Operator-Centric Design Patterns for Information Visualization Software

Zaixian Xie, Zhenyu Guo, Matthew O. Ward, and Elke A. Rundensteiner

Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA, USA

ABSTRACT

Design patterns have proven to be a useful means to make the process of designing, developing, and reusing software systems more efficient. In the area of information visualization, researchers have proposed design patterns for different functional components of the visualization pipeline. Since many visualization techniques need to display derived data as well as raw data, the data transformation stage is very important in the pipeline, yet existing design patterns are, in general, not sufficient to implement these data transformation techniques. In this paper, we propose two design patterns, operator-centric transformation and data modifier, to facilitate the design of data transformations for information visualization systems. The key idea is to use operators to describe the data derivation and introduce data modifiers to represent the derived data. We also show that many interaction techniques can be regarded as operators as defined here, thus these two design patterns could support a wide range of visualization techniques. In addition, we describe a third design pattern, modifier-based visual mapping, that can generate visual abstraction via linking data modifiers to visual attributes. We also present a framework based on these three design patterns that supports coordinated multiple views. Several examples of multivariate visualizations are discussed to show that our design patterns and framework can improve the reusability and extensibility of information visualization systems. Finally, we explain how we have ported an existing visualization tool (XmdvTool) from its old data-centric structure to a new structure based on the above design patterns and framework.

Keywords: Design patterns, framework, information visualization, data transformation

1. INTRODUCTION

Over the past 15 years there has been increased attention on the use of information visualization techniques as a mechanism for understanding and exploring large and complex data sets. Researchers in the information visualization area have developed many visualization techniques to aid users in analyzing data, which has resulted in a growing number of academic and commercial software systems.

From the viewpoint of software engineering, a successful software system needs a good design in order to ensure straightforward implementation and maintenance; this is obviously the case with information visualization software. Designers can expect many common issues among different software systems, especially within the same domain. For example, all information visualization techniques need a mapping from data values to visual attributes, and thus it is worthwhile for information visualization software developers to reuse or borrow ideas from other systems. Researchers in software engineering proposed a term, *design patterns*,[1] to describe general solutions for recurring problems. The term design pattern can be considered at different granularities, from the high level architecture to the internal structure of a module. Many design patterns proposed by software developers are represented in an object-oriented style. As stated by Gamma et al.,[1] software design patterns are "descriptions of communicating objects and classes that are customized to solve design problems within a particular context."

Some design patterns have been proposed in information visualization. Stolte et al. described several design patterns for zooming within multi-scale visualizations.[2] Giereth and Ertl presented three design patterns for the rapid prototyping of information visualization applications .[3] These patterns can help create a visualization system configured by scripts, in which users can dynamically change settings of visual mappings. Heer et al. presented twelve design patterns[4] based on an analysis of Prefuse, an information visualization toolkit. These design patterns covered many issues for information visualization software, including data representation, derived data, interaction, and visual encoding. Hong proposed a new

Further author information: (Send correspondence to Zaixian Xie)
Zaixian Xie: E-mail: xiezx@cs.wpi.edu
Project Homepage: http://davis.wpi.edu/˜xmdv

concept: visualization design patterns.[5] They are not real design patterns, as software developers cannot directly use them to design software. Instead "they are used by users of visualization systems to model, design, and perform visualization tasks."

However, it is our opinion that currently existing design patterns are not sufficient to support the design of data transformations, which are an important stage in a visualization pipeline, since many visualization techniques display not only the raw data but also derived data. For example, Heer's *Cascaded Table*[4] supports a derived table via a subclass (derived data) inheriting from the base class (the original data). This design pattern is clear and easy to use, but we have found it does not handle complex requirements well and is difficult to extend. First, we have to change the interface of a cascaded table when we need more derived information. Second, this pattern is only suitable for a single step data transformation, which is not sufficient in many real applications. Finally, Heer's patterns focus on the data representation, and do not provide much guidance in the design of the data transformation. To overcome these shortcomings, we propose two design patterns especially for data transformation. The first one uses a vector of operators to represent a multi-step data transformation, namely the *operator-centric transformation* pattern. The second is the *data modifier* pattern, whose key idea is to attach a vector of modifiers to a data to describe derived data. In addition, we propose a third design pattern, *modifier-based visual mapping*, to provide a general solution for the generation of visual abstraction via linking modifiers to visual attributes.

The main contributions of this paper are as follows:

- We propose two operator-centric design patterns, operator-centric transformation and data modifier, to facilitate the design of information visualization software, focusing on the data transformation stage. We provide a categorization of such operators and modifiers.
- We propose a third design pattern, which is a general solution to do visual mapping via linking modifiers to visual attributes.
- We present a visualization framework based on the above design patterns. Since this framework can manage multiple pipelines, and different pipelines may share input datasets and operators, it can easily support coordinated multiple views.
- We describe several examples to show that these design patterns and the framework can improve the flexibility and extensibility of visualization software.
- We explain how we use the proposed design patterns and framework as we rewrite XmdvTool,[6] a public-domain software package for the interactive visual exploration of multivariate data sets.

In comparing our work with that of Heer and Agrawala,[4] although we propose fewer design patterns and cover fewer issues, our patterns are interrelated and can be easily assembled to generate a complete system. Most of design patterns proposed by Herr and Agrawala are separate from each other, thus a developer must put significant effort to combine them within one system. For example, they proposed a design pattern, *Operator*, to describe visual encoding, and other ones for data representation, but did not discuss how to link data with visual attributes. Our proposed design patterns can overcome this shortcoming and enable developers to more easily create a whole system.

