeled here. They coordinate specifics of the representations of two visualization components, such as sharing a common color-mapping scheme or displaying the same arbitrary region of a visual space. In the example of displaying the same region (e.g. synchronized pan

and zoom of the axes of two plots), if the region is strictly determined by a set of tuples then the tuple-based technique will work. Otherwise, components must share parameters of the visual representation. DEVise [LRB97] has demonstrated an approach for handling a subset of representation-centric coordinations based on sharing attribute ranges of 1- and 2-dimensional spaces (its "visual" and "cursor" links).

In the general case, as in the example of sharing a common color scheme, a more detailed level of integration is needed. Pattison and Phillips propose a view coordination architecture that attempts to coordinate representation by coordinating at presentation, model, and specification layers [PP01]. However, a stricter formalization of the architecture is needed to integrate these concepts with Snap's data-centric coordination. Improvise [WL02] employs an MVC approach that treats such properties of the visual representation as sharable data. Components must share many complex data structures that complicate inter-component communication and must be specified by users. Hence, representation-centric coordinations currently trade off with ease of extensibility and usability.

Chapter 4 Database Schemas

4.1 Overview

Providing a web based system for database visualization motivates many of the goals for the Snap architecture. Users should be able to connect to local databases, server databases, and publicly available third-party databases. With the goal of data flexibility, users need a solution that supports database technology independence. They should be able to connect to a dataset regardless of whether it is stored in Oracle, MS SQL Server, or something as simple as a spreadsheet or flat file. These goals have driven the decisions behind the architecture and implementation of Snap's database connectivity. The bottom layer of the Snap architecture is designed to support these goals for technology and access independence (Figure 25).

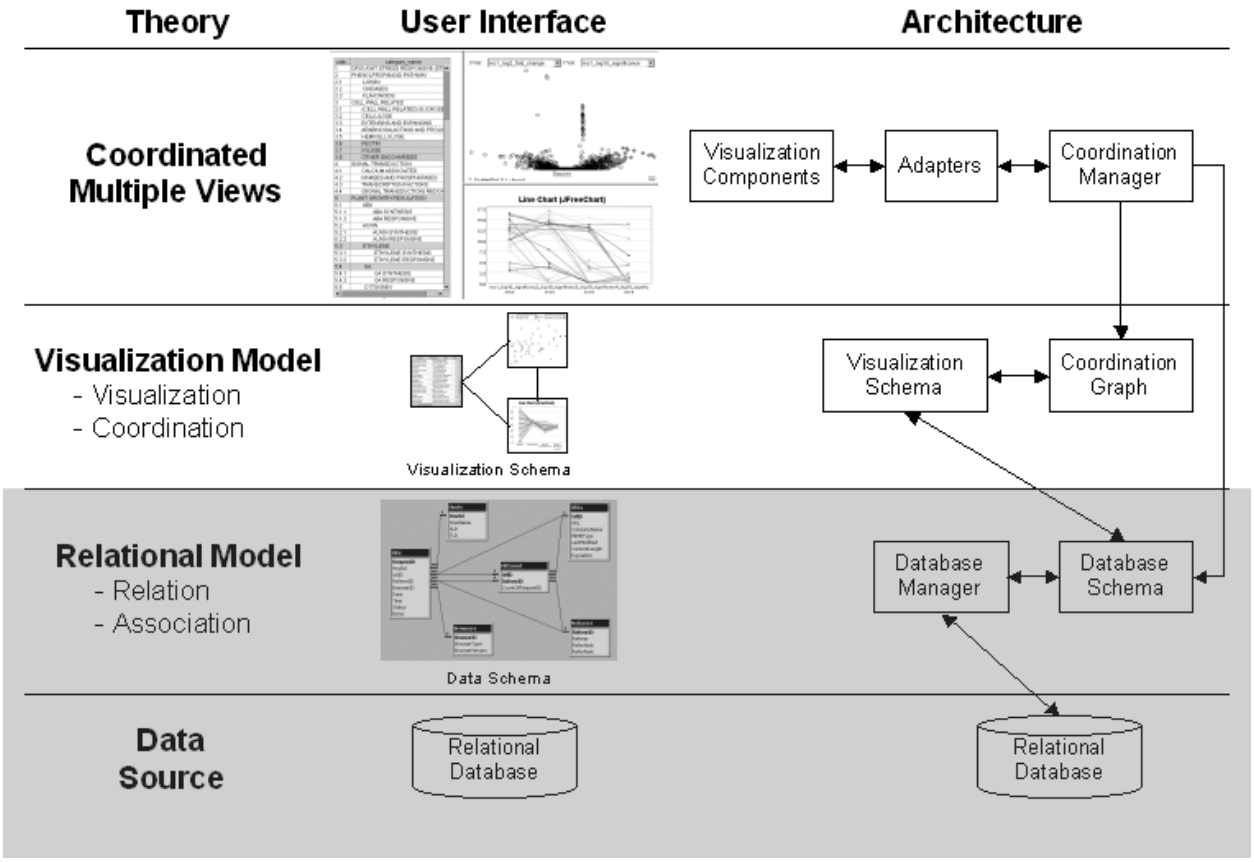


Figure 25 – Database Schemas provide an abstraction of the database.

The bottom layer of the Snap architecture supports the automatic retrieval of relations and their associations (schema) from a database. This layer is an abstraction of the database, providing support for querying the database and performing joins based on the database schema. It is composed of three subsystems: the relational database, the Database Manager, and the Database Schema. The relational database maintains the data and provides a flexible system for managing the data. The Database Manager manages the connections and queries to the database. The Database Schema maintains a data structure containing the tables and their associations. It is the subsystem responsible for providing services to higher layers.

4.2 Database Manager

The Database Manager is responsible for establishing database connections and executing queries. These connections may be to local databases or publicly available network databases. The Database Manager is responsible for managing concurrent database queries, allowing the Coordination Manager to be multi-threaded and providing quicker feedback to the user. The Database Manager utilizes JDBC and ODBC technologies to provide for access to many databases independent of their technology. Once a connection is established to a specific data source, JDBC provides a standard interface for querying the data. The Database Manager also provides the user interface for establishing connections (Figure 26).

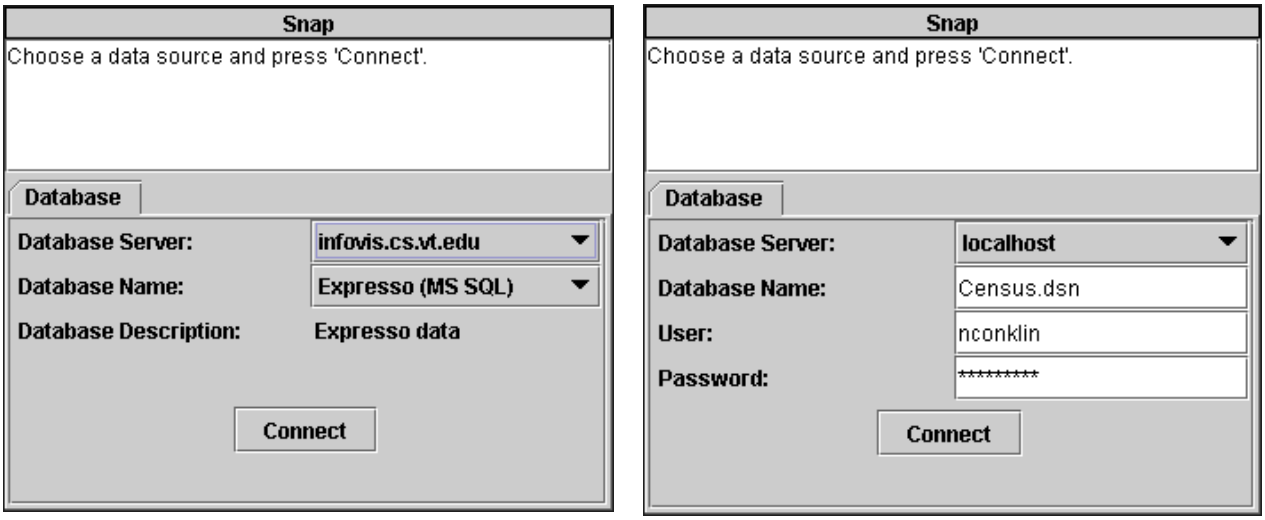


Figure 26 – Database Manager's interface for connecting to data sources.

4.2.1 Database Connectivity

Sun's JDBC (Java Database Connectivity) architecture provides an object-based framework for connectivity to tabular data sources [Sun02]. Snap utilizes JDBC to bridge outside of the Java Runtime Environment (JRE) into ODBC drivers. The JDBC architecture allows a component to use custom drivers to connect to specific databases through a standard interface.

Snap utilizes Sun's JDBC-ODBC bridge as a Type 1 driver to establish local ODBC connections. This is a common driver that is distributed with Sun's JRE [Sun02]. Snap also utilizes NetDirect's JDataConnect Type 3 driver for connecting to network databases. This driver has a small object base that is included with Snap. JDataConnect also requires a server-side component that allows ODBC databases to be served across the network [Net02].

ODBC drivers provide support for connecting to both local and remote databases. ODBC is a common interface for accessing a variety of database management systems (DBMS). Specific database systems are responsible for providing drivers that support the ODBC interface [M02]. By utilizing ODBC, Snap supports interoperability and reduces the need to massage data

before visualizing it. Figure 27 demonstrates the access provided with the use of JDBC and ODBC.

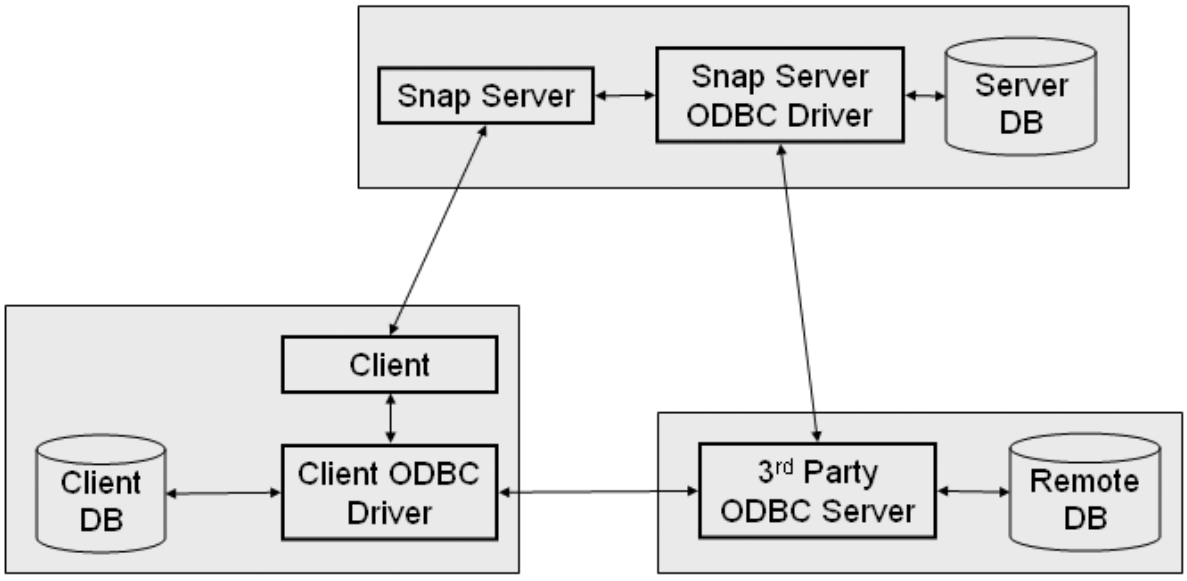


Figure 27 – Snap supports database connectivity to local and remote database access points.

4.2.2 Security and Platform Independence

Providing Snap with access to remote and local databases requires Snap to have privileges to get outside of the Java security sandbox [Sun02]. Because Snap interacts with other frames in the web browser, a Java Web Start deployment would limit Snap's capabilities. Solutions to this problem include signing the Snap code with a certificate issued from an authorized certificate authority or having the user provide Snap with additional security privileges through the use of Java's policytool.

ODBC connectivity does not strongly support cross-platform database connectivity. ODBC is a standard interface developed by Microsoft with strong support on Windows

platforms. However there are a few solutions. A few projects are working towards providing platform independent support for ODBC. These projects include iODBC [PIO02] and unixODBC [u02]. Another solution is to integrate other Type 2, 3, or 4 JDBC drivers with Snap [Sun02]. However, these drivers typically are limited to connecting to a single type of database such as Oracle, MySQL, or PostgreSQL.

4.2.3 Concurrent Queries and Connection Pooling

Event propagation must support rapid updating of the individual views. The database is used in event translation and the formation of queries for 'Load' events. To support rapid updating, Snap's coordination management is multi-threaded rather than serial. The Database Manager handles concurrent access to the database. This can be tricky because most JDBC drivers are not thread-safe. Because of this, the Database Manager must either synchronize database access or manage multiple connections to the database.

Snap manages multiple connections to a database. It utilizes the JDBC and ODBC driver support for connection pooling. These drivers provide both client-side and server-side connection pooling. However, each JDBC driver has varying levels of support. This accounts for performance differences when connecting to different databases.

4.3 Relational Database Schema Structure

Snap requires an understanding of how the data is related in order to coordinate events between two views that encapsulate different data. At a minimum, data-centered coordination requires an understanding of data relatedness. Snap has tightened this model by building on relational database concepts. Related data is described in the database by associations. The Database Schema retrieves these associations from the database and builds a data structure to model the data relatedness.

Once Snap has connected to a database, it automatically retrieves the schema structure. Snap uses this Database Schema to automatically perform coordinations. The initial implementation of Snap placed the burden on a user, requiring the user to specify the association between the views. The Database Schema subsystem is responsible for maintaining this data structure. It uses this structure to provide an interface to the Visualization Schema and Coordination Manager. The following interface is provided to higher layers:

4.3.1 Interface to the Structure

```
getTables()

getFields(String tableName)

getPrimaryKey(String tableName)

getAssociation(String srcTable, String destTable)
```

The Database Schema provides this interface into the structure of the database schema. These methods are used by the Visualization Schema to allow a user to configure the data that is encapsulated by the visualization. getTables() returns a list of relations (tables and views) available from the database. getFields() returns a list of column headings available for a relation. getPrimaryKey() returns the column identifier of the primary key of a relation. This identifies the column data used when firing and receiving events. getAssociation() returns the cardinality of the association between the source and destination relations.

4.3.2 Interface to the Data

```
getResultSet(Query sql)

translateEvent(String srcTable, String destTable,
     Vector inputKeys)
```

The Database Schema provides this interface for querying the database. The Coordination Manager uses these methods when loading data into a component or translating events between coordinated visualizations. getResultSet() supports database queries. translateEvent() utilizes the database join to map input keys from the source relation to output keys for the destination. This is described further in the event translation section.

4.4 Event Translation

When coordinating events between two components that encapsulate different relations, event translation is needed to join the relations [NCI02]. In the scenario described in Figure 23 and Figure 24, events occurring in the tree-view visualization of URLs must be translated when coordinated to the components displaying Hits and Referrers. When the tree-view of URLs fires a "Select" action event, it sends Snap a list of the URL IDs of the URL tuples selected by the user. Snap then propagates the event to the scatter plot of Hits according to the coordination in the Visualization Schema. When firing the event to the plot, Snap must first translate the URL IDs to the associated Hit IDs by performing a data join. The plot then receives the translated event, and highlights the appropriate Hit tuples in the display. Similarly, event translation is needed when coordinating visualizations of the URL and Referrer relations.

