Binding Design Patterns for Visualization Libraries

Yung-Pin Cheng

Dept. of Computer Science and Information Engineering
National Central University
300 Zhongda RD., ZhongLi City
TAIWAN

ypcheng@csie.ncu.edu.tw

Abstract. *Building a visualization library or visualization tool may face several design choices. These design choices may affect the reusability and flexibility of the system. The most essential problem is how to let users bind their data to the visualizations. In this paper, four binding design patterns are described, studied, and discussed. By defining these design patterns, we can help clarify the directions of design and implementation in building future visualization tools.*

Categories and Subject Descriptors

•Software and its engineering → Design patterns

•Human-centered computing → Visualization toolkits

General Terms

Design, Human Factors

Keywords

Visualization Library, Data Binding, Visualization Tools, Design Pattern

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. Preliminary versions of these papers were presented in a writers' workshop at the 5th Asian Conference on Pattern Languages of Programs (AsianPLoP). AsianPLoP'2016, February 24-26, Taipei, Taiwan. Copyright 2016 is held by the author(s). SEAT ISBN 978-986-82494-3-1.

1. Introduction

Visual perception is the most important sense of humans. In the areas of information visualization, software visualization, 3D visualizations, etc., visuals are the entities to communicate a message which can make sense of the information. Visuals can be images, diagrams, or animations that rendered by a piece of code. Let the piece of code be called "visualization code." Visualization code so far is often tied to a specific platform or a render library. A common platform is a GUI system, which provides a set of basic drawing APIs to allow application to draw graphics on GUI components. Another example is Scalable Vector Graphics (SVG). SVG is an XML-based vector image format for drawing two-dimensional graphics. It is very popular for drawing 2D graphics in web browsers.

Although there has been plenty of visualization code implemented in different applications, due to the variety of platforms and programming languages, programmers often find that they need to implement visualization code from scratch. Some reasons are: the aesthetic needs for visualization are often application specific; the visualization code is often tightly coupled with application-specific data types, making reuse of visualization code difficult.

Depending on the aesthetic needs of visualizations, visualization code can be large and complicated. A complicated visualization may include complex algorithms to layout the visuals or provide friendly interactions between the visuals and users. So, aiming for reducing the cost and efforts in developing visualization code, visualization libraries have been proposed continuously for different platforms and programming languages. Unfortunately, there are some fundamental issues to be studied, determined, and resolved when a visualization library is constructed. So far, no optimal solutions can be concluded. One major issue in reusing a visualization library is how to bind the data to the visuals by users. Several approaches have been developed in the existing visualization libraries. These approaches, unfortunately, can deeply affect the system architectures, flexibility, and extensibility of the libraries. In this paper, we describes these approaches into four binding design patterns. The pros and cons are studied, discussed, and compared.

2. Background and Related Work

Some common goals [9] pursued by visualization tools are: to separate data and visual models to enable multiple visualizations of a data source, to separate visual models from displays to enable multiple views of a visualization, and use modular controllers to handle user input in a flexible and reusable fashion. There are several separation of concerns to be taken care by visualization tools. Different design choices often can results in serious consequences with respect to complexity, extensibility, and reusability of visualization tools.

In [9], Heer describes a reference model pattern (see Fig. 1) which provides a general template for structuring visualization tools that address the separation of concerns. Since understanding this reference model is crucial to the understanding of this paper. The details of the reference model are explained here.

A DataSource component loads the data sets to be visualized. One or more data sets can then be registered with a visualization using an observer design pattern. In principle, DataSets should not be aware of the existences of visualization, so data is separated from visual attributes such as location, size, shape, and color. If there are more than one observer (visualization) registering to the DataSet, a single abstract data set can be used in multiple visualizations.

The visualization in this reference model is also called visual models. The Visual models, View, and Control classes are standard Model-View-Controller (MVC) pattern [11] in user interface development. A visual model can be shown in more views; that is, different displays in different platforms. Views accepts the user input and then often have user inputs processed by controls. Plenty of software frameworks adopt this template structures, including Advizor [6], Improvise [15], Polaris [13], and prefuse [10].

There are actually two observer design patterns applied in this reference model, one between DataSet and Visualizations and one between Visualizations and views. So, Heer in [9] describes the reference model as a two-tiered MVC. Here, we focus on the observer pattern between data set and visual models -- how data is bound to a visual model.