2. THREE DESIGN PATTERNS

2.1 Operator-centric Transformation

This design pattern is the core of this paper. Its purpose is to help developers easily construct a module to support multi-step data transformation via a set of operators having a uniform interface. It includes three basic classes, *Transformation*, *Operator*, and *Data* (See Figure 1). The main body of *Transformation* is composed of a vector of *Operators*. Each operator represents a single step transformation, such as sampling, sorting and clustering. The input and output of an operator are both instances of the *Data* class. The function *doOperation()* in *Operator* is responsible for the conversion from input to output. In order to support different types of operators and data, the actual data types and operators instantiate the subclasses of *Data* (Figure 1(b)) and *Operator* (Figure 1(c)). Note that all subclasses of operator should override the function *doOperation()*, in which we can define the specific behavior of data derivation. Based on the above description, the function *transform* can be implemented via the pseudocode shown in Algorithm 1.

Algorithm 1 Doing transformation via an operator sequence

1: Data* result ← the raw data;
2: **for** $i = 0$ to opList.size()-1 **do**
3: opList[i].setInput(result) ;
4: opList[i].doOperation() ;
5: result ← opList[i].getOutput() ;
6: **end for**
7: **return** result

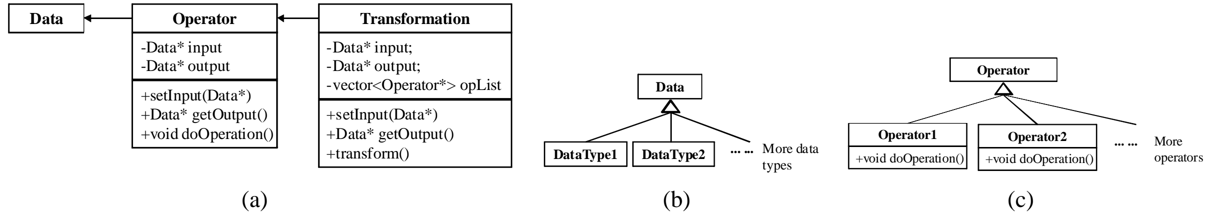

Figure 1. The main body of operator-centric transformation. It is composed of three classes, transformation, operator and data, as shown in (a). *Operator* and *data* both can have multiple subclasses to represent the actual operators and data types.

Although this design pattern is for data transformation, it also can be employed for the implementation of many interaction techniques. For example, *brushing* is a commonly used interaction technique for selecting a subset of data for highlighting, masking, deletion, and other tasks. Based on our design patterns, we can implement brushing using an operator, namely *BrushOp*, and visual mapping. *BrushOp* can tag a subset based on the brush definition, and visual mapping can then be used to highlight the data items in this subset.

Our operators are inspired by the framework proposed by Chi and Riedl.[7] The operators in Chi and Riedl's framework not only exist in data transformations, but also work for visualizations and visual mapping transformations. Their framework aims to give a uniform representation for different stages of the visualization pipeline via operators and states. We, however, have a different goal, which is to provide patterns for the design of operators especially in the data transformation stage. Heer and Agrawala also proposed a design pattern, namely *Operator*,[4] that has the same name as ours. However, their operators are used to describe visual encodings rather than data transformations.

2.2 Data Modifier

Communicating classes in the design pattern, *Data Modifier*, are shown in Figure 2. The purpose of this pattern is to provide a flexible data structure to represent the derived data. The original data is denoted by the class, *Data*. In order to represent the derived information for the original data, we introduce a class, *DataModifier*, which is the key idea of this design pattern. Since one *Data* can have more than one type of derived data, we use a class, *DataModifierManager*, to manage a vector of *modifiers*. Different types of data modifiers are described by subclasses of *DataModifier*. The biggest advantage of this design pattern is its excellent extensibility. When we want to add more derived data to extend the existing system, we only need to add a new class inheriting from *DataModifier*, instead of change the existing data structure.

An example is *SamplingModifier*, which represents the data transformation result of a sampling operator. As we know, sampling is often used in visualization to pick a subset of the original data to display, as a means for reducing visual clutter. Assume that the original data is a set $\{D_1, D_2, ..., D_n\}$. The sampling result (*SamplingModifier*) can be represented by a vector $(a_1, a_2, ..., a_n)$. $a_i (1 \leq i \leq n)$ can only be 0 or 1. D_i is in the displayed subset if and only if $a_i = 1$. Recall the operator *BrushOp* discussed in Section 2.1. The output of this operator also can be represented by such a vector $(a_1, a_2, ..., a_n)$ as a modifier to denote which data items to select.

Another issue is how to design the function members in class *Data* to reflect the existence of data modifiers. For example, the result of a function getData(LineNo), which serves the rendering class by returning a single item in the displayed subset, is impacted by a *SamplingModifier*. To solve this problem, we let *DataModifierManager* work as an agent to manage data modifiers and provide appropriate derived data to other objects such as visual mapping and rendering. All of the requests to access data in the class *Data* will be first sent to this agent class, and then *DataModifierManager* will seek an appropriate modifier to handle the request.

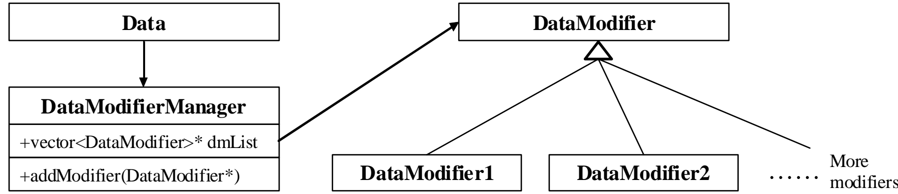

Figure 2. Data Modifier: A vector of *DataModifier* is managed by *DataModifierManager*. Each *DataModifier* describes its own derived information for the original data.