The Coordination Manager utilizes the Visualization Schema to propagate events and determine the relations encapsulated by each component. It then utilizes the Database Schema to translate events appropriately based on the underlying data join associations. The Database

Schema translates events across single and compound joins. The events are translated by constructing a join query between the two relations. In the example of Figure 24, selecting multiple URLs in the tree-view requires Snap to translate the event for coordinated components. If three URL tuples are selected with UrlIDs of 4, 5, and 6, then the Database Schema constructs the following query to translate the URLs into Hits for the scatter plot:

```
SELECT Hits.RequestID
FROM Hits INNER JOIN URLs ON Hits.UrlID = URLs.UrlID
WHERE URLs.UrlID IN (4,5,6);
```

For coordinating across intermediate relations, event translation requires a compound join query. For the table-view of Referrers, the following query retrieves the Referrer tuples associated with the selected URL tuples:

```
SELECT Referers.RefererID
FROM Referers INNER JOIN
    (Hits INNER JOIN URLs ON Hits.UrlID = URLs.UrlID)
ON Referers.RefererID = Hits.RefererID
WHERE URLs.UrlID IN (4,5,6);
```

4.5 Limitations

Database queries have a performance impact on the feedback of the multiple-view visualization. This is especially true when Snap is connected to remote data sources. These queries affect "Load" actions and event translation. Result caching and preloading of IDs could provide performance improvements. However, the architecture does not have a performance impact when each view encapsulates the same data. In these cases, event translation is not

needed. This is typically the architecture used in dynamic query systems such as HomeFinder [AS94] and TimeSearcher [HS02].

Snap also provides little feedback as events are propagated to other views. As the Database Schema performs event translation and "Load" actions, visual feedback of progress could be provided to the user. However, the current implementation does not support this feedback. These limitations may be addressed in further extensions of the Database Schema layer.

4.6 Extensions

The Snap architecture provides a foundation for several extensions to the Database Schemas layer:

4.6.1 Additional JDBC Drivers

Additional JDBC drivers may be provide by the Database Manager. These drivers could support common Unix databases that are not supported through ODBC. The Database Manager would need to provide user interface support for connecting with various drivers.

4.6.2 Support Arbitrary Database Joins

The current Database Schema interface and implementation only supports event translation for basic join associations between two relations. Support is needed for arbitrary and compound database joins as described in the event translation section. This support would need to include the following additions to the interface:

```
getJoinPaths(String srcTable, String destTable)

translateEvent(JoinPath joinpath, Vector inputKeys)
```

The Database Schema should provide a list of joins (getJoinPaths()) that are possible between the source and destination relations. The method would traverse the graph structure of the database's schema to build this list. The translateEvent() method would then be modified to support these arbitrary joins. The user would choose the join association when coordinating events between two views.

4.6.3 User Interface for Relating Data Sources

Many data sources do not explicitly store associations between the relations. Instead, they are implicit within the data. A spreadsheet is a common example. Data in one sheet may be related to a second, but the association is not made explicit. Snap could provide a user interface that would allow a user to associate data in these separate relations.

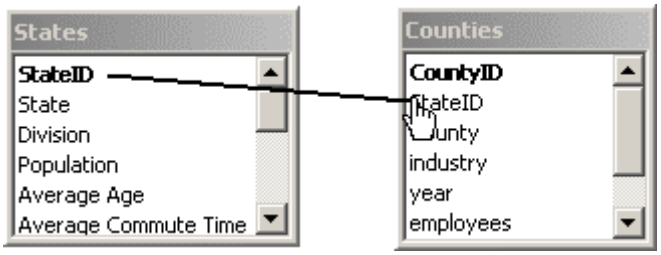


Figure 28 - Database Schema UI extension where users create associations by connecting related fields.

This user interface could be a direct manipulation interface, allowing users to connect related fields by dragging them across relations (Figure 28). The Database Schema UI could provide feedback during event translation. The user interface may also be integrated with higher layers. Events may be used to support drag-and-drop and other interactions with the Database Schema UI.

4.6.4 Integrating Multiple Data Sources

Adding the ability to connect Snap to multiple data sources would be a great extension to Database Schemas [NCI03]. The Database Manager would need to manage connections to multiple sources. Additionally, the Database Schema would need to provide the ability to associate relations between the separate data sources. A user interface similar to Figure 28 would allow a user to create such an association.

Event translation across multiple databases would require the Database Schema to create multiple SQL queries. The Database Schema could build one query joining multiple tables. However, crossing into a separate database will require an additional query built from the results of the first query. For example, the Database Schema shown in Figure 23 could be built with the "Referers" relation stored in one database and the other relations stored in a second database. If three URL tuples are selected with UrlIDs of 4, 5, and 6, then the Database Schema would execute the following queries to translate the URLs into Referers:

```
SELECT Hits.RefererID
FROM Hits INNER JOIN URLs ON Hits.UrlID = URLs.UrlID
WHERE URLs.UrlID IN (4,5,6);

SELECT Referers.RefererID
FROM   Referers
WHERE  Referers.RefererID IN (<Results of previous query>);
```

4.6.5 Integrated Data Mining

The creation of discovery tools that integrate visualization and data mining enable more effective exploration while preserving user control [Shn02]. Further extensions to the Snap model and Database Schemas may include the addition of generalized data mining algorithms [NCI03]. The extensions could support the selection and specification of mining algorithms to establish dynamic associations between relations. Results of mining algorithms may be used to calculate associations at run-time, such as statistical significance or inference rules.

Chapter 5 Visualization Schemas

5.1 Overview

Snap-Together Visualization enables users to dynamically construct a coordinated visualization without programming. Users need to be able to visually build the coordinations, enabling them to create a visual concept for the organization and interactions supported by the multiple-view visualization. The Visualization Schema within the Snap architecture provides the user with this capability (Figure 29). Its goal is to support flexibility in the construction of information visualizations [NCS02]. This layer visually presents the structure of coordinations along with providing an interface for manipulating that structure. The Visualization Schema also provides an overview of how the multiple-view visualization is coordinated, enabling users to quickly learn the capabilities of a new visualization.

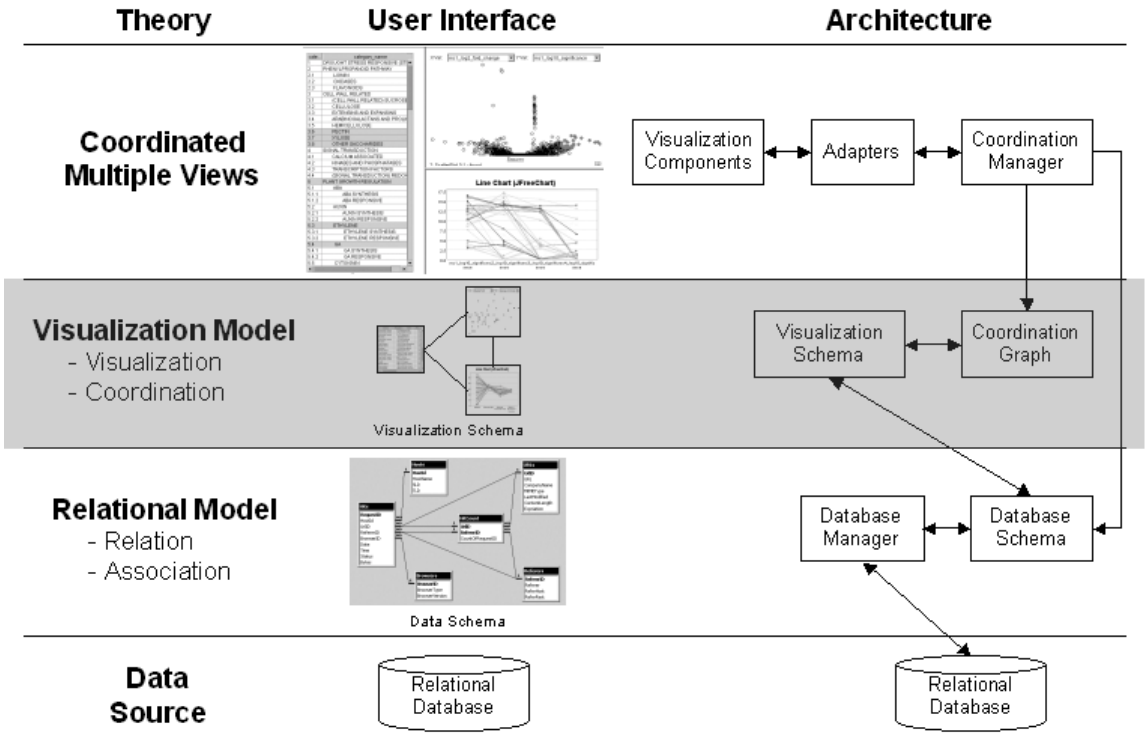


Figure 29 – Visualization Schemas layer provides a framework for visual construction of coordinations.

This layer includes two subsystems: the coordination graph and its user interface. The Coordination Graph encapsulates the organization of coordinations. It maintains the language of coordinated visualization and provides an interface to the Coordination Manager as it propagates events between components. The Visualization Schema is the user interface for constructing the Coordination Graph. It allows users to specify the visualization and the data that they encapsulate. It enables the run-time construction and deconstruction of coordination links. Visualizations Schemas provide top-down support for visualization construction. A user specifies the visualization first and then indicates the data to be loaded into the visualization.

5.2 Coordination Graph

The Coordination Graph is the structure used to represent the organization of a multiple-view visualization. Nodes in this graph encapsulate a visualization component. Links represent the coordination or tight coupling of actions between visualizations. As shown in Figure 30, the Coordination Graph is the central structure used in organizing the multiple-view visualization.

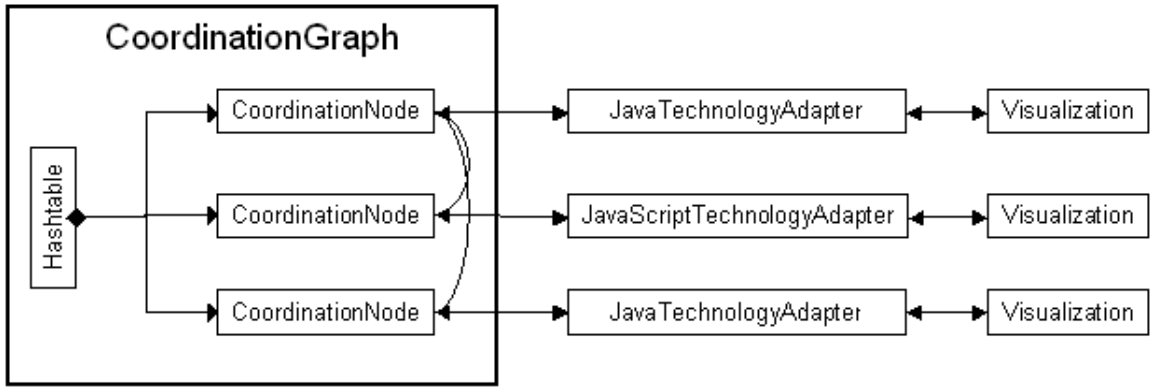


Figure 30 – Coordination Graph of three nodes depicting each node with its handle to the visualization along with its links to other nodes.

Because the Coordination Graph is shared between the Visualization Schemas user interface and the multi-threaded Coordination Manager, access to the graph must be synchronized. A monitor is used to synchronize the reading and writing of the Coordination Graph.

5.2.1 Nodes (Visualizations)

The graph is organized into the class structure shown in Figure 31. The Graph and Node components store the basic structure. The Graph provides an interface for adding and removing nodes. Each Node maintains the handle to the visualization, the query identifying the data encapsulated, and the connections associated with the node. Inheritance is used to provide additional interfaces to the Coordination Manager and Visualization Schema.

The Coordination Graph and Coordination Node extend the basic graph structure and provide additional support for graph traversal. Coordination Nodes maintain properties used during the mark-and-sweep traversal algorithm. While the current implementation does not make use of the Visitor pattern [GHJ95], the Coordination Node is the location proposed for implementing the Accept(Visitor v) extension. The UI Graph and UI Node provide additional extensions, enabling user interface specific properties and methods to be associated with the coordination graph.

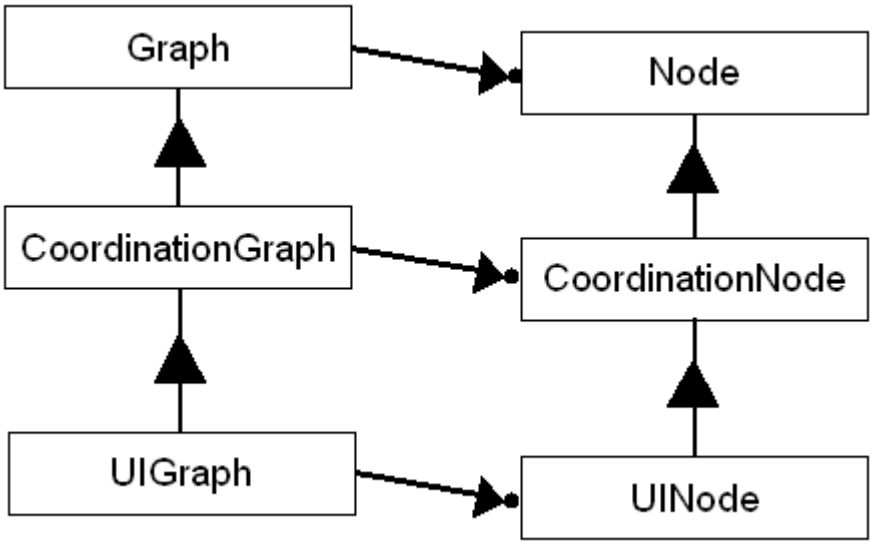


Figure 31 – Coordination Graph class diagram.

The Graph structure provides an interface for accessing the graph and configuring properties of each of the nodes. It maintains a list of event listeners that are notified when the graph has changed (GraphChangeEvent). This implementation of the Observer pattern [GHJ95] provides a capability to coordinate changes between the user interface, coordination manager, and possibly multiple user interfaces in the future.

5.2.2 Links (Coordinations)

Each node maintains a hash table for storing its connections. The hash table is keyed by the actions supported at the node. The value is a list of Node-Action pairs that are connected to this action. The structure of this hash table can be shown as:

```
Hashtable.Key -> List of Node-Action pairs
```

This can be demonstrated using Figure 32. The node for VisA would contain a hash table with an item associated to its "Select" action. Its value is shown in the following list:

```
VisA."Select" -> ( (VisB, "Select"), (VisC, "Select") )
```

This structure states that when a selection event occurs at VisA, the selection events at VisB and VisC should also be fired.