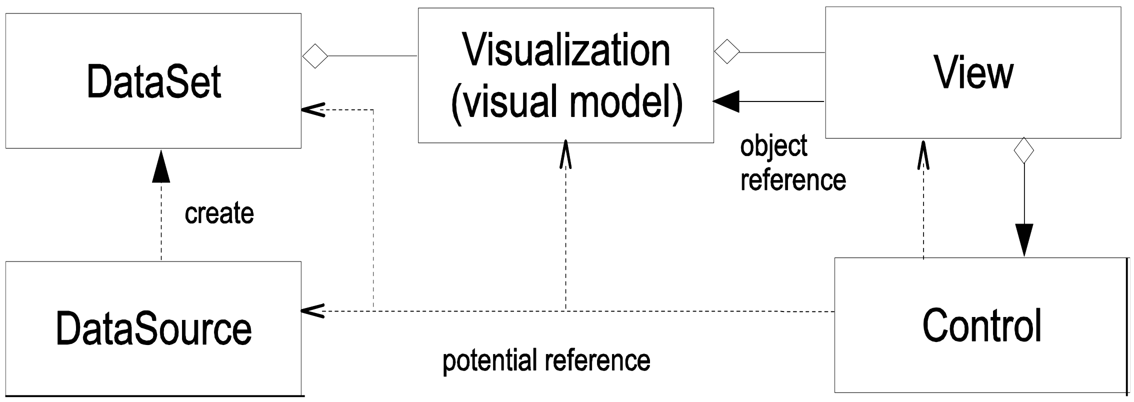


Figure 1. The visualization reference model

2.1 What is a visual model?

In a MVC, a model captures the behaviors of the application in terms of its problem domain, independent of user interfaces [12]. The model directly manages the data, logic and rules of the application. However, a model in visualization MVC has some significant differences than a typical MVC. That is why in Fig. 1, we annotate the visualization class as a visual model.

Fig. 2 shows some visualization screen-shot from D3 [9,2]. Behind each visualization, a visual model must be defined first. For example, visualizations such Population Pyramid or Stacked Bars assume data is organized in a form of table (multiple columns or rows). On the other hand, Force-Directed graph renders two major visuals - nodes and links, which assume data must be structured in a style of binary relation. So, if DataSet is not organized or structured in a way that is defined by a visual model, it is impossible to map the DataSet to a visual model. That is, a visual model actually poses constraints on data that can be fed from DataSet. On the other hand, the structure of a DataSet can actually be used to filter out unsuitable visual models.

So, when a visualization is implemented as a library, typically its programmer already has a visual model in his mind. A visual model typically contains some visual model entities. For example, a visual model for a graph visualization mainly deals with two major entities, node and link. A node has several properties, such as its shape, position, color, etc. Sometimes, the relation between a visual model entity and a data type are not as explicit as in the graph example. For example, a bar chart visualization typically assume data to be stored as tables. So, their visual model entities are bars but data only control the length attributes of the bars.