2.3 The Categorization of Operators and Modifiers and Further Discussion

Categorization of Operators: We notice that there are two types of operators, namely *modifier operators* and *creation operators*. The former only attaches a modifier to the original data, e.g., the sampling operator, while the latter will create a totally new data set. For example, a hierarchical clustering operator can be regarded as a creation operator because its output, a cluster tree, is a new data type totally different from the original data. Note that an operator's type depends on how the developer designs the system. For example, we can define a cluster tree as a modifier of the original data, thus the clustering operator will be a modifier instead of a creator, although this might make the design and reuse of the software more difficult.

As shown in Figure 1(a), *Operator* has a function *doOperation()*. This function is responsible for performing data transformations, which is the main behavior of *Operator*. For different types of operators, their *doOperation()* functions will be significantly different from each other. Figure 3 shows the detailed semantics of the *modifier operator* (Figure 3(a)) and *creation operator* (Figure 3(b)). As shown in Figure 3(a), the input of the *modifier operator* probably has been attached some modifiers (from modifier-1 to modifier-k in this case), thus the output has to contain these existing modifiers as well as a new modifier. In Figure 3(b), *Data1* and *Data2* may instantiate from the same or different classes, which both inherit from the base class *Data*. For example, the input and output of an operator *ClusterOp* are a multivariate dataset and a cluster tree, respectively. However, an operator to perform multidimensional scaling (MDS) can apply an algorithm on a high-dimensional data set and produce a lower dimensional data set. Both input and output can be the instances of a data type representing multivariate data.

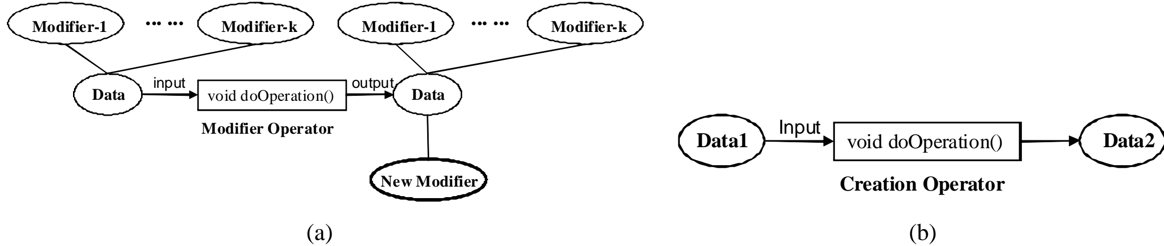

Figure 3. Different behaviors for two types of operators: (a) modifier operator; (b) creation operator.

Categorization of Data Modifiers: Data modifiers can be categorized into two types: *view modifiers* and *attribute modifiers*. The first one only provides a view, namely mapping from the original dataset to the derived data, to the objects using the data, e.g., rendering. It does not add additional data to the original dataset. For multivariate data we can identify two types of mapping, one on records(rows) and the other on dimensions(columns). For instance, sampling modifiers represent mappings on records, and dimension on/off/reordering modifiers correspond to the mapping on dimensions. Compared to view modifiers, attribute modifiers contain new data as the attributes of the original data. One example of such a modifier, namely *HighlightModifier*, represents the output of the operator *BrushOp* via a vector $(a_1, a_2, ..., a_n)$ (see Section 2.2). Note that n is the number of data items in the original dataset. The ith data item is in the highlighted subset if and only if $a_i = 1$. Obviously, this modifier is similar to the sampling modifier in the form of representation, but they have different semantics.

With the above categorization, we add more details to the data modifier design pattern to reflect behaviors for different types of modifiers (Figure 4). In *AttributeModifier*, the main function members aim to provide an access to the attributes.

For multivariate data, three functions, *getAttr(int rec, int dim)*, *getRecAttr(int rec)*, and *getDimAttr(int dim)*, can return the attribute values for a specified value, record or dimension. For other types of data, e.g., 3D spatial data, software designers can develop other functions using the same style. Let us recall *HighlightModifier*. In this modifier, the function *getRecAttr(i)* can return a boolean value to represent whether the data item in the record i will be highlighted in the final display. Regarding *ViewModifier*, we define two subclasses, *RecViewModifier* and *DimViewModifier*, corresponding to the mapping on records and dimensions. They both provide two functions, *map* and *invmap*. The function *map(i)* returns the position in the view for a specified record or dimension whose index is i in the original dataset. The other function, *invmap*, does the inverse mapping. It can find the index for a record or dimension whose position in a view is known. The inverse mapping is very useful for those objects that need to access data. For example, the visual mapping and rendering class only show the data in the final view, but they need to get the physical data to determine attributes, such as the position, length, and size, of visual elements. Thus they need the inverse mapping (*invmap*) to create the linkage between views and the original data.

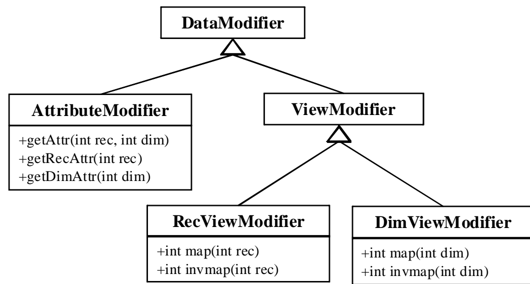

Figure 4. The design pattern, *Data Modifier*, with more details.

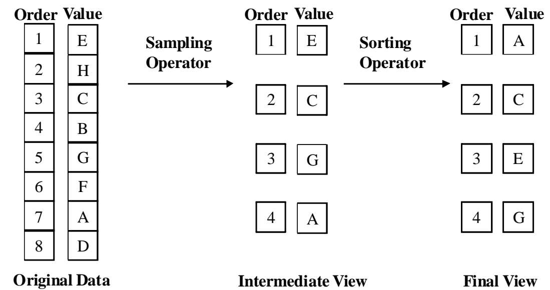

Figure 5. An example of view modifiers, two of which are applied to one dataset in a specified order.