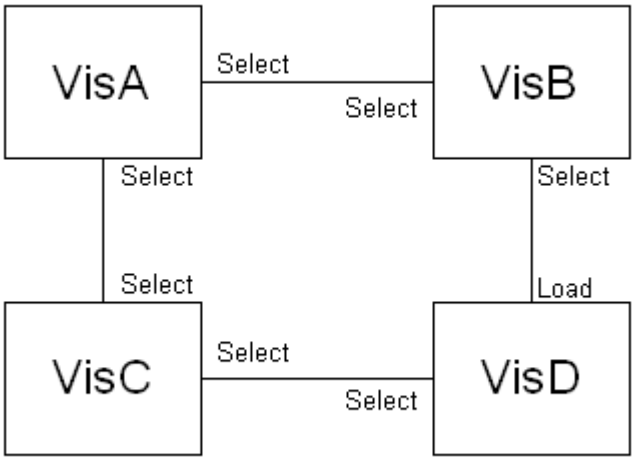


Figure 32 – Example Coordination Graph with four coordinated visualization components.

Similarly, VisD would maintain the following hash table structure at its node:

```
VisD."Select" -> ( (VisC, "Select") )

VisD."Load"   -> ( (VisB, "Select") )
```

This structure improves upon the previous graph structure [Nor00] by supporting both bi-directional and one-directional coordinations. For instance, the bi-directional coupling of VisA and VisB is represented with the following structure:

```
VisA.”Select” -> ( (VisB, “Select”) )
VisB.”Select” -> ( (VisA, “Select”) )
```

These nodes could have a one-directional link by removing one of the hash table entries at either VisA or VisB. This structure closely mimics the language introduced in North's taxonomy for tight coupling of multiple-views [Nor01].

5.3 User Interface

The Visualization Schema is the user interface for coordinating the visualization components. As shown in Figure 33, it provides a visual representation of the Coordination Graph. Users can manipulate the coordinations by interacting with the Visualization Schema. The Visualization Schema provides an interface for flexible design of multiple-view visualizations [NCS02].

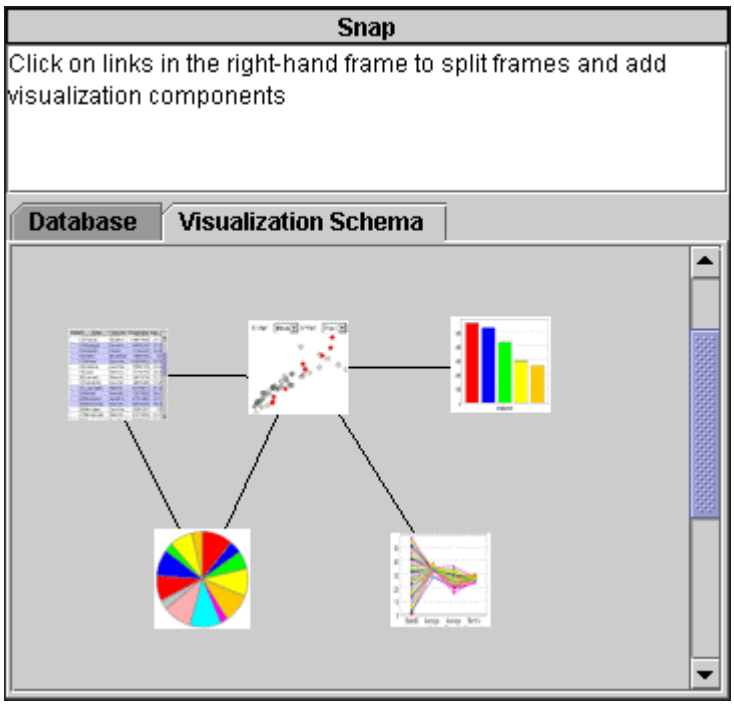


Figure 33 – Visualization Schema depicting the coordination of several visualization components.

5.3.1 View Properties

Users can use the Visualization Schema to configure the properties for an individual view. By right clicking on an icon, a user can either configure the view properties or add a coordination to another component (Figure 34).

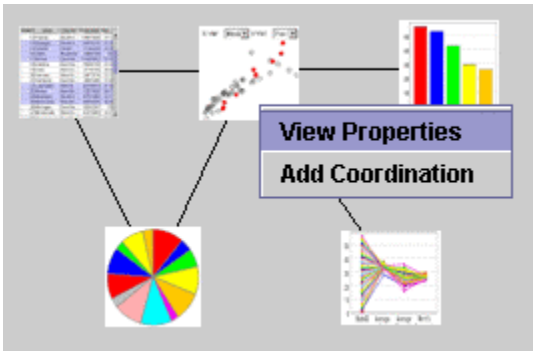


Figure 34 – Right clicking on the component icon brings up a menu for configuring the component.

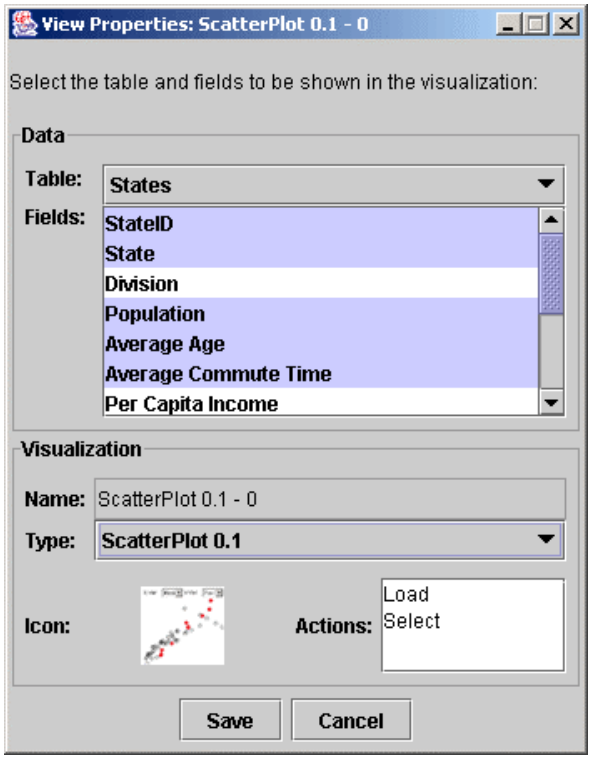


Figure 35 – The View Properties window allows for the configuration of the data and the visualization.

The View Properties dialog window allows a user to specify the data that will be encapsulated by the view (Figure 35). This provides a simple interface for specifying the query that is maintained at the node in the Coordination Graph.

5.3.2 Coordination Properties

A link between two icons represents a coordination between the two views. A single link indicates that one or more coordinations exist between the two views. By right clicking on a link, a user can either Edit or Delete a coordination (Figure 36).

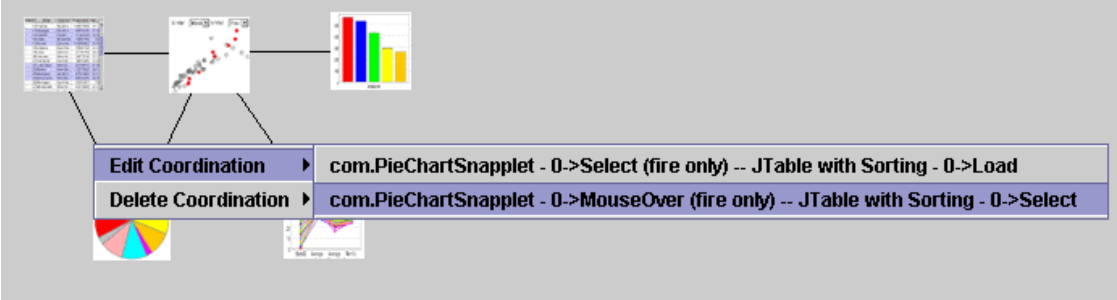


Figure 36 - Right clicking on the link brings up a menu for configuring the coordinations.

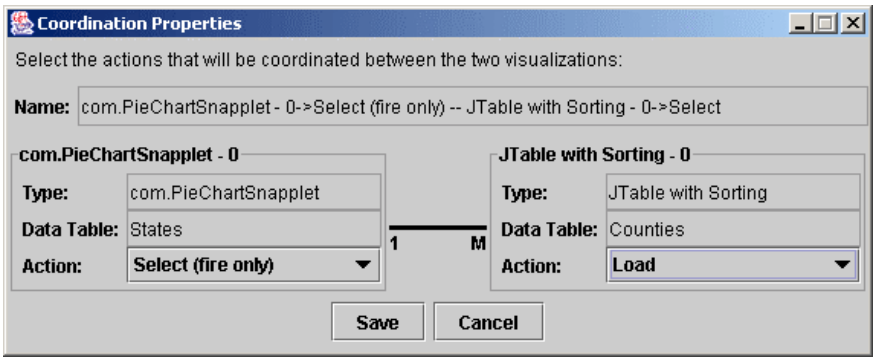


Figure 37 - The Coordination Properties window allows for the specification of a coordination.

When a user deletes a coordination, it is removed from the Coordination Graph. When all coordinations between two views are removed the link is no longer drawn. When a user chooses to edit a coordination, the coordination properties dialog window appears (Figure 37). The Coordination Properties dialog window allows a user to specify the details of the coordination. This window shows the user the properties of each view (visualization type and data relation) along with the cardinalities of the data association between the views (1-M in Figure 37). The user chooses the actions for tight coupling.

5.3.3 Visualization Overview

As a user chooses components in the visualization workspace (right set of frames in Figure 38), their icons appear in the Visualization Schema. The Visualization Schema provides an overview of how the components interact in the multiple-view visualization.

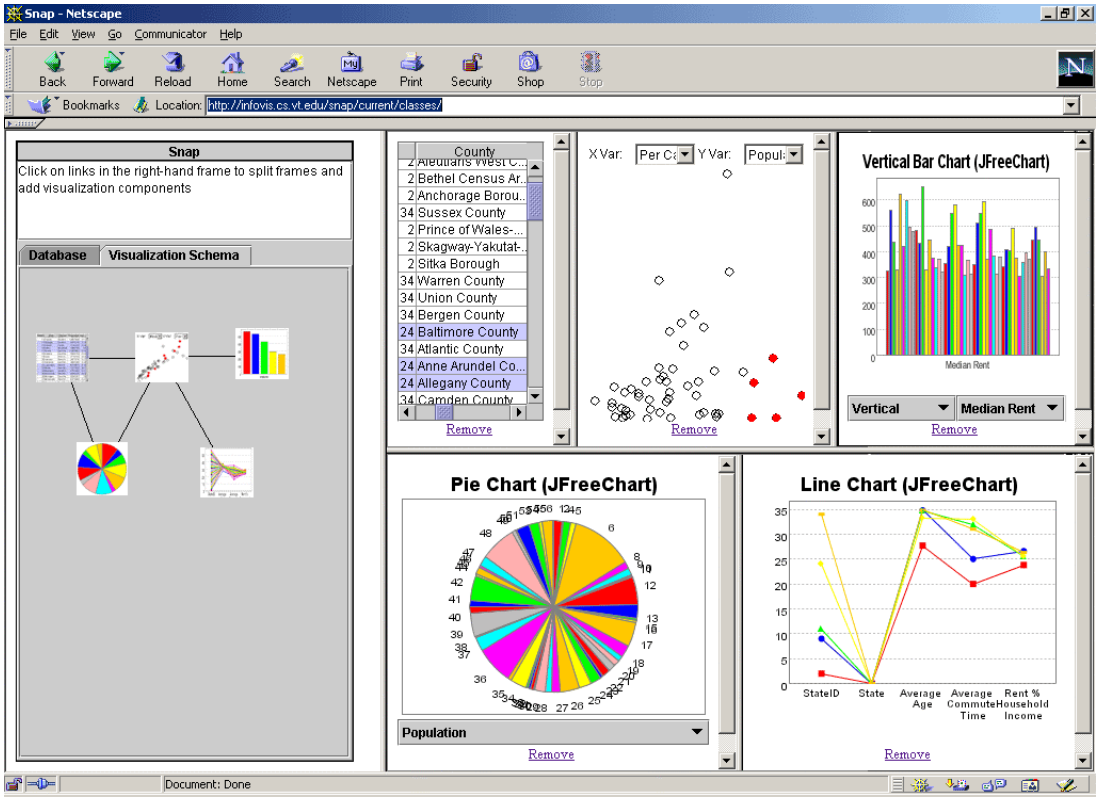


Figure 38 – The Visualization Schema provides a visual overview of how the views are coordinated.

5.4 Summary

The Snap architecture provides a framework for visual construction of the Coordination Graph. This architecture introduces enhancements to the graph structure along with a simple user interface for visually manipulating the structure. This architecture provides a foundation for Visualization Schemas and the capability to build enhanced user interfaces that further exploit the Visualization Schema concept [NCS02].

5.5 Extensions

The Snap architecture provides a foundation for several extensions to the Visualization Schemas layer:

5.5.1 Full Support for Visualization Model

The current implementation of the Coordination Graph and Visualization Schemas only provides support for a self join or a single join. Further support is needed for implementation of a compound join and the selection from multiple alternative joins. Such support would require enhancements to the Coordination Graph. Instead of having links that tightly couple actions, the links will also need to maintain the chosen join that associates the data. The hash table structure of a Coordination Node could be altered in the following manner:

```
Node.Action -> ( (Node, Action, JOIN) )
```

As the Database Schema user interface evolves, further support is needed to integrate the Database Schema with the Visualization Schema. Such an extension would better support integrated data and visualization design.

5.5.2 Multiple User Interfaces

The Coordination Graph architecture supports the Observer pattern [GHJ95] in preparation for supporting multiple user interfaces for visualization construction. Different user interfaces have demonstrated strong support for specific tasks in constructing a visualization [NCS02]. Visualization Schemas enable flexible design of coordinations while DataCompass supports simplified exploration of complex database schemas. Snap could support coordinated user interfaces for constructing the Coordination Graph.

5.5.3 Saved Layout and Bookmarks

The web-based architecture enables a framework for constructing and saving the multiple-view visualization. Further support could be added to save the visualization to a server and allow users to publish their visualization by sending a single URL. The URL would act as a bookmark, loading the constructed visualization from the server and allowing users to begin with the previously constructed visualization.

5.5.4 Compiled Snapplications

Snap could provide users with the ability to compile the constructed visualization. This newly compiled system would roll Snap and the components together into a single distributable file (Snapplication). Executing this Snapplication would allow a user to utilize the custom constructed visualization, but would not provide the user with the capability to redesign the visualization. Such Snapplications would be useful for deployment and support of specific tasks.

5.5.5 Collaboration

An integrated Database Schema and Visualization Schema could support sharing and collaboration in visualization. Such a system could allow users to collaboratively construct and use the multiple-view visualization. Users could share connections to a database and collaborate on how to integrate the data and the visualization. Such collaboration could lead to better designed multiple-view visualizations.

5.5.6 Integrated Window Manager

The top-down approach to constructing visualizations has its strengths and weaknesses. Users can quickly utilize a single visualization and explore the database schema by reassigning data; however, it does not support the ability to quickly reassign a visualization when exploring a

single data relation. Also, the browser-based frame manager introduces technical complications for saving the layout and building compiled Snapplications.

It would be very useful to move to an integrated window manager that would give users more control over the visualization workspace. As well, the Visualization Schema could integrate user actions with actions in the tiled window manager. For instance, the selection of a component or coordination could highlight the corresponding views in the window manager. Such support would better integrate the Visualization Schemas user interface enabling quicker learning of the user interface.

5.5.7 Feed-forward

Feed-forward support could be added to the Visualization Schemas user interface. This support would help users in coordinating visualizations. It would disallow the coordination of visualizations that encapsulate separate unrelated views. Currently, the events are still propagated but not translated. It is assumed that the component receiving the event understands the IDs that are passed to it. The feed-forward would also notify the user when cycles are constructed in the Coordination Graph. The Visualization Schema would then assist the user in creating direct connections that remove ambiguity.