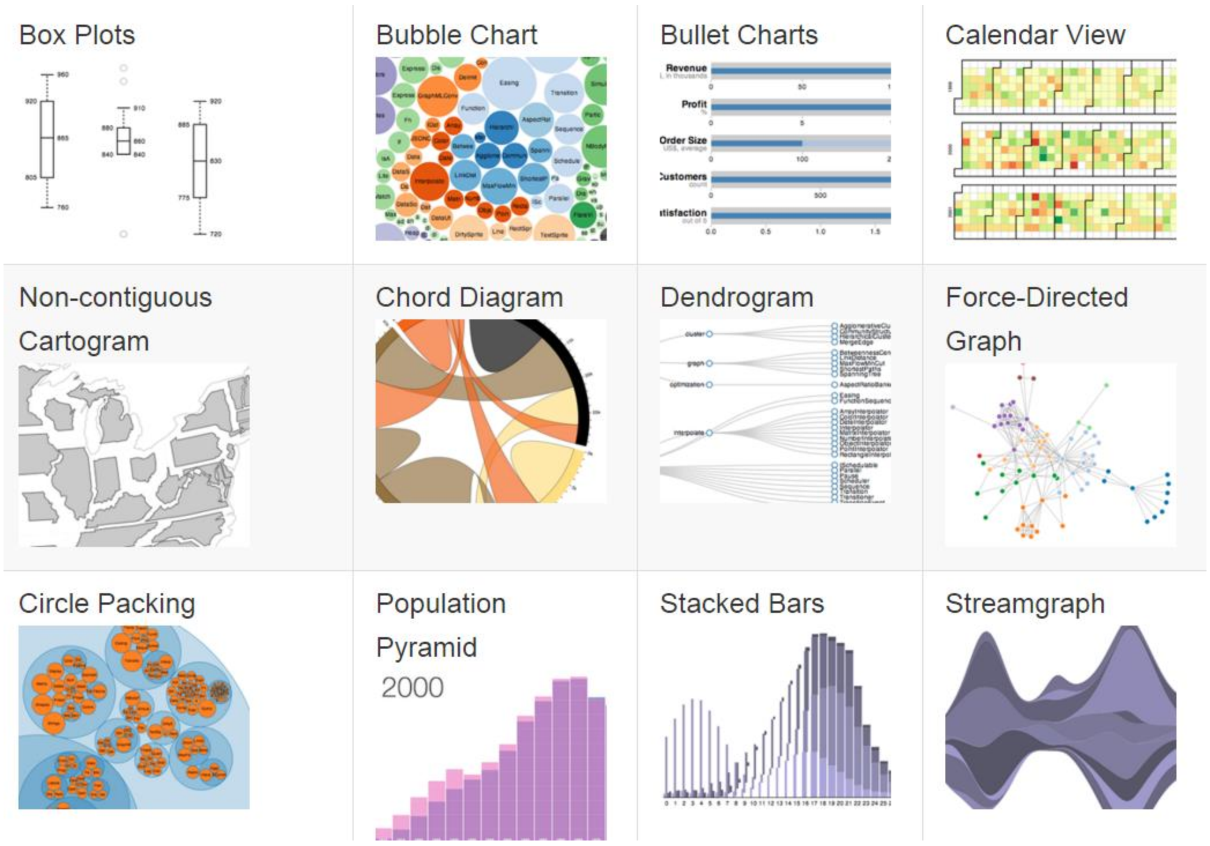


Figure 2. Some visual model examples from D3 Data Driven Document

3. Decoupling Data from Visuals

When it comes to write the visualization code, if the model-view paradigm is unknown to a programmer, it is very common to write the code in the following. The code uses a binary tree node as an example, in which visualization code is a method of the class.

```
class binary_tree_node {
  void draw() {
    ...// draw the center node
    ...// draw the left child pointer
    ...// draw the right child pointer
  }
}
```

Although it is object-oriented, it is obvious that the visualization code is coupled with the data type and thus not reusable by other types or problems. So, to further decouple the code, it is common to introduce the model-view paradigm described in the earlier section. To adopt the paradigm, the visualization code is encapsulated as an observer (see Fig. 3 for the observer design pattern [7]) to hold the visualization code as follows:

```
class binary_tree_node_observer {
  binary_tree_node* bt ; // ***
   void update() {
     ...// draw the center node based on bt
     ...// draw the left child pointer to bt.left
     ...// draw the right child pointer to bt.right
   }
}
```

In principle, the observer should register to a binary_tree_node object (aka DataSet). So, when DataSet is modified, update() of binary_tree_node_observer is then invoked to redraw the visual. When the pointer/reference of an binary_tree_node object is set to variable bt,the binding of the data and visualization code happens.

Writing visualization code in this way DOES separate data and visualization code; that is, the visualization code is moved to an observer, making a binary_tree_node object a pure data object. Unfortunately, the visualization code in binary_tree_node_observer is bound to the type of binary_tree_node, making the code in the observer not reusable elsewhere.