Given these two types of modifier, we define the following rules to avoid possible conflicts among different modifiers: (1) In all attribute modifiers, attributes have the same order as the original data. Thus, if other objects want to access attribute values and they only know its position in the view, they need to use view modifier to get the physical position of the requested attribute value. (2) If one dataset has more than one view modifier, a fixed order of these modifiers should be predefined. Any mapping should be applied to the dataset in this order.

To clarify the second rule, we show an example in Figure 5, in which a sampling operator (with sampling rates=50%) and then a sorting operator are applied to a dataset having 8 data items. We list the return values of the functions *map(i)* and *invmap(i)* of these two operators in Table 1. Note that *map(i)*=0 means this data item does not exist in the view. Assuming we want to know the position of 'G' in the final view starting from the original data, we can use $map_2(map_1(5)) = map_2(3) = 4$ to get it. In the reverse order, if we start from the final view, and want to get the index of 'E' in the original data, we can calculate this using $invmap_1(invmap_2(3)) = invmap_1(1) = 1$. In general, if we have n view modifiers, the mapping from the original data to the final view should use $map_n(map_{n-1}(...map_1(i)...))$ and the mapping in the other direction can be performed via $invmap_1(invmap_2(...invmap_n(i)...))$.

Sampling Modifier				Sorting Modifier			
i	$map_1(i)$	i	$invmap_1(i)$	i	$map_2(i)$	i	$invmap_2(i)$
1	1	1	1	1	3	1	4
2	0	2	3	2	2	2	2
3	2	3	5	3	4	3	1
4	0	4	7	4	1	4	3
5	3	-	-	-	-	-	-
6	0	-	-	-	-	-	-
7	4	-	-	-	-	-	-
8	0	-	-	-	-	-	-

Table 1. The return values for the functions *map(i)* and *invmap(i)* in two operators shown in Figure 5.

If we explore how operators derive data, we can find an obvious efficiency issue. As shown in Figure 3(a), for a *modifier operator*, its output should include all of the raw data and modifiers in the input as well as the new modifier.

Assume that there are n operators in the pipeline, the last operator needs to copy $n-1$ modifiers. Since some operators include vectors whose sizes are comparable to the raw data, this will cause a significant time cost if we do the copy item by item. However, in a traditional framework for information visualization, one component is responsible for a type of data derivation and has its own data structure to represent the derived data, thus such a copy is not necessary. In order to avoid this possible reduction on performance while using our design patterns, we can make each *modifier operator* only copy the reference of raw data and modifiers. This is an operation with constant time complexity in most of programming languages. For example, in C++, the reference can be represented via pointers, thus the copy of the reference is only an assignment operation.

2.4 Modifier-based Visual Mapping

In multivariate visualization, one data item will normally be visualized by one or a set of visual items. For example, in a scatterplot matrix, one tuple corresponds to N^2 points, where N is the number of dimensions. In the visual mapping (or visual abstraction) stage of multivariate visualization, visual attributes of these visual items can be determined by dimension values or attributes. For instance, *HighlightModifier* representing the result of brushing, can determine the colors of visual items to denote whether they are highlighted or not. Thus we propose a design pattern for visual mapping as shown in Figure 6. The key idea is to associate a data modifier with visual attributes. Although this design pattern is for multivariate data, it is easy to extend it to other data types.

The core class of this design pattern is *VisualMap*. It maintains two maps, *visModi* and *visData*, which associate an arbitrary visual attribute with a specific attribute modifier or dimension values, respectively. The developer can use functions, *register(AttributeModifier*, VisAttr*)* and *register(int dim, VisAttr*)*, to create both maps. The function *doVisualMap()* generates an instance of class *VisualMapResult*. This result contains a vector of instances of class *VisAttrList*. The size of this vector is normally the number of displayed data items. Each instance represents the visual attributes of one data item. Since one visual item can have multiple attributes, such as color, size and shape, the class *VisualAttrList* is an aggregation of class *VisualAttr*, which represents a single visual attribute. Some visualization techniques do not fit the above description very well. For example, we can color each segment in parallel coordinates by the value on the corresponding dimension. It should be easy to implement this based on the above design pattern with only minor changes.

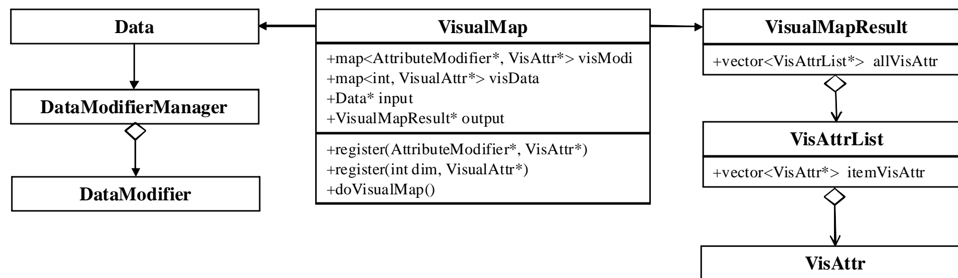

Figure 6. The design pattern for visual mapping based on the data represented by data modifier.