5.5.8 Feedback during Event Propagation

The Visualization Schemas interface could provide feedback to the user as the Coordination Manager propagates and translates events. Links could be highlighted as an event traverses through the graph structure. This feedback would help users in understanding the visualization and how Snap coordinates the individual views. The Visualization Schema could also provide feedback during the time intensive “Load” events. This feedback could indicate to the user the progress of a visualization as it is loading records from the database.

Chapter 6 Coordination Management

6.1 Overview

When users invoke an action in a visualization, Snap will propagate its effects across linked coordinations. Snap will execute all actions that have been coordinated either directly or indirectly to the initiating visualization. The Coordination Manager is responsible for propagating and executing events for each of the coordinated visualizations. The Coordination Manager traverses the Coordination Graph to propagate the event and utilizes the Database Schema to translate the event when necessary (Figure 39).

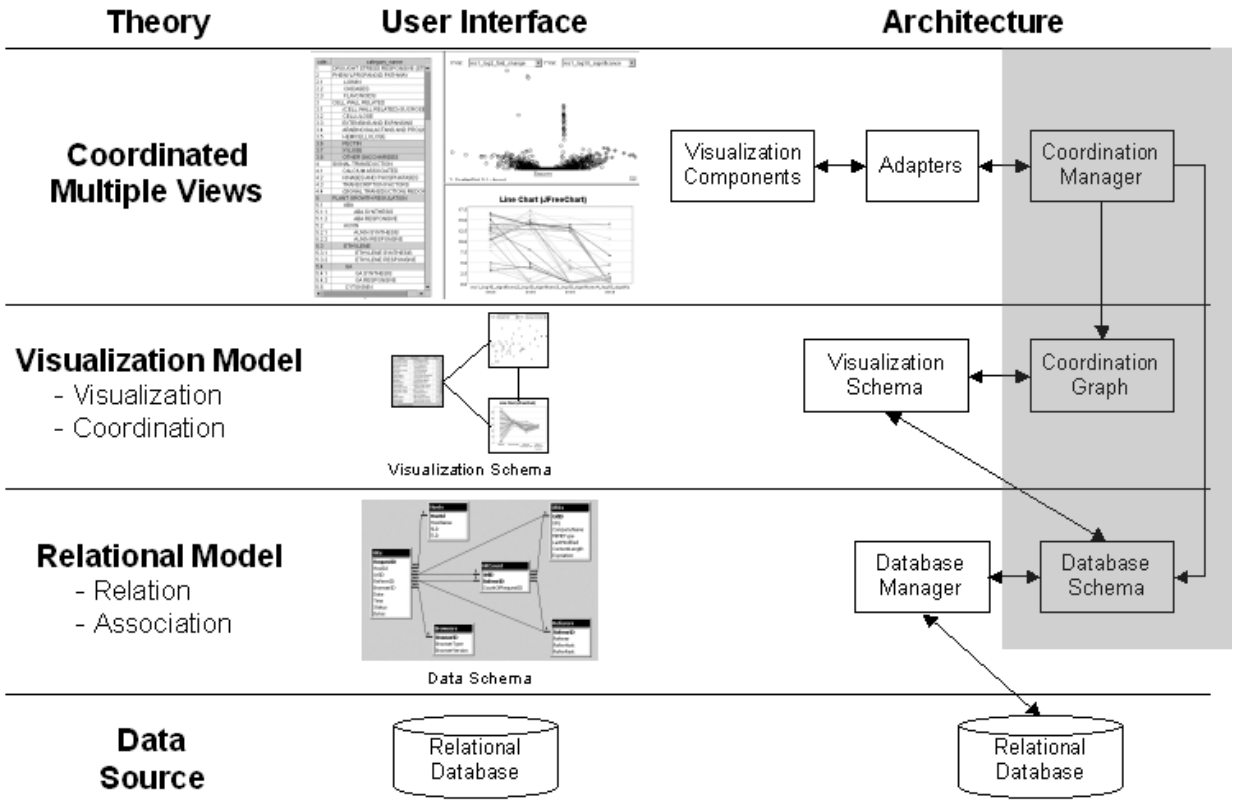


Figure 39 – The Coordination Manager utilizes the Coordination Graph and Database Schema for propagating and translating events.

Each component will fire and receive events based on the data that it encapsulates. The relation's primary key is used as the identifier (ID) when firing and receiving events. The Coordination Manager is responsible for receiving the event and propagating the event to coordinated visualizations. Because related visualizations may encapsulate different data, the Coordination Manager is responsible for translating the event so that a visualization will receive IDs from the perspective of the data that it encapsulates. This allows users to integrate an assortment of visualizations to explore diverse data.

6.2 Event Propagation

By utilizing the Coordination Graph and Database Schema, the Coordination Manager is able to propagate events automatically to coordinated visualizations. The user no longer needs to enforce foreign-key actions or construct parameterized queries. This improvement in the model and implementation allows Snap to automatically coordinate events across associated relations and provide support for multiple-tuple actions.

The Coordination Manager acts as a Mediator [GHJ95] between the individual visualization components. The Snap architecture only requires the visualization component to fire and receive events. This shields developers from having to communicate with other visualization components directly. Visualization components must implement a standardized *Snapable* application programming interface (Chapter 7). This API is designed to be very simple and minimize developers' required effort.

6.2.1 Graph Traversal

Event propagation is done by traversing the Coordination Graph using a mark and sweep algorithm. This algorithm ensures that each coordinated visualization receives the appropriate event only once. Several algorithm alternatives were considered for propagating events and each

has a different effect depending on the structure of the coordination graph. The previous Snap model was conflict free, meaning that there was no ambiguity in propagating events. The implementation utilized a depth first traversal and fired events within a single thread of execution. However, the new model forces the implementation to deal with ambiguity when propagating events.

Algorithm alternatives that were considered include marking each node versus marking each action type associated with a node. Marking each node's action type would allow a component to receive multiple events during the propagation. However, inconsistencies can occur when the actions conflict by operating on different tuples. This can occur when propagating an event through different paths that join two relations in a database. A node may receive a "Load" action and a "Select" action with both actions specifying different tuples. Alternatively, marking each node during event propagation means that only one event will occur in each component. A node may receive different action events, but Snap will only fire the first event that the node receives. The Coordination Manager implements marking at each node so that there are no inconsistencies when propagating events.

Additional alternatives considered include depth-first versus breadth-first traversal. The traversal algorithm affects the order in which events arrive at each node. The Coordination Manager uses a breadth-first algorithm when propagating events, which causes shortest path coordinations to occur first. This approach matches user expectations when creating the Visualization Schema.

As the Coordination Manager traverses the Coordination Graph, it utilizes the Database Schema's event translation interface. As an event is propagated from one node to another in the Coordination Graph, the Coordination Manager requests that the Database Schema translate the

event from the source to the destination. If the Database Schema is not able to determine the association between the relations, no event translation will occur.

6.2.2 Multi-Threaded Event Firing

Once a node is marked, the event is fired to the visualization. The event propagation is executed in a single thread of execution. However, the firing of an event runs in a separate thread for each event. Each node in the Coordination Graph has its own thread used to handle the component's event queue. Event queues are maintained by the node's adapter and are used during both firing and receiving events. This allows the Coordination Manager to continue propagating events while each visualization handles the event that it receives. Multi-threaded event firing allows for continued and rapid interaction within each component even when they are coordinated with slowly updating components.

6.2.3 Cycles

Typical multiple-view visualizations have a tree-structured Coordination Graph. However, cycles may occur in the Coordiantion Graph and methods are needed for handling these cycles. Figure 40 demonstrates a Coordination Graph with cycles. The first example starts with a "Select" event occurring at VisB corresponding to two events ("Select" and "Load") that may occur at VisD. Marking at each node combined with the depth-first traversal provides a consistent solution where VisD will fire the "Load" event because it is the shortest path from VisB.

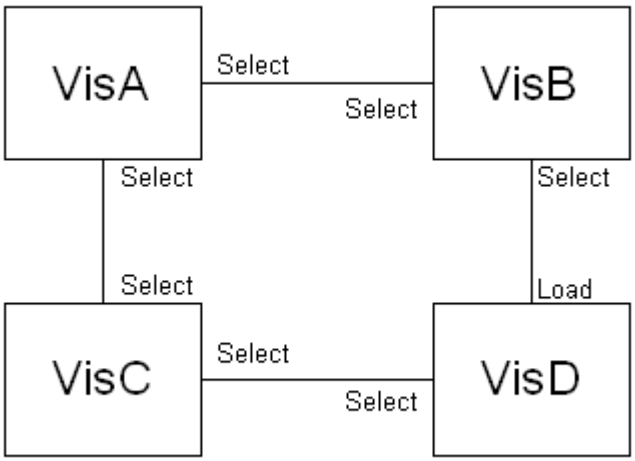


Figure 40 – Example Coordination Graph (also Figure 32) used to demonstrate how cycles are handled.

A second example demonstrates the behavior when two equal-length paths exist. It starts with a "Select" event occurring at VisA corresponding to two events ("Select" and "Load") that may occur at VisD. Both events are an equal number of hops from the initial select event (Figure 40). The Coordination Manager fires events based on the order they were constructed in the Coordination Graph. This ordering is determined from the starting node. Therefore, if the connection from VisA to VisB was created before the connection from VisA to VisC, then the "Load" event will occur at VisD. This implementation detail is subtle and may often be overlooked by a user because they are not always interested in how VisA affects VisD. However, if the user prefers one action to another then they can directly link VisA to VisD. This direct path removes any ambiguity regarding which event will be fired. A fully connected graph eliminates ambiguity because of the breadth-first traversal.

6.3 Extensions

The Snap architecture provides a foundation for several extensions to Coordination Management:

6.3.1 Alternative Joins

As mentioned in the Database Schemas section (4.6.2), support could be added to allow a user to choose from alternative joins. The Coordination Graph would need to support storing the selected join association and the Database Schema interface would utilize the selected join association when translating events for the Coordination Manager.

6.3.2 Multiple Events

The event propagation could be designed to enable multiple events to be fired to an individual component. It is important to consider the ordering of events for the component. Snap could be responsible for ordering events. Another alternative is to pass all events to the component and allow it to resolve the ordering for applying the events. The second alternative would allow Snap to remain flexible regarding the number and types of actions that are supported.

6.3.3 Constraint-based Coordination

A future approach to consider is a constraint-based approach to coordination. Visualization properties could map to fields within the data. For instance, instead of propagating a select event the data would have a column identifying whether or not the tuple is selected. This approach may or may not be exposed to the individual components. The Coordination Manager could still fire events to and from the components while propagating events using the constraint-based coordination. If this model were exposed to the components, the representation of events would need to be well defined and would require a different interface for component interaction.

However, a constraint-based, data-centric model could be built enabling the coordination of actions and visual presentation between visualization components.

Chapter 7 Adapters and the Snapable API

7.1 Overview

The initial implementation of Snap supported a static group of visualization components and required a large install base with each component installed on the machine. Adding new visualization components required recompiling the Snap system, limiting Snap's ability to grow as visualizations were added. The new Snap architecture is run-time extensible, allowing the addition of new visualization components without the need to rebuild or restart the system. It provides developers with a lightweight framework that supports the reuse of visualization components. The architecture introduces the concept of adapters designed to support future integration of various component technologies and data models (Figure 41).

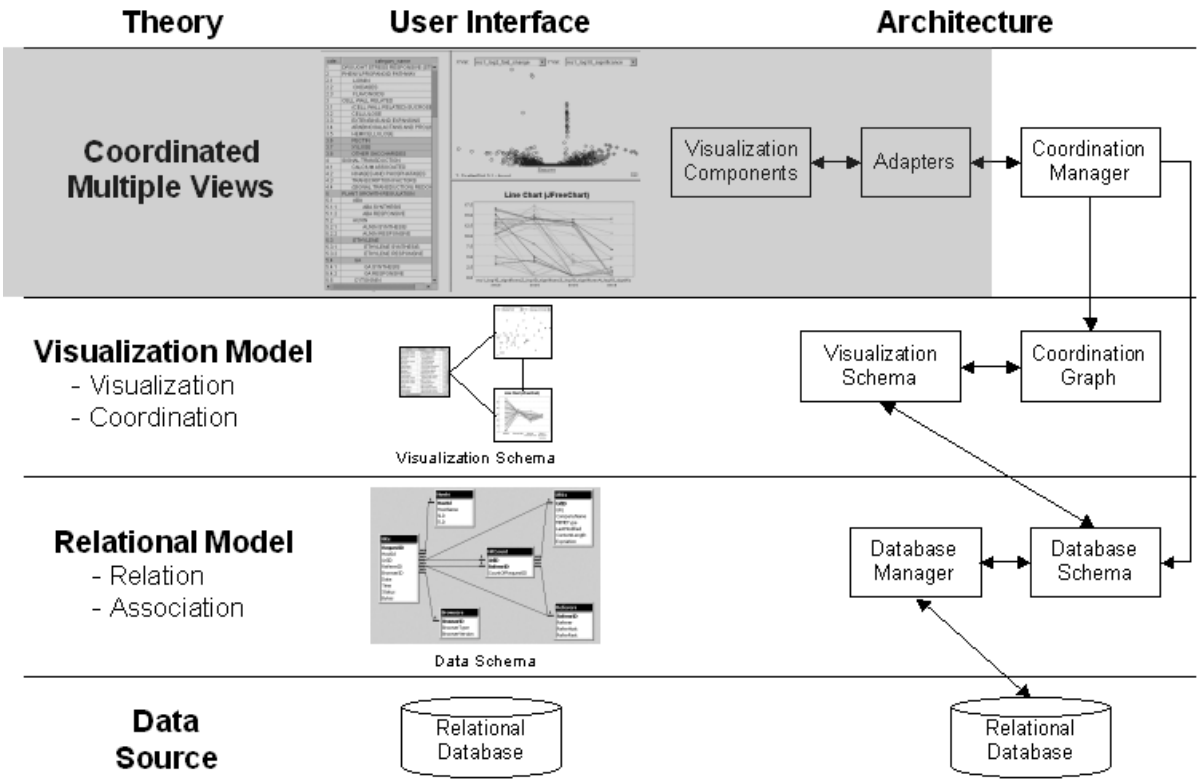


Figure 41 – Adapters enable run-time extensibility and introduces support for various technologies.

The Visualization Workspace allows users to organize and specify the visualizations to load. The visualization is initialized and the Component Loader is responsible for obtaining a handle to the visualization and adding the visualization into the Coordination Graph. This enables top-down construction of the multiple-view visualization by allowing users to select a visualization component first and then load data into the visualization.

7.2 Visualization Workspace

The Visualization Workspace is a set of HTML and JavaScript based frames. It allows the user to split a page into sections by breaking the current frame either horizontally or vertically. This workspace provides a simple space filling approach for tiled windows (Figure 42). Users then click on the component that they choose to load into the visualization.

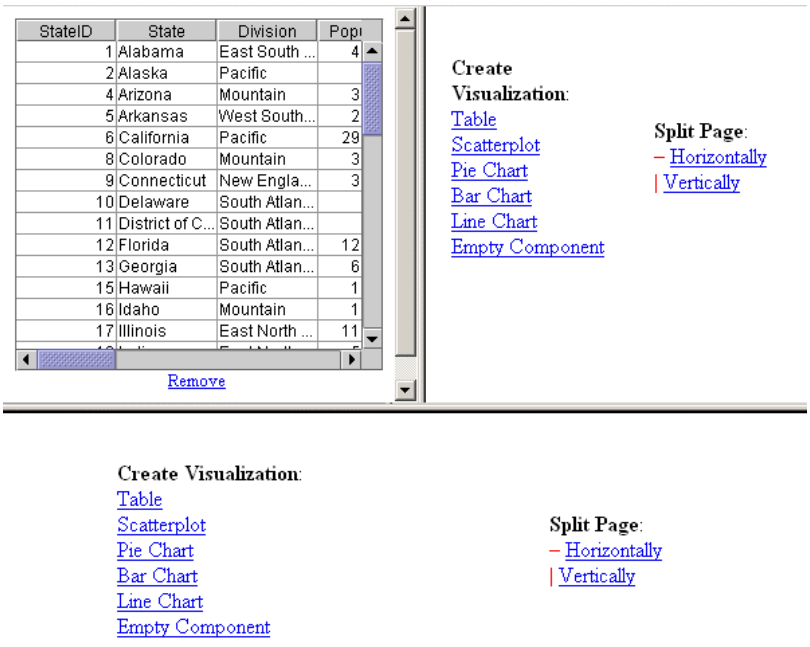


Figure 42 – Visualization Workspace allows the user to arrange and specify visualization components.

7.3 Component Loader

When a visualization is selected to load in a frame, the visualization is initialized and a Component Loader is responsible for obtaining a handle to the visualization and adding the visualization into the Coordination Graph. When applets are selected an Applet Loader is placed at the bottom of the page. The Applet Loader retrieves the visualization's handle using Netscape's LiveConnect interface to JavaScript [Liv02]. The Applet Loader then passes the component's handle to Snap using shared memory [Hil00].

Architectural support for Component Loaders allows Snap to be flexible as advances to the Visualization Workspace are created. If Snap were to move towards an integrated window manager (as described in 5.5.6), the window manager can act as the component loader by adding the visualization component to the Coordination Graph. Similarly, support for an integrated loader and technology adapter could be created to allow Snap to communicate with components outside of the browser.

7.4 Technology Adapters

7.4.1 Overview

Snap enables quick integration of visualization components by developers. New or legacy components can be modified to support Snap's communication API. These components can still operate as stand-alone visualizations outside the Snap environment as needed. However, they can now be coordinated with many other components within Snap.

The Snap architecture makes use of a standard interface to technology specific adapters. Developers can integrate various technologies by implementing the adapter interface. The adapters provide for run-time extensibility in terms of both technologies and components that are supported by the Snap architecture.

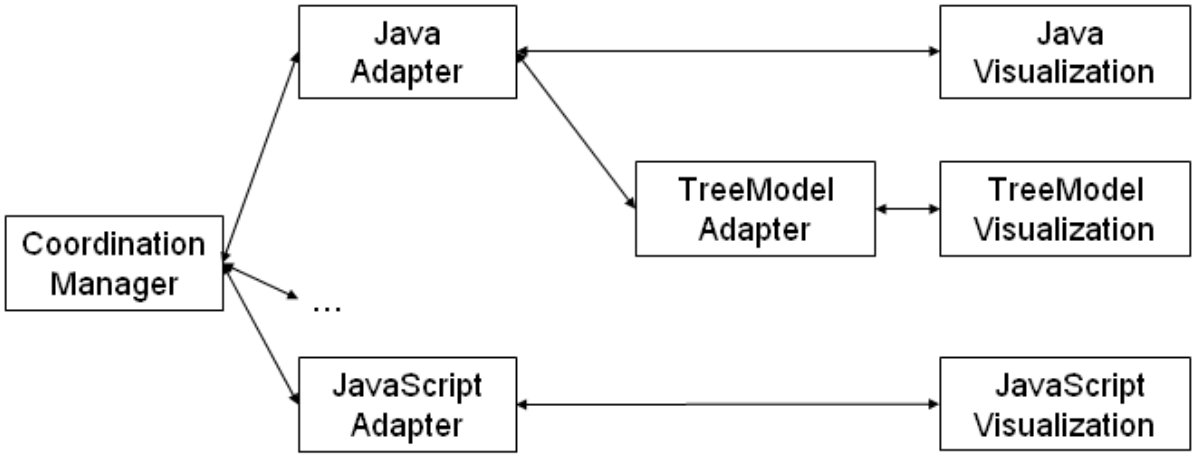


Figure 43 – Snap's architecture supports run-time extensibility through its use of adapters.

The adapter model allows Snap to support various visualization technologies as well as data models (Figure 43). The Adapter pattern [GHJ95] is common in visualization architectures [SML97], [RJH02]. Each adapter translates the interface with the Coordination Manager into technology specific methods and events.

7.4.2 Design Considerations

Adapter design alternatives include support for one adapter per technology versus one adapter per visualization component. Using one adapter per technology would allow the adapter to share large memory structures when interfacing with multiple components. However, it would force the Coordination Graph to maintain a reference for both the adapter and visualization at each node. The alternative design is to have one adapter per visualization. The Coordination Graph would then maintain one reference per visualization. The Coordination Manager would talk to the visualization through the adapter.

The Snap architecture supports the design using one adapter per visualization. This simplifies the interface between the Coordination Manager and the adapters. In cases where adapters require memory intensive components, they can share the component as a Singleton [GHJ95].

7.4.3 Java Technology Adapter

Snap uses Java's ability to dynamically detect and connect to components in a web browser without modification to the Snap system. The Java Technology Adapter is an adapter between the Coordination Manager's general interface and the interface specific for Java visualizations (Snapable Interface described in 7.4.4). This adapter supports multi-threaded events by running in its own thread of execution. This is done because Java's event listener architecture requires event handling to run in the event dispatch thread of execution. By handling events in a separate thread of execution, the Coordination Manager can propagate events without waiting for slower components to update. The adapter queues events as they are passed between the Coordination Manager and the Snapable visualization. All events are queued and fired in a FIFO order.

The Java Technology Adapter also provides a mechanism to ignore possible talkback during event handling. When the adapter fires an event to the visualization component, it is possible that a poorly implemented component would perform the event and fire more SnapEvents in the process. The Java Technology Adapter ignores events that come from the component while the component is processing an event from the adapter. This eliminates possible talkback and allows Snap to cleanly propagate events to all coordinated components.

The Java Technology Adapter adds support for a "Load" action. When any "Load" events are passed to the visualization, the adapter translates the event into a database query. It

then queries the Database Schema and passes the results into the Snapable visualization's load() method. Having the adapter be responsible for querying the database on "Load" events provides support for future technology extensions. The current Java Technology Adapter retrieves results in the form of JDBC ResultSets. Other adapters may be able to retrieve results as JDO Rowsets, ADO Rowsets, or other object based interfaces to a database.

7.4.4 Snapable Interface

Java based visualization components must either implement the Snapable interface or extend the abstract Snapplet class. This interface is designed to allow for an unlimited amount of action events that a visualization can support. The interface makes use of a common SnapEvent that is used to describe events in the Snap architecture. The SnapEvent maintains an association to the source, an event type, and a list of primary key values. The Snapable interface contains the following methods:

```
public void load(ResultSet rs, String primaryKeyColumnName);
public void performSnapEvent(SnapEvent e);
public void addSnapEventListener(SnapEventListener sel);
public void removeSnapEventListener(SnapEventListener sel);
public Enumeration getSupportedActions();
public Icon getIcon();
```

The load() method is called to pass data to the visualization component. The data model used is a standard ResultSet. The primaryKeyColumnName allows the component to identify which column to be used as an ID when firing and receiving events.

The performSnapEvent() method is called when Snap fires an event to the component. The addSnapEventListener() and removeSnapEventListener() methods allow the Coordination Manager to add itself as a listener for fired SnapEvents. When the component needs to fire an event, it will iterate through the list of SnapEventListeners and call the snapEventOccured() method. This closely models the design of event handling in JavaBeans [Sun02].

The getSupportedActions() method provides a list of actions that are implemented by the component. The getIcon() method allows the component to set an icon. If null is returned, the adapter will set a default icon. Detailed instructions for implementing this API are provided in Appendix A.

7.5 Extensions

The Snap architecture provides a foundation for several extensions to the adapter layer and its interface with visualization components:

7.5.1 Additional Adapters

The adapter layer was designed with the intention of adding several different adapters (Figure 43). Support is currently provided for a Java Technology Adapter that provides data in the form of a ResultSet. Several components have built-in support to convert the data from a ResultSet into individual data models. For instance, the table component builds a TableModel and the tree component builds a TreeModel [Sun02]. Adapters could be built that do this translation for the component. The visualization's load() method would then take the model as a parameter. Instead of firing events based on primary keys, the events could be fired based on

identifiers specific to the data model. For instance, the TreeModel interface would fire and receive events that contain TreeNodes.

Additional adapters could also be added to bridge across different technologies. A JavaScript or HTML adapter could be built using Netscape's LiveConnect [Liv02] or Java's support for DOM [Sun02]. Additionally, support could be built to bridge the connection from Java to ActiveX [Rod98].

7.5.2 Decorators

Additional components may also be built to provide services that are needed by several adapters. For instance, the Java Technology Adapter supports multi-threading within the adapter. This service could be pulled out of the Java Technology Adapter and implemented as a Decorator [GHJ95]. Other adapters that need this service could then reuse the multi-threading decorator.

7.5.3 Adapter Configuration Integrated into Visualization Schemas

Adapters could provide user interface support for configuration. Each adapter could implement a user-configurable panel that could be shown by the Visualization Schemas user interface. A user could right click on the visualization icon and have an option to "Configure the Adapter." Configurable properties may include the size and use of event queues, attribute encoding, and other parameters for communicating with the visualization.

7.5.4 Support for Component Submission

Support is needed to automate the process of component submissions. A web-based system could be put into place that would allow visualization developers to upload their components to the Snap system. The system would then automatically integrate the new

component into the system, making it available to users. This could be done using server-side scripting technologies such as PHP, ASP, or perl.

7.5.5 Event Design supporting Dynamic Queries

An event specification could be built that details the events needed to support dynamic queries. Event propagation between components encapsulating the same relation is fast. This is because no event translation is needed and the event is propagated without additional queries to the database. This is typically how visualization components and data are designed in dynamic query systems [AS94], [AW95]. Events and coordinations could be specified in order to support dynamic queries. These events may include "RemoveFromCurrentSelection" and "AddToCurrentSelection". Such events would then be coordinated by linking together all remove events and all add events. Further coordination design and implementation is needed to verify support for dynamic queries.

Chapter 8 Conclusions

8.1 Contributions

The Snap architecture enables a user to visually construct and coordinate a multiple-view visualization. It allows the user to build a custom visualization that supports their needs for information exploration. This research builds on the initial work that introduced Snap-Together Visualization [Nor00]. This work provides a detailed architecture and introduces the following contributions:

- **Enhanced Visualization Model**: This model allows for the tight coupling of multiple-tuple actions and provides support for event translation using underlying data associations. The new model reduces the coupling of visualization components, removing the user's need to construct parameterized queries that connect primary-key and foreign-key actions. The new model provides enhancements to the underlying theory for specifying coordinations.
- **Extensible Architecture**: This architecture supports run-time extensibility and provides flexibility in the construction of multiple-view visualizations. New visualization components can be added without the need to rebuild or restart the system. The architecture provides developers with a lightweight framework that supports the reuse of visualization components. A simple visual language is introduced that promotes flexibility in constructing and coordinating a multiple-view visualization.
- **Web-based Visualization Server**: The web-based system provides universal access, easy distribution, and the capability to load visualization components on demand. The server makes it easy to integrate and exploit existing components. This research builds a foundation

for a web-based visualization server that can support future goals of visualization publishing and collaboration.

- **Diverse Data Integration**: The new architecture enables connectivity to both local and networked database systems. It supports automatic retrieval of relations and their associations from a database. This architecture enables the integration of diverse data allowing users to spend less time massaging the data prior to visualization.

8.2 Limitations and Future Work

The limitations and potential future extensions have been discussed in each chapter. The Snap architecture builds a foundation that supports many future goals. The architecture provides a framework for these extensions, providing direction so that these extensions can now be implemented and easily integrated. The Snap team is currently working on projects that support several extensions.

Snap's reliance on database queries introduces performance issues. This is especially the case when connecting to remote databases or when coordinating highly frequent actions, such as "Mouse Over", to database intensive actions such as "Load". Performance improvements can be made by introducing data caching at the Database Schema level. These performance issues will continue to lessen as Java's database technologies improve.

Additional extensions that are underway include an enhanced user interface that improves upon the Visualization Schemas layer and introduces the capability for integrated layout management. These extensions will integrate into the Snap architecture and provide greater support to the user. Work is also underway towards the construction of a web-based system for

component submission. This extension will automate component integration and enable developers to quickly distribute their visualization components.

Additional extensions include the integration of generalized data mining techniques. Data mining algorithms can be used to compute new associations based on selections, or to filter static associations based on probability or confidence calculations. This would enable Snap to become a flexible discovery tool [Shn02]. Such an extension could further generalize the model of coordination. The architecture supports the integration of general visualization components, but could further evolve to support the integration of generalized coordinations. Such coordinations could be built on database associations, data mined associations, or even dataflow concepts.