2.5 Extensibility of Proposed Design Patterns

Compared to other design patterns and frameworks, the biggest advantage of the operator-centric design patterns is that they make the visualization system easy to extend. The main reason is a uniform interface of operators. This can be further explained by our experience in the design of XmdvTool. In the currently released version of XmdvTool (7.0), we used a data-centric structure. We added each new feature as a separate component that directly manipulated the existing data structures. Thus we had to handle potential conflicts among different components, which resulted in many bugs and slower development time. Sometimes it was difficult to make two components work together, and we had to disable one component when users activated the other component. For example, the current released version of XmdvTool does not allow users to hide some of the dimensions while using the structure-based brush.[8] Another example is the conflict between the multidimensional brush and dimension on/off. Because we introduced dimension on/off after implementing the brushing feature, we had to rewrite a significant amount of code to resolve the conflicts between these two components.

The above problems (and others) we faced were the main reasons to push us toward the development of operator-centric design patterns for use in redesigning XmdvTool. The design patterns force all operators to have a uniform interface, so the impact on one operator from other operators is extremely limited. Now, as we redesign XmdvTool using the proposed design patterns. the multidimensional brush, structure-based brush, dimension on/off, and many other components are all represented by operators. The co-existence of different components can be easily enabled if the semantics are valid.

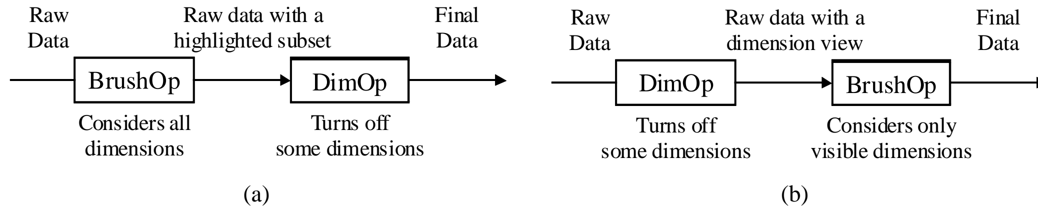

Figure 7. Different brush semantics generated by reordering the operators. (a) All dimensions are considered (as in XmdvTool); (b) Only visual dimensions are involved in the brush definition (as in Tableau).

We show another example to explain how our proposed design patterns help us to easily obtain new features via changing the combination and sequence of operators. This is in regards to the multidimensional brush in XmdvTool and Tableau,[9] both of which use dimension ranges as the parameters in brush definition. Only those data items falling into dimension ranges are highlighted. Because users often turn off some dimensions to focus on those attributes of interest, one problem is whether we need to consider all dimensions or only those visible dimensions when we judge whether one data item should be highlighted. XmdvTool, in its current framework, always considers all dimensions, while Tableau only uses visible dimensions to perform this task. In our new version (soon to be released), we use an operator to do brushing (*BrushOp*) and another to do dimension on/off/reordering (*DimOp*). If we put *BrushOp* before *DimOp*, we are doing the same thing as the current XmdvTool (Figure 7(a)). If we reverse the sequence, the behavior of the system is the same as Tableau (Figure 7(b)).

3. THE FRAMEWORK

Based on the design patterns described in Section 2, we propose a framework for information visualization as shown in Figure 8. This framework can contain multiple pipelines. Each pipeline is composed of three stages: data transformation, visual mapping, and rendering. Different pipelines can share operators. For the convenience of design, we provide an operator pool that contains operator instances used in the whole system. When each pipeline is created, operator instances in the operator pool are requested and added into the pipeline. For most interaction techniques, arguments in operators or the visual mapping stage need to be changed to reflect users' requests. Thus we link interactions to the operators or visual mapping stage in the pipeline. For example, when a user changes the dimension ranges in a multidimensional brush, we only need to change the brush parameters in the *BrushOp* and then send the raw data to the pipeline again and repaint the canvas, resulting in a view with the new brush.

Since this framework uses the operator-centric design patterns, visualization software systems based on it will exhibit enhanced reusability and extensibility. Moreover, this framework can support the design and implementation of coordinated multiple views (CMV). Our approach to implementing CMV is as follows. We create multiple pipelines, each of which corresponds to one of the linked views. These pipelines share some operators and/or visual mappings. A typical style of sharing is a fan out solution, as shown in Figure 9. When a user performs interactions within one view, changes in the parameters of shared operators are distributed among all the linked pipelines and the views are updated.

4. USING THE DESIGN PATTERNS AND FRAMEWORK IN XMDVTOOL

In this section, we briefly explain how we use the presented design patterns while redesigning XmdvTool, a public domain multivariate data visualization package developed at WPI. Some of the operators we have introduced into XmdvTool are listed in Table 2. The descriptions of these operators are as follows. Note that if we do not list the input of one operator, it means the input is a multivariate dataset by default.

FlatBrushOp

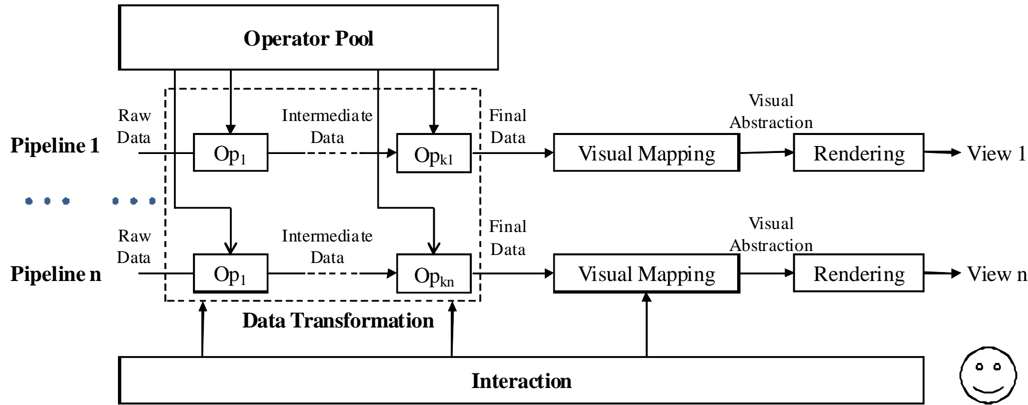

Figure 8. The framework based on the design patterns proposed in this paper. It can contain multiple pipelines that share operators from an operator pool. Interaction is associated with operators and visual mappings.