8.3 Discussion

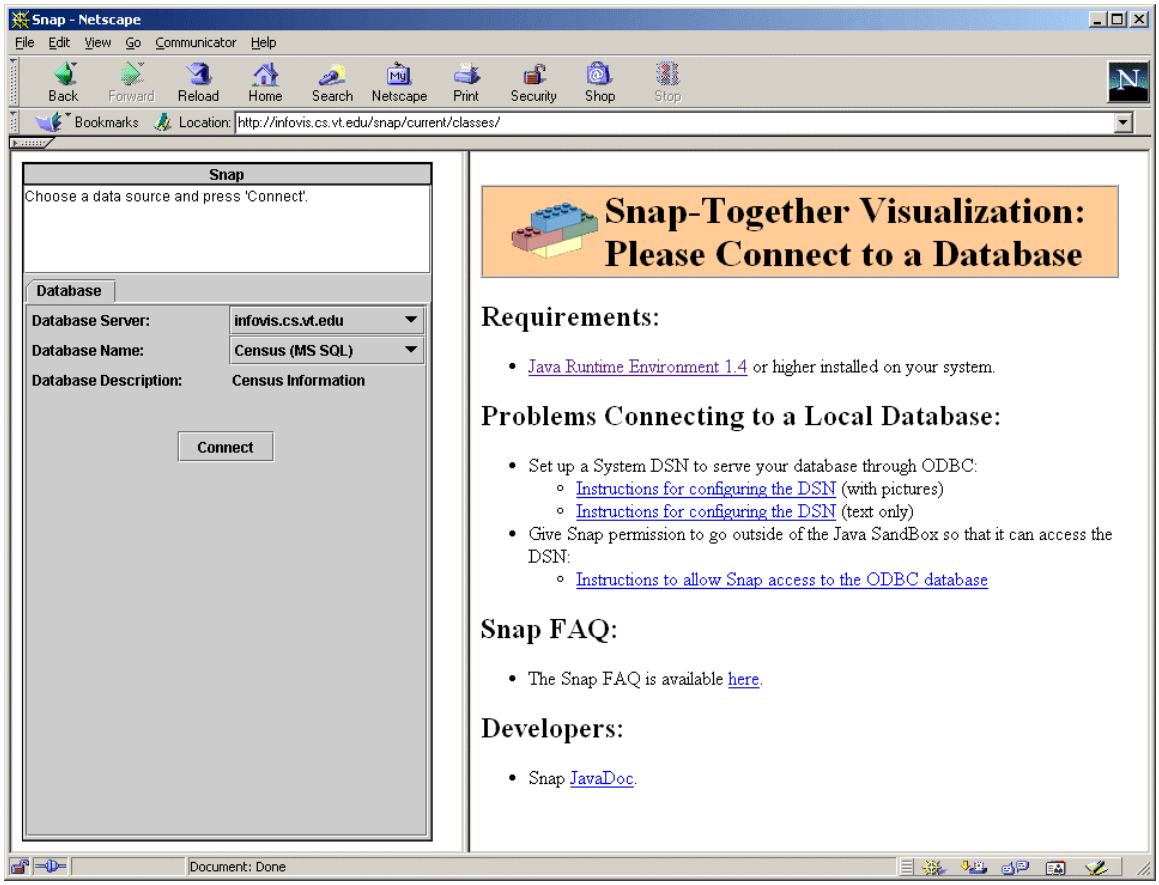
As mentioned throughout this text, many extensions can be made to Snap-Together Visualization. This research provides a strong foundation for the Snap architecture and supports further possibilities for applied information visualization. It provides innovation to important work [Nor00] and is another step towards "crossing the chasm" [Shn01] – helping a wider range of users to explore data, make complex decisions, and apply their creativity. The Snap architecture progresses towards applied information visualization and enabling visualization for the masses.

Appendix A Component Developer Instructions

Install Java

1. Retrieve the latest version of J2SE from http://java.sun.com/j2se/downloads.html. If you are using an IDE, you will need at least the J2SE's JRE. If you are using Sun's compiler, you will need J2SE's SDK. In our example, we are using J2SE 1.4.1 and decide to download the SDK.

2. Execute the **classes\index.html** to test out Snap on your installation. Follow the documentation (in the right frame) to set up an ODBC database and allow Snap access to the database. Test out Snap using a simple MS Access database.



Creating a Source Directory Structure

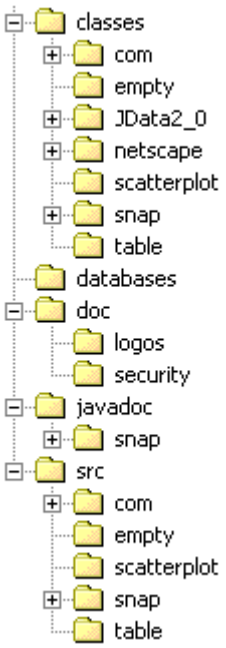
1. Unzip the Snap source into a working directory

2. Decide on a **package name** for your component. In this example, we have chosen *table* for our package name.

3. **Create a directory** with the name of your package under the **src** and **classes** directories. In our example, the directories needed include: *src\table* and *classes\table*.

4. Place your source code in your directory. It should have the line **"package <packageName>;"**. In our example, we include the line *"package table;"*

5. Add the **classes** directory to you **CLASSPATH** variable. This may be done in your IDE or Makefile.



6. When the class files are compiled, they need to end up in the **classes\packageName** directory. In our example, the table classes appear in the *classes\table* directory. You may use a Makefile or configure your IDE to automatically compile these files in the appropriate directory.

Create Supporting HTML Files

1. Copy the **classes\table\index.html** into the classes directory for your package.

2. Change the **TITLE** on Line 3 so that it matches your package name.

3. Change the value of **appletCODE** on Line 23. The new value should be in the format of **"packageName.sourceFile.class"**. In our example, it is *"table.TableSnapplet.class"*.

4. The **appletCODEBASE** value should point to the classes directory. By default, a value of ".." does this for you.

5. Edit the **classes\visualizations.html** file to add a link to your new component:

   a. Copy and paste the line for the Table (Line 15):

   ```
   <br><a href="table/index.html">Table</a>
   ```

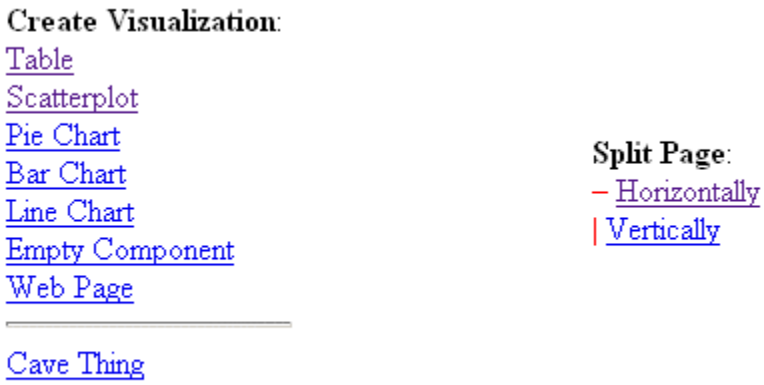
   b. Change the values for the link and the component name:

   ```
   <br><a href="packageName/index.html">New Component Name</a>
   ```

   c. The final result should look like this:

   ```
   <br><a href="table/index.html">Table</a>
   <br><a href="packageName/index.html">New Component Name</a>
   <br><a href="scatterplot/index.html">Scatterplot</a>
   ```

6. A link for your new component should now appear and connect to your index.html file.



Extend Snapplet or Implement Snapable Interface

1. **Copy skeleton code** from src\empty\EmptySnapplet.java into your directory. Note that the skeleton class file extends Snapplet (found in src\snap\Snapplet.java).

2. You will need to **change the package declaration, class declaration, and the file name** (to match the class name). In our example, the package declaration becomes *"package table;"* (Line 2), the class declaration becomes *"public class TableSnapplet extends Snapplet"* (Line 17), and the file name becomes *"TableSnapplet.java"*.

3. **Compile and test** your basic Snapplet component. It should act the same as the EmptySnapplet.

4. If you choose not to extend Snapplet and are Java savvy, feel free to simply have your code implement the Snapable interface. The JavaDoc documentation for Snap will help you with understanding this interface.

Implement the Load Method

1. The **void load(ResultSet rs, String primaryKeyColumnName)** is the method used to load data into your component. Snap will pass your component data in the form of a ResultSet. The primaryKeyColumnName identifies the name of the column that holds the identifier of the data. This identifier is used when firing or receiving events.

2. Change the load method to read the data from the ResultSet and store into a data structure appropriate to your component. A sample data structure to use is a ResultSetTableModel found in src\table\ResultSetTableModel.java. Feel free to copy this class into your src directory (make sure to change its package information).

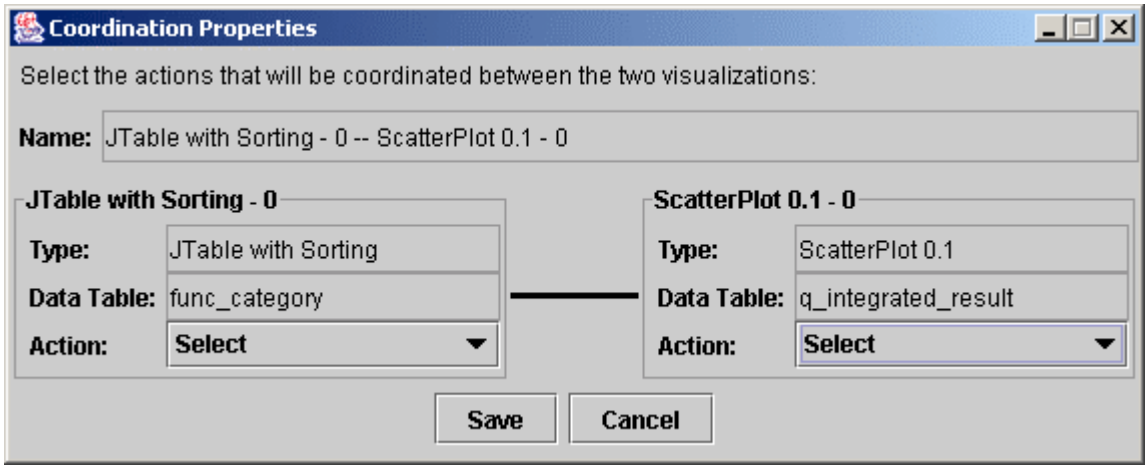
3. Update your view. Remember that the load() method may be called multiple times.

Implement the Component Actions

1. The **Enumeration getSupportedActions()** method is used by Snap to find the actions that your component supports. For example, the EmptySnapplet specifies the "Select" action as a supported action (Line 30).

2. **Potential actions** may include: Load, Select, MouseOver, ScrollTo, ZoomTo, and other actions related to specific visualizations.

3. Implement the getSupportActions() method to return the Enumeration of actions that your component will support. These actions will show up in the drop-down when configuring a link.



4. The **void performSnapEvent(SnapEvent e)** method is called when you receive an event at your component. The **SnapEvent** (found in src\snap\SnapEvent.java) contains the **eventType** and the **keys** for the event. For example, our table would receive a selection event that includes an eventType of "Select" and the keys Vector would include the list of identifiers for the rows to select in the table.

5. This method should determine which SnapEvent has occurred and execute that event for your component.

6. To fire a SnapEvent, call the **void fireSnapEvent(SnapEvent snapEvent)** method. This method is inherited when you extend the Snapplet class. You will need to instantiate a **SnapEvent** object with the **eventType** and **keys** Vector for the event that occurred in your component. The SnapEvent object will be passed to fireSnapEvent(). For example, our table will create a SnapEvent for selection with an eventType of "Select" and the keys Vector that includes the list of identifiers for the rows that were selected.

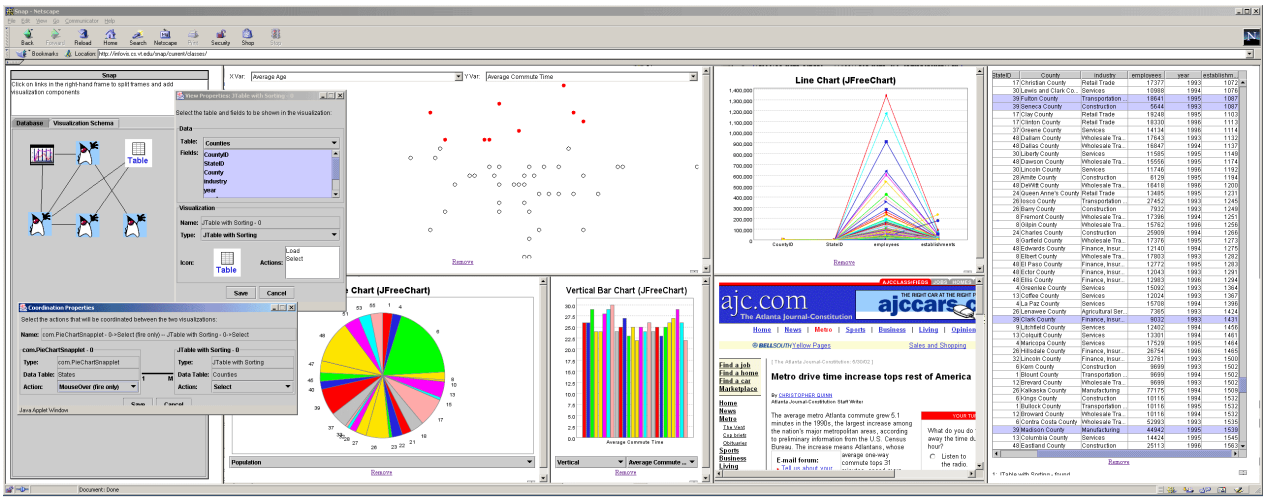
7. Note: If you are implementing the Snapable interface instead of extending the Snapplet class, you will need to keep track of the listeners and fire events where appropriate. Feel free to copy the addSnapEventListener(), removeSnapEventListener(), fireSnapEvent() methods from the Snapplet.java source.

Additional Implementation Notes (Not required methods)

1. The **Icon getIcon()** method is used to retrieve the Icon for component. This Icon shows up in the Snap user interface.

2. If you would like to change from the default Icon, overload the getIcon() method:

```
public Icon getIcon(){
      Icon icon = null;
      try{
            String base = getCodeBase().toString();
            String iconURL = base + "table/table.gif";
            icon = new ImageIcon(new URL(iconURL));
      }
      catch(Exception e)
      {
            System.err.println("Table Demo: " +
                  "Unable to create an icon: " + e.toString());
      }
      return icon;
}
```



3. The String getAppletInfo() is used to set the name of your component in the Snap user interface. By default, the component's name is the class name.

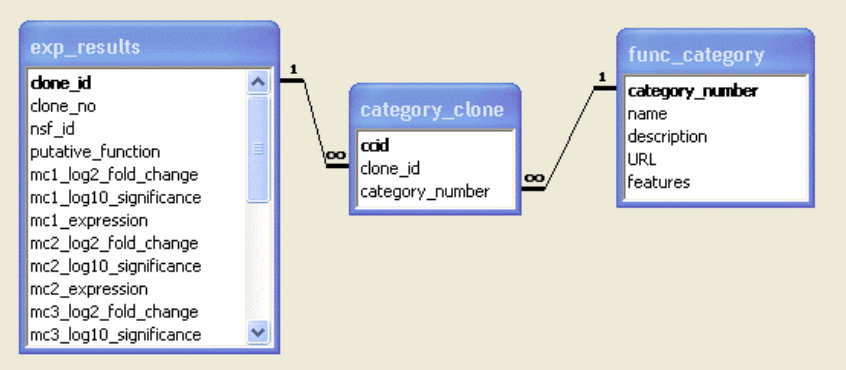
4. Overload this method to change the component's name:

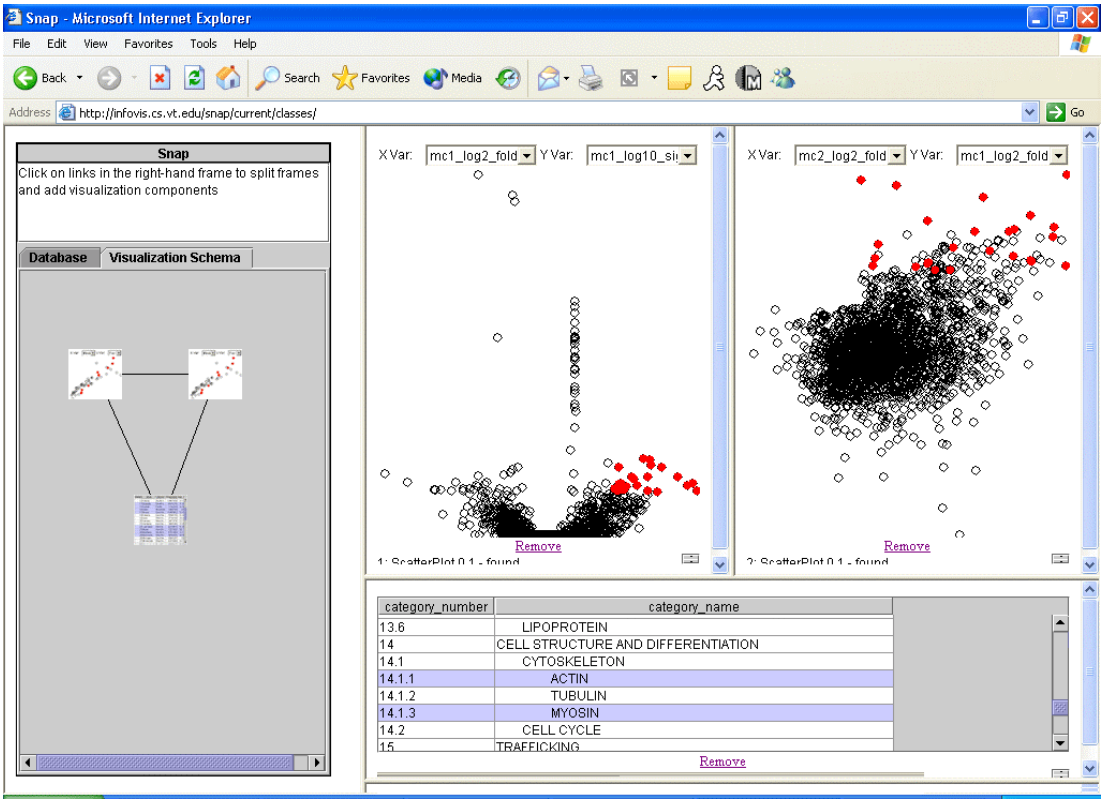
```
public String getAppletInfo(){
      return "Nathan and Craig's kewl table component"
}
```

Appendix B Scenarios

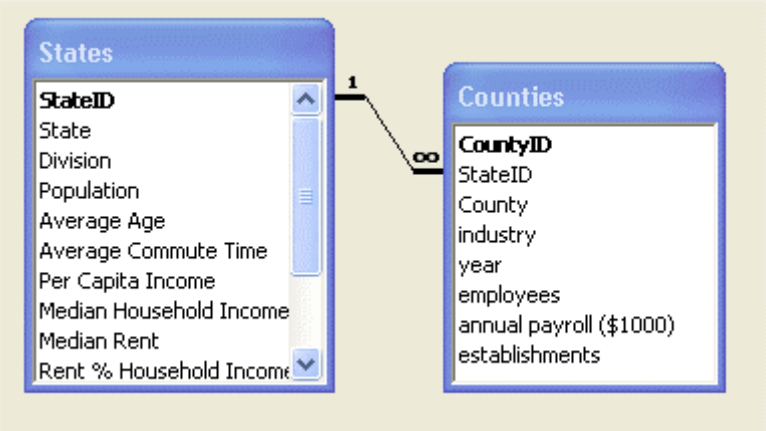
The Snap visualization server has been used to construct a variety of visualizations and explore a variety of data sets. Scenarios are shown that depict both database schemas and corresponding visualizations that have been built to visualize the data.