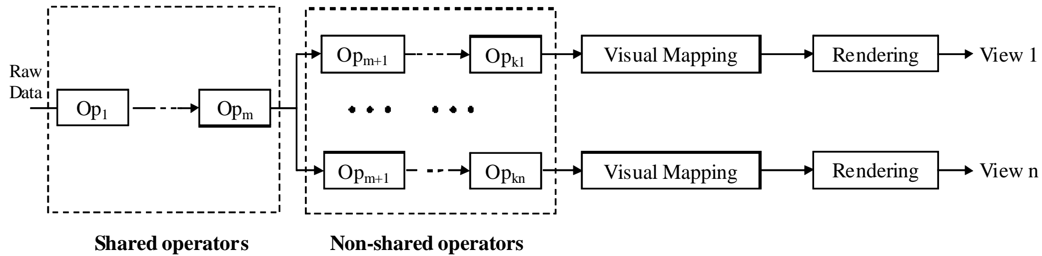

Figure 9. A fan out solution for the design of coordinated multiple views based on our proposed framework.

Output: A modifier that contains a bit array to denote which data items are highlighted in the final display.

Operation: Select a subset whose datapoints fall into specified dimension ranges.

SamplingOp

Output: A modifier to map from the original data to a sampled subset.

Operation: Apply a uniform sampling to the original dataset. Other sampling methods are easily added.

DimOp

Output: A modifier to map from the original dimension order to a new one.

Operation: Reorder the dimensions or disable/hide a subset of them based on either user interactions or a heuristic algorithm that reorders dimensions to reduce visual clutter.[10]

SortOp

Output: A modifier to map from the original dataset to a new one with records ordered based on the values in a specified dimension.

Operation: Sort datapoints in terms of values on a specified dimension.

ClusterOp

Modifier Operator	FlatBrushOp, SamplingOp, DimOp, SortOp
Creation Operator	ClusterOp, ClusterLODViewOp, DimTreeOp, DimReductViewOp

Table 2. Some important operators introduced into XmdvTool to port from a data-centric structure to the current operator-centric framework.

Output: A hierarchical cluster tree. Each leaf is a data item in the original dataset. Similar data items compose a cluster, and in turn similar clusters form a higher level cluster, until the entire dataset is represented by a single cluster.[8]

Operation: Perform hierarchical clustering on the input.

ClusterLODViewOp

Input: A hierarchical cluster tree.

Output: A new multivariate dataset which is an abstraction of the original dataset.

Operation: Select a subset of the data hierarchy to view. This operator is associated with an interaction interface, namely the structure-based brush, as shown in Figure 10.[8] The tree shape is approximated by its leaf contour (see (c)). The colored bold contour (see (b)) represents the current selected level-of-detail. The interactive brush handles (see (e)) determine a range based on a pre-defined attribute value. The user can drag the colored bold contour (see (b)) to change the LOD parameters, and adjust the interactive brush handles to define a new range. This operator can generate an abstraction of the original dataset in the form of a multivariate dataset that has the same number of dimensions as the original dataset but with fewer data items (one per cluster). This abstraction is visualized as shown in figure 12. The detailed abstraction generation steps are as follows: (1) Retrieve all nodes on the selected level of detail (see (b)). (2) Map each node to a data item with dimension values being the mean values of data in this cluster. (3) Organize these data items into a new multivariate data set. (4) Attach a modifier to represent the colors of clusters. Their colors are determined by the order of clusters (see (f)) if they are out of the range defined by the brush handles (see (e)), or are bold red if they are within the range. (5) Add a modifier to represent the cluster size for each data item, which is denoted by the band width in the final visualization (Figure 12).

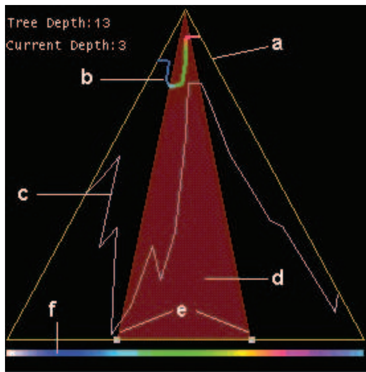

Figure 10. Structure-based brushing tool. (a) The tree frame; (b) Contour corresponding to the current level-of-detail; (c) Leaf contour approximates shape of the tree; (d) Structure-based brush; (e) Interactive brush handles; (f) Colormap legend for level-of-detail contour.

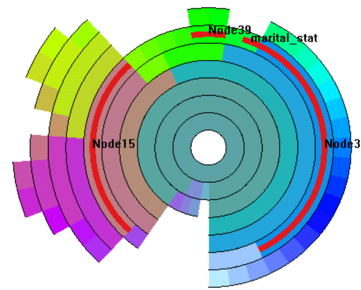

Figure 11. An InterRing display to allow users to select nodes on a dimension cluster tree, which is generated from a census income dataset (42 dimensions). Note that the user has selected 4 dimension clusters, one of which actually is an original dimension.[11]

DimTreeOp

Output: A dimension cluster tree. Tree leaves denote the original dimensions. Similar dimensions are put into clusters, and similar clusters in turn will be put into clusters at a higher level.[11] In order to facilitate the design of the next operator, *DimReductViewOp*, we attach the original dataset to this dimension hierarchy.