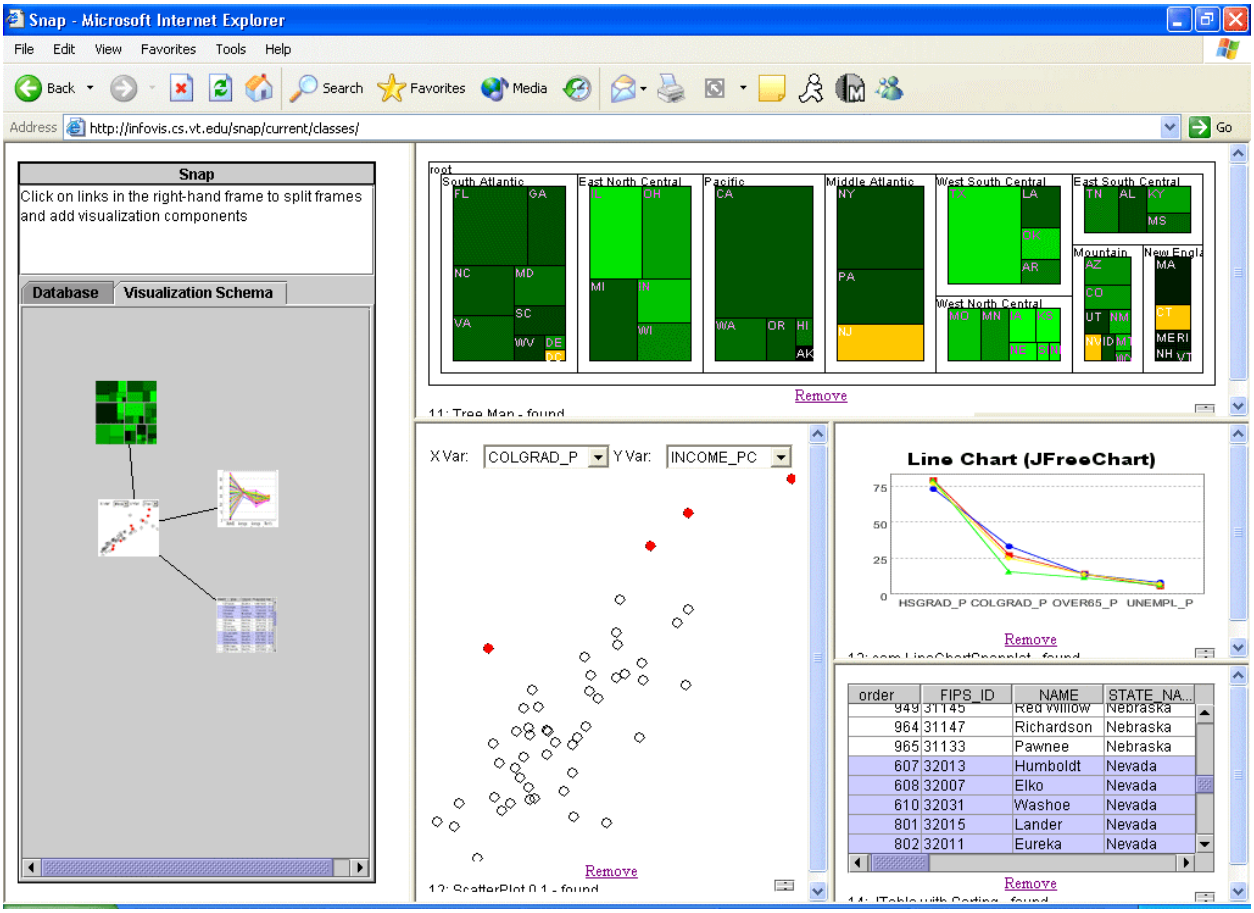
Bioinformatics Data



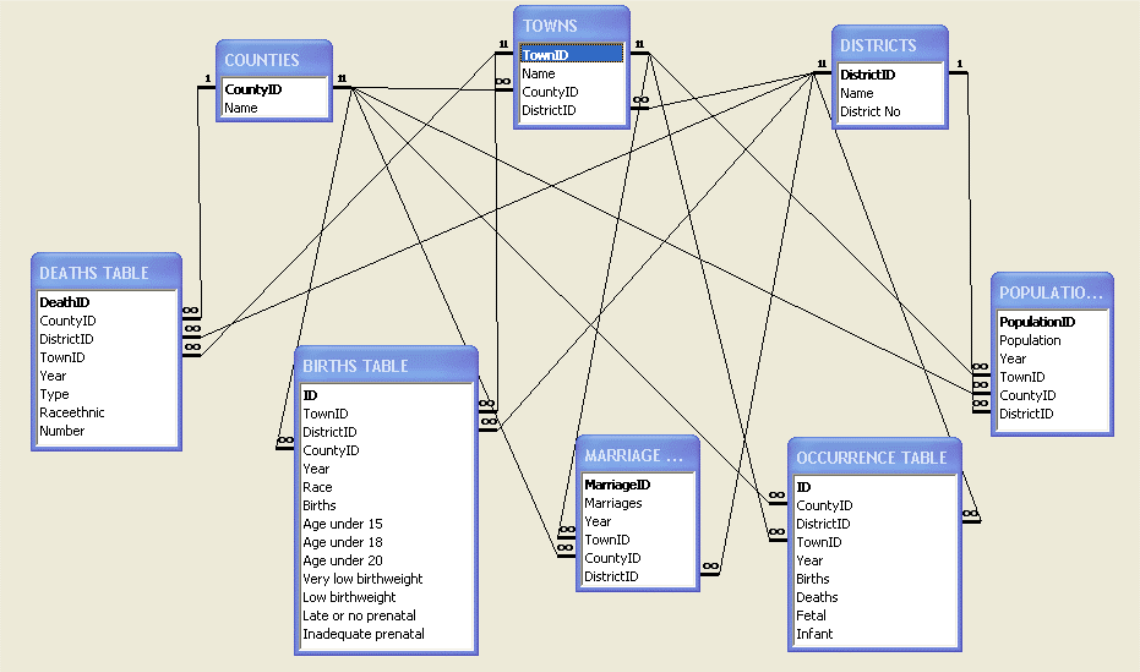


US Census Data

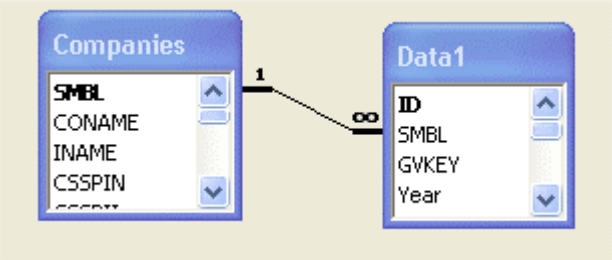


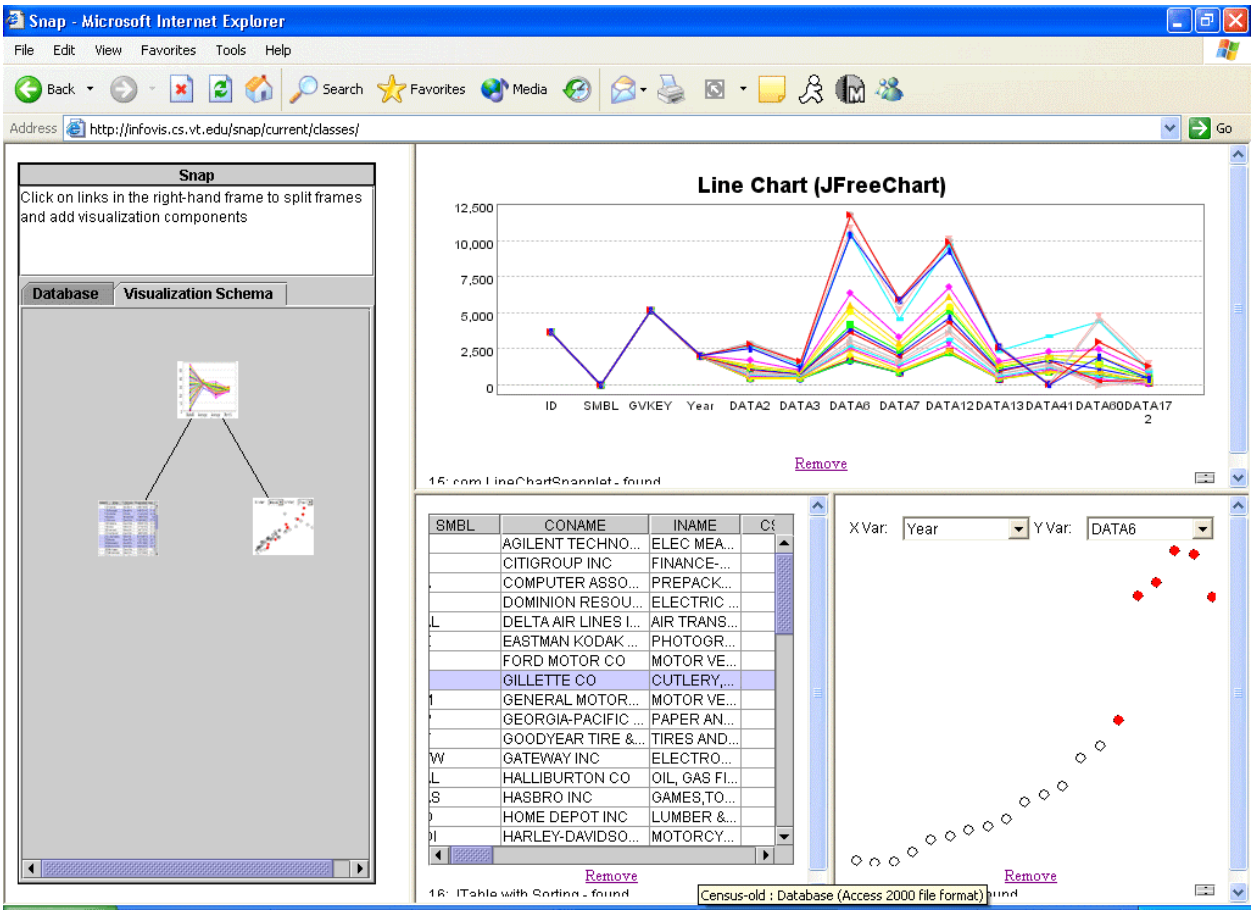


Connecticut Vital Statistics

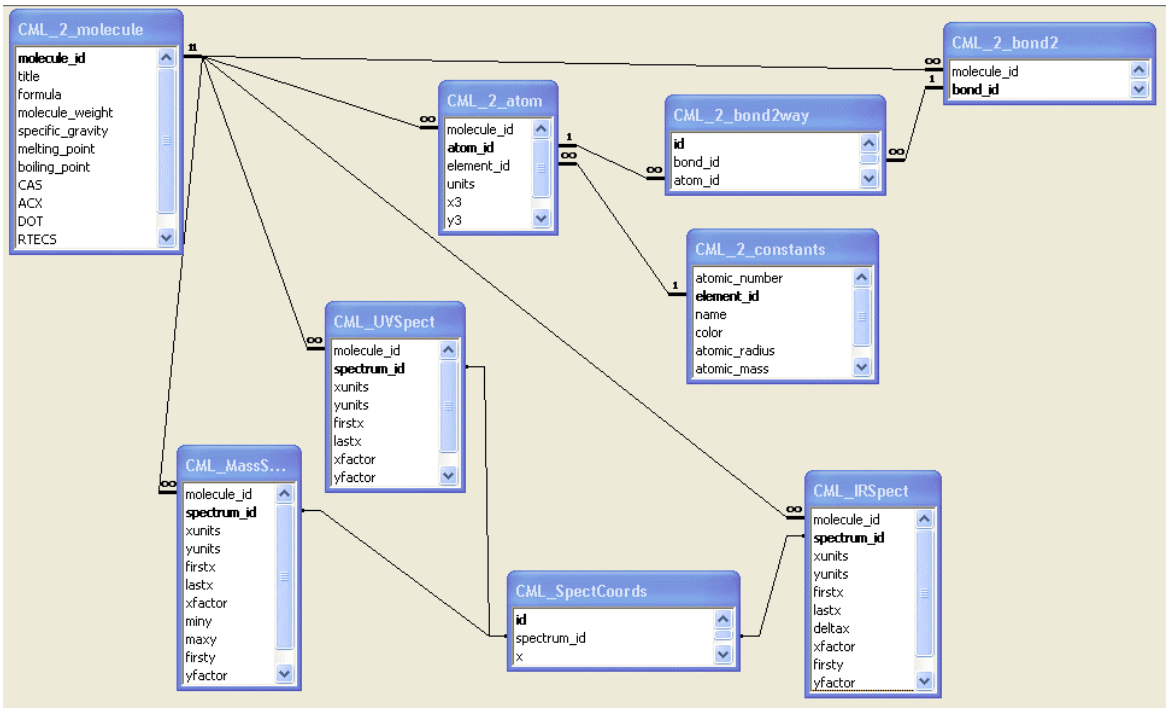


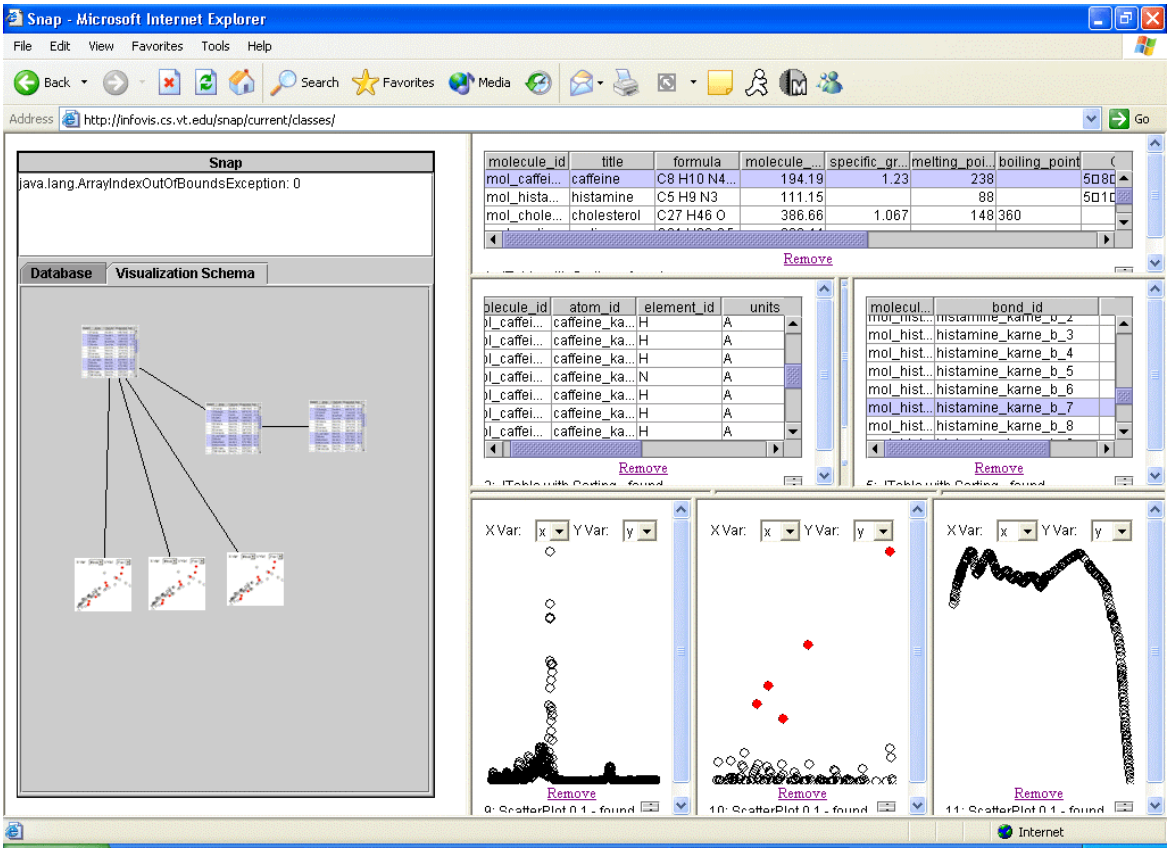
COMPUSTAT Financial Data





Molecular Data





References

[ACS96] Aiken, A., Chen, J., Stonebraker, M., Woodruff, A., "Tioga-2: A Direct Manipulation Database Visualization Environment", *Proc. 12th International Conference on Data Engineering*, pp. 208-217, (February 1996).

[AS94] Ahlberg, C., Shneiderman, B., "Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays", *Proc. ACM SIGCHI 1994*, pp. 313-317, (1994).

[AW95] Ahlberg, C., Wistrand, E., "IVEE: An Information Visualization and Exploration Environment", *Proc. IEEE Information Visualization 1995*, pp. 66-73, (1995).

[BWK00] Baldonado, M., Woodruff, A., Kuchinsky, A., "Guidelines for Using Multiple Views in Information Visualization", *Proc. ACM Advanced Visual Interfaces 2000*, (2000).

[BC87] Becker, R., Cleveland, W., "Brushing scatterplots", *Technometrics*, 29(2): 127-142, (1987).

[BMG00] Bederson, B., Meyer, J., Good, L., "Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java", *Proc. ACM UIST 2000*, pp. 171-180, (2000).

[BBG96] Bergin, J., Brodlie, K., Goldweber, M., et. al., "An overview of visualization: its use and design", *Proc. ACM SIGCSE 1996*, pp 192-200, (1996).

[BST00] Bosch, R., Stolte, C., Tang, D., Gerth, J., Rosenblum, M., Hanrahan, P., "Rivet: A Flexible Environment for Computer Systems Visualization", *ACM SIGGRAPH Computer Graphics*, 34(1), (2000).

[CMS99] Card, S., Mackinlay, J., Shneiderman, B., *Readings in Information Visualization: Using Vision to Think,* Morgan Kaufmann, (1999).

[CRI01] Carroll, J.M., Rosson, M.B., Isenhour, P.L., Ganoe, C.H., Dunlap, D.R., Fogarty, J., Schafer, W.A. and Van Metre, C.A., "Designing Our Town: MOOsburg", *International Journal of Human-Computer Studies*, (2001).

[CBR97] Chi, E. H., Barry, P., Riedl, J., Konstan, J., "A Spreadsheet Approach to Information Visualization", *Proc. IEEE Information Visualization 1997*, pp. 17-24, (1997).

[CPN02] Conklin, N., Prabhakar, S., and North, C., "Multiple Foci Drill-Down through Tuple and Attribute Aggregation Polyarchies in Tabular Data", *Proceedings of IEEE InfoVis 2002 Symposium*, pp. 131-134, (October 2002).

[D90] Dyer, D., "A dataflow toolkit for visualization", *IEEE Computer Graphics and Applications*, 10(4), pp. 60-69, (1990).

[FNP99] Fredrikson, A., North, C., Plaisant, C., Shneiderman, B., "Temporal, Geographical and Categorical Aggregations Viewed through Coordinated Displays: a Case Study with Highway Incident Data", *Proc. ACM CIKM '99 Workshop on New Paradigms in Information Visualization and Manipulation*, (1999).

[GHJ95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, (1995).

[HCY99] Hayden, S., Carrick, C., Yang, Q., "Architectural Design Patterns for Multiagent Coordination", *Proc. International Conference on Agent Systems 1999,* (1999).

[Hil02] Hill, Tobias, "Java Tip 101: An alternative way for applet-to-applet communication", [WWW document], http://www.javaworld.com/javatips/jw-javatip101_p.html, (accessed November 2002).

[HS02] Hochheiser, H., Shneiderman, B., "A Dynamic Query Interface for Finding Patterns in Time Series Data", *Proc. ACM CHI 2002*, pp. 522-523, (2002).

[IBH97] Isenhour, P., Begole, J., Heagy, W., Shaffer, C., "Sieve: A Java-Based Collaboractive Visualization Environment", *IEEE Visualization 1997*, pp. 13-16, (1997).

[KR93] Konstantinides, K., Rasure, J., "The Khoros Software Development Environment for Image and Signal Processing," *IEEE Journal of Image Processing*, (1993).

[KP88] Krasner, G.E., Pope, S.T., "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80", *Journal of Object Oriented Programming*, 1(3):26-49, (1988).

[KGM02] Kuchinsky, A., Graham, K., Moh, D., Creech, M., "Biological Storytelling: A Software Tool for Biological Information Organization Based upon Narrative Structure", *Proc ACM Advanced Visual Interfaces*, (May 2002).

[Liv02] LiveConnect, [WWW document], http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/livecon.htm, (accessed November 2002).

[LRB97] Livny, M., Ramakrishnan, R., Beyer, K., Chen, G., Donjerkovic, D., Lawande, S., Myllymaki, J., Wenger, K., "DEVise: integrated querying and visual exploration of large datasets", *Proc. ACM SIGMOD'97*, pp. 301-312, (1997).

[MWH99] MacEachren, A., M. Wachowicz, D. Haug, R. Edsall, and R. Masters. "Constructing Knowledge from Multivariate Spatiotemporal Data: Integrating Geographic Visualization with Knowledge Discovery in Database Methods". *Intl. Journal of Geographic Information Science*, 13(4): 311-334. (1999).

[Mac86] Mackinlay, J., "Automating the design of graphical presentations of relational information", *ACM Transactions on Graphics*, 5(2), pp. 111-141, (1986).

[MMB95] Masui, T., Minakuchi, M., Borden, G., Kashiwagi, K., "Multiple-View Approach for Smoth Information Retrieval", *Proc. ACM Symposium on User Interface Software and Technology (UIST 95)*, pp. 199-206, (1995).

[M02] Microsoft Open Database Connectivity, [WWW document], http://msdn.microsoft.com/library/en-us/odbc/htm/dasdkodbcoverview.asp, (accessed November 2002).

[Net02] NetDirect's JDataConnect, [WWW document], http://www.j-netdirect.com/, (accessed November 2002).

[Nor00] North, C., "A User Interface for Coordinating Visualizations Based on Relational Schemata: Snap-Together Visualization", *University of Maryland, Computer Science Dept.*, Doctoral Dissertation, (2000).

[Nor01] North, C., "Multiple Views and Tight Coupling in Visualization: A Language, Taxonomy, and System", *Proc. CSREA CISST 2001 Workshop of Fundamental Issues in Visualization*, pp. 626-632, (2001).

[NCI02] North, C., Conklin, N., Indukuri, K., Saini, V., "Visualization Schemas and a Web-based Architecture for Custom Multiple-View Visualization of Multiple-Table Databases", *Information Visualization*, Palgrave-Macmillan,1(3), (2002).

[NCI03] North, C., Conklin, N., Indukuri, K., Saini, V., "Fusion: Interactive coordination of diverse data, visualizations, and mining algorithms", *Accepted to ACM CHI 2003*, (2003).

[NCS02] North, C., Conklin, N., Saini, V., "Visualization Schemas for Flexible Information Visualization", *Proc. IEEE InfoVis 2002*, pp. 15-22, (2002).

[NS01] North, C., Shneiderman, B., "Component-Based, User-Constructed, Multiple-View Visualization", *Proc. ACM SIGCHI 2001*, pp. 201-202, (2001).

[NS00a] North, C., Shneiderman, B., "Snap-Together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata", *Proc. ACM Advanced Visual Interfaces 2000*, pp. 128-135, (2000).

[NS00b] North, C., Shneiderman, B., "Snap-Together Visualization: Can users construct and operate coordinated visualizations?", *International Journal of Human-Computer Studies*, pp 715-739, (2000).

[O02] OpenDX Home Page, [WWW document], http://www.opendx.org/, (accessed November 2002).

[PP01] Pattison, T., Phillips, M., "View Coordination Architecture for Information Visualization", *Proc. Australian Symposium on Information Visualization*, (2001).

[PCS95] Plaisant, C., Carr, D., Shneiderman, B., "Image browsers: taxonomy, guidelines, and informal specifications", *IEEE Software*, 12(2), pp. 21-32, (March 1995).

[PIO02] Platform Independent ODBC, [WWW document], http://www.iodbc.org/, (accessed November 2002).

[PCN02] Prabhakar, S., Conklin, N., North, C., Thirunavukkarasu, M., Dandapani, A., and Panchanathan, G., "Breakdown Visualization: Multiple Foci Polyarchies of Tuples and Attributes", *Proceedings of ACM CHI*, March 2002.

[Rad99] Radestock, M., "Coordination in Adaptive Open Distributed Systems", *Imperial College of Science, Technology, and Medicine, Dept. of Computing*, Thesis, (1999).

[RCK97] Rodrigues, L., *The Awesome Power of Java Beans*, Manning, (1998).

[RCK97] Roth, S., Chuah, M., Kerpedjiev, S., Kolojejchick, J., Lucas, P., "Towards an Information Visualization Workspace: Combining Multiple Means of Expression", *Human-Computer Interaction Journal*, 12(1&2), pp. 131-185, (1997).

[RJH02] Rymon-Lipinski, B., Jansen, T., Hanssen, N., Lievin, M., and Keeve, E., "Interactive Poster: A Software Framework for Medical Visualization", *Posters Compendium, IEEE InfoVis 2002 Symposium*, pg. 100-101, (2002).

[SML97] Schroeder, W., Martin, K., Lorenson, B., *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 2nd ed., Prentice Hall, (1997).

[SG96] Shaw, M., Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, (1996).

[Shn92] Shneiderman, B., "Tree visualization with treemaps: a 2-d space-filling approach", *ACM Transactions on Graphics*, 11(1), pp. 92-99, (1992).

[Shn01] Shneiderman, B., “Supporting Creativity with Advanced Information-Abundant User Interfaces”, *Human-Centered Computing, Online Communities, and Virtual Environments*, Springer-Verlag, pp. 469-480, (2001).

[Shn02] Shneiderman, B., “Inventing discovery tools: combining information visualization with data mining”, *Information Visualization*, Palgrave-Macmillan, 1(1), pp. 5-12, (2002).

[Sun02] Sun’s Java 2 SDK, Standard Edition Documentation, [WWW document], http://java.sun.com/j2se/1.4.1/docs/index.html, (accessed November 2002).

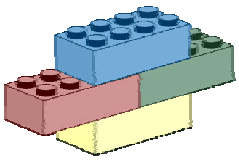
[TG02] Takatsuka, M. and Gahegan, M., “GeoVISTA Studio: A Codeless Visual Programming Environment For Geoscientific Data Analysis and Visualization”, *Journal of Computers and Geosciences*, (2002).

[u02] unixODBC, [WWW document], http://www.unixodbc.org/, (accessed November 2002).

[UFK89] Upson, C., Faulhauber, T., Kamins, D., Laidlaw, D., et. al., “The Application Visualization System: A Computational Environment for Scientific Visualization”, *IEEE Computer Graphics and Applications*, pp. 30-42, (1989).

[War97] Ward, M. “Creating and Manupulating N-Dimensional Brushes”, *Worcester Polytechnic Institute Computer Science Department*, (1997).

[WL02] Weaver, C., Livney, M., “Metavisualization of Dynamic Queries”, *Posters Compendium, IEEE InfoVis 2002 Symposium*, pg. 54-55, (2002).



Vita

Nathan James Conklin
nathan@conklinfamily.net
http://www.conklinfamily.net/

Master of Science, Computer Science, December 2002
Virginia Polytechnic Institute and State University, Blacksburg, VA

Master of Business Administration, December 2002
Virginia Polytechnic Institute and State University, Blacksburg, VA

Bachelor of Science, Computer Science, March 1999
Georgia Institute of Technology, Atlanta, GA