Operation: Organize a dimension cluster tree to represent the similarity among the original dimensions.

DimReductViewOp

Input: A dimension cluster tree.

Output: A new multivariate dataset adapted from the original dataset but with fewer dimensions.

Operation: This operator aims to generate a new dataset in a lower dimensional space, which is useful for exploring a dataset that has a large number of dimensions. We link this operator to an interface for dimension reduction, namely

InterRing[11] as shown in Figure 11, in which users can select nodes in the dimension cluster tree. This operator projects the original dataset to a lower dimensional space containing only those selected clusters as dimensions. A specific dimension value in the new dataset can be from a user-selected or random dimension in the cluster, or the first principal component after applying Principal Component Analysis (PCA) to all the dimensions in the cluster.[11] As an example, in Figure 11, the user chooses 4 clusters, and this operator generates a new dataset having only 4 dimensions; this is then visualized via parallel coordinates as shown in Figure 13.

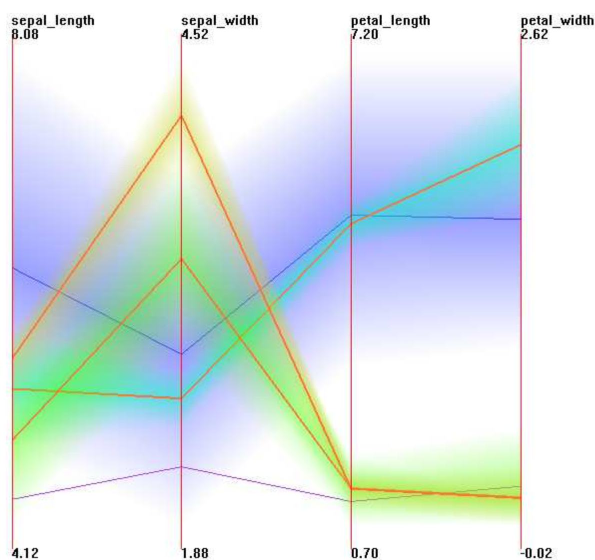

Figure 12. A hierarchical parallel coordinates display of the *Iris* dataset. It shows five clusters on the selected level-of-detail (Figure 10). The bold red color means that the cluster is currently being selected by the structure-based brush. The line color denotes the cluster order, except the brushed clusters, and the band widths represent sizes of the clusters.[12]

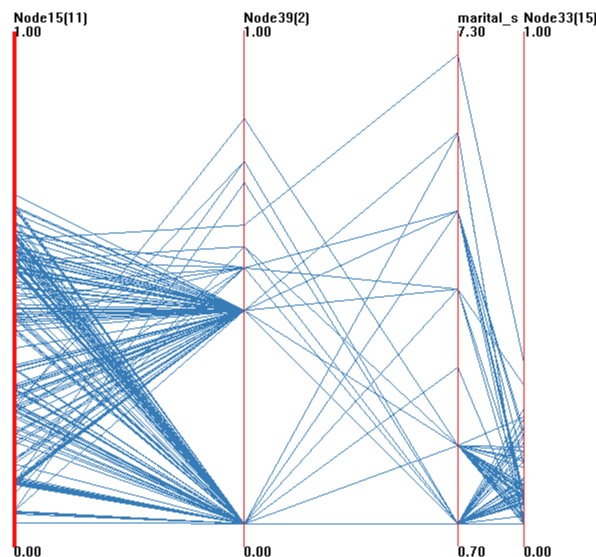

Figure 13. This parallel coordinates display shows a multivariate dataset by projecting the original dataset to a lower dimensional space. It contains the dimensions selected in Figure 11. Note that the dimension name “Node15(11)” means that the dimension “Node15” is a dimension cluster having 11 descendent leaf nodes corresponding to original dimensions. Axis width conveys the variability within the dimension clusters.[11]

When we ported XmdvTool from the old version to the new framework, we defined different modes, each of which corresponds to a pipeline configuration. Figure 14 shows the data transformation stage for each mode. We briefly describe each mode:

Flat Mode: As shown in Figure 14(a), the core of this mode is a *FlatBrushOp* representing a multidimensional brush. This model also supports sampling using *SamplingOp*, sorting via *SortOp*, and dimension on/off/reordering by *DimOp*. Note that we can have different brush semantics if we exchange the position of *FlatBrushOp* and *DimOp* as shown in Section 2.5. In the future, we can add more operators to do more complex transformations, such as principal component analysis[13] and multidimensional scaling.[14] This is the basic mode, suitable for exploring small datasets with a modest number of dimensions.

Structure-based Brush Mode: This mode is used to explore datasets whose number of records is very large; this is done by displaying abstractions of the original dataset to reduce visual clutter (Figure 14(b)). It is derived from the flat model by adding two operators, *ClusterOp* and *ClusterLODViewOp*. First, *ClusterOp* is applied to the input dataset to create a cluster tree; then *ClusterLODViewOp* provides users with a structure-based brush to choose a specific level-of-detail and region of interest. A dataset containing all nodes on the selected level will be generated as the output of operator. Finally this new dataset will pass through other operators in the flat model and be visualized. This mode is much more powerful than the hierarchical display in the data-centric structure because we can easily apply multidimensional brush and dimension on/off/reordering operations to the structure-based brush results.

Dimension Reduction Mode: In this mode (Figure 14(c)), operators *DimTreeOp* and *DimReductViewOp* support the interactions needed for dimension reduction. The operator *DimTreeOp* can generate a dimension hierarchy, and then users

can use the InterRing display associated with *DimReductViewOp* to select dimension clusters for exploration. *DimReductViewOp* can project the original dataset to a lower dimensional space containing those selected dimension clusters. The projection result will go to those operators contained in the flat model and be displayed via multivariate visualizations. Similar to structure-based brush mode, this mode enables us to easily do multidimensional brushing and dimension on/off/reordering on the dimension reduction result, which would have been difficult to implement in the old data-centric structure, because of conflicts among different components.

In addition to the above list, some other modes will be implemented in our future work, such as a combination of structure-based brush mode and dimension reduction mode, which will enable us to further explore datasets having both large numbers of records and dimensions. Because of the extensibility of our design patterns, this will be easy to design and develop.

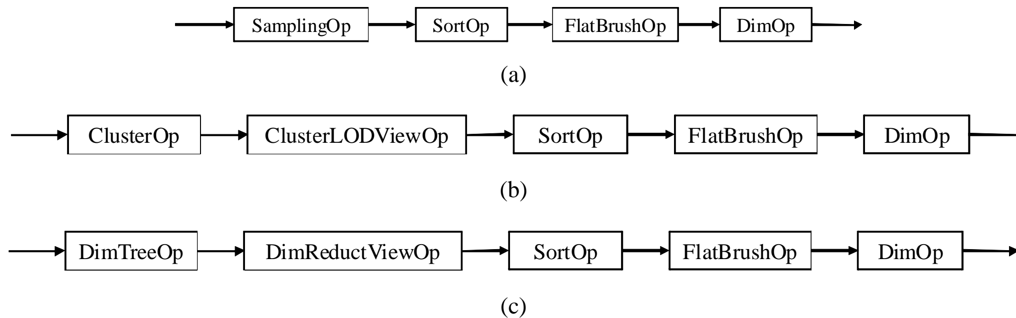

Figure 14. Three modes in XmdvTool: (a) Flat mode; (b) Structure-based brush mode; (c) Dimension reduction mode.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we presented operator-centric design patterns for developing information visualization software. The key idea is to use an operator class to represent a single step data transformation; a complex multi-step data transformation can then be performed using a vector of such operators. We distinguished two types of operators, creation operators and modifier operators. The former generates totally new data, while the latter adds a data modifier to the original data. A data modifier can describe a view or an attribute applied to the original data. The third design pattern, modifier-based visual mapping, can generate visual abstractions via linking modifiers to visual attributes. Based on these three patterns, we proposed a framework to enable the creation of multiple pipelines within a system. We also showed that the proposed design patterns can significantly improve the reusability and extensibility of visualization software. Because these pipelines may share operators, this framework can easily support coordinated multiple views. We developed these design patterns and framework based on the assumption that the raw data consists of multivariate tables. However, we believe the patterns and framework could be readily adapted to other data types. Some potential future work includes:

- **A More General Definition for Operators:** An operator proposed in this paper has only one input and one output. However, more than one input is possible, e.g., doing join operations on two datasets. Thus it could be useful if we extended the current design patterns to enable operators to handle multiple input and output.
- **Dynamic Configuration of Data Transformations:** A visualization system could be more powerful and flexible if users were allowed to dynamically configure operator sequences, such as adding or removing operators and changing operator order interactively. A possible solution is to further investigate the semantics of operators and add limitations to the linkage among operators. This could help identify invalid sequences and accept only those with consistent semantics.

ACKNOWLEDGMENTS

This work is supported under NSF grants CCF-0811510 and IIS-0812027.

REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M., [*Design Patterns: Elements of Reusable Object-Oriented Software*], Addison-Wesley, Bloomington, MN, USA (1994).

[2] Stolte, C., Tang, D., and Hanrahan, P., "Multiscale visualization using data cubes," in [*Proc. IEEE Symposium on Information Visualization*], 1–8 (2002).

[3] Giereth, M. and Ertl, T., "Design patterns for rapid visualization prototyping," in [*International Conference on Information Visualisation*], 569–574 (2008).

[4] Heer, J. and Agrawala, M., "Software design patterns for information visualization," *IEEE Trans. Visualization and Computer Graphics* **12**(5), 853–860 (2006).

[5] Chen, H., "Towards design patterns for dynamic analytical data visualization," in [*Visualization and Data Analysis, Part of IS&T/SPIE Symposium on Electronic Imaging*], 75–86 (2004).

[6] "Xmdvtool home page." http://davis.wpi.edu/˜xmdv/.

[7] Chi, E. and Riedl, J., "An operator interaction framework for visualization systems," in [*Proc. IEEE Symposium on Information Visualization*], 63–70 (1998).

[8] Fua, Y., Ward, M., and Rundensteiner, E., "Structure-based brushes: A mechanism for navigating hierarchically organized data and information spaces," *IEEE Trans. Visualization and Computer Graphics* **6**(2), 150–159 (2000).

[9] "Tableau software — visual analysis and data visualization." http://www.tableausoftware.com/.

[10] Peng, W., Ward, M., and Rundensteiner, E., "Clutter reduction in multi-dimensional data visualization using dimension reordering," in [*Proc. IEEE Symposium on Information Visualization*], 89–96 (2004).

[11] Yang, J., Ward, M., Rundensteiner, E., and Huang, S., "Visual hierarchical dimension reduction for exploration of high dimensional datasets," in [*Joint Eurographics/IEEE TCVG Symposium on Visualization*], 19–28 (2003).

[12] Fua, Y., Ward, M., and Rundensteiner, E., "Hierarchical parallel coordinates for exploration of large datasets.," in [*Proc. IEEE Visualization*], 43–50 (1999).

[13] Jolliffe, J., [*Principal Component Analysis*], Springer Verlag (1986).

[14] Kruskal, J. and Wish, M., [*Multidimensional Scaling*], Sage Publications (1978).