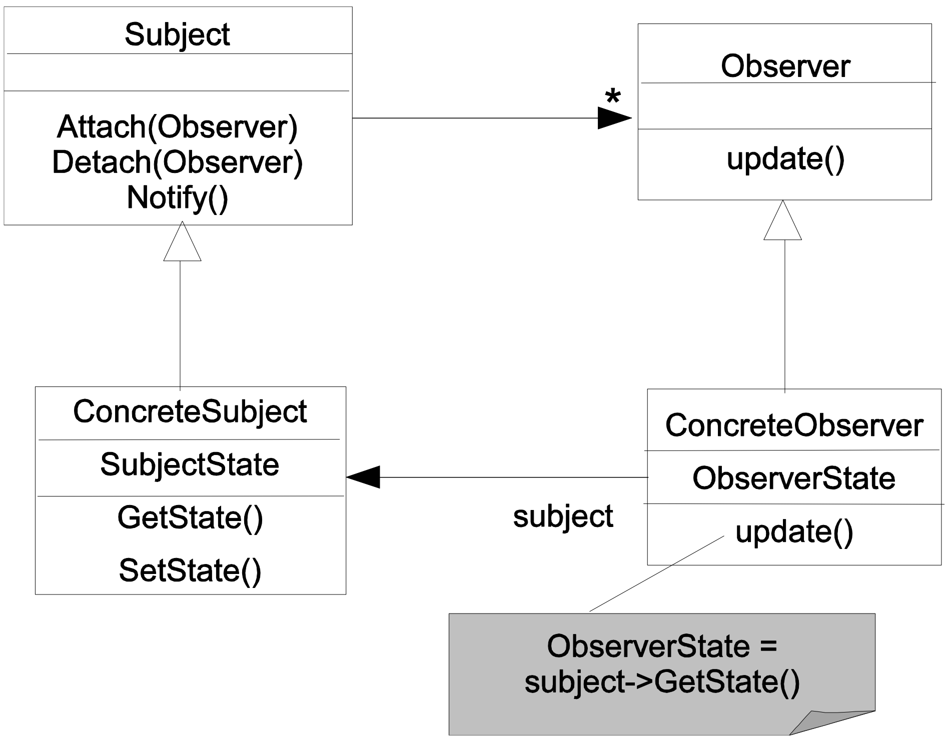


Figure 3. The Observer design pattern.

3.1 When observer design pattern fails?

Technically, to separate the data and visualization, a visualization should be constructed without the knowledge of how it will be used or what kind of data it will visualize, i.e. it should be decoupled from data. Any dependency on data makes a visualization component limited to render that data, unless the data is of a primitive type.

In Fig. 3, the observer pattern is shown. In this pattern, any ConcreteSubjects that are intended to be visualized should inherit the interfaces from a base class Subject. A ConcreteObserver which wants to visualize a ConcreteSubject should inherit a base class Observer and register at a ConcreteSubject. When a ConcreteSubject changes its state, a ConcreteObserver is notified by update(), which invokes subject->GetState() to get the newest state to redraw the visualization. Since subject is a reference or pointer to a ConcreteSubject, the source code of a ConcreteObserver is coupled and dependent on the TYPE of a ConcreteSubject. This dependency limits the ConcreteObserver to only visualizing the type of a ConcreteSubject. In other words, implementing visualization code naively with observer design patterns makes your visualization code not reusable to other applications, but it could be perfectly fine for your own applications. So, if your goal is to build a visualization library to be reused, a naive observer design pattern should never be adopted.

3.2 Maximum reusability with primitive types

So, to make a visual reusable, its shape should only be controlled/parametrized by “primitive types” (integer, float, etc.) in its visualization code. This finding can be simply explained. To draw a rectangle, the very basic APIs may be:

```
drawRectangle(int x,int y,int width,int height) { ... }
```

Such a fundamental API typically allows maximum reusability.

On the other hand, it is common to see that some additional APIs are supported to promote OO principle such as:

```
drawRectangle(Rectangle rect) {...}
drawBinaryTreeNode(BinaryTreeNode node) {...}
```

If your goal is to allow this visualization code to be reused, such a move actually reduce code reusability in an opposite way. By introducing more complicated data types in APIs to draw a visual, the reusability of the code is degraded.

Sometimes, some visualization libraries may still choose such a design. Therefore, to use the API, the data types of Rectangle or BinaryTreeNode must be provided by the library and users are responsible for using the types to create objects and then fitting their data into these objects before invoking the APIs. In this case, bindings occur when data are fitted into these objects. Fitting the data (by writing code) into these objects is called the binding method for reusing such an API.

4. Binding Design Patterns

The design of binding methods from data to visual models is very critical in constructing a visualization library/tool. Not only can the design affect the system reusability, but it also determines system architecture, extensibility, and flexibility. The problem of how to let users bind the data to visual models is a difficult ones. So far, there is no unique answer. In this section, four binding design patterns in visualization libraries/tools are described, studied, and compared. Understanding these patterns can help the programmers of visualization libraries/tools to determine their implementation directions.

4.1 Library object

Context:

Visualization code has been implemented as a set of function calls. In visualization code, some key data structures have been defined and implemented by a programming language to realize the visual model entities. Algorithms, such as layout algorithms, have been implemented to work on these key data structures.

Problem:

You want to publish your visualization library as a set of function calls.

Force:

The solution provided in this design pattern is most straightforward. So, there is no force to make the solution difficult. However, this design pattern is served as a basis for comparison.

Solution:

To implement a visualization as a function call, a visual model must be defined first explicitly or implicitly. Take a graph visualization again, its visual model contains two major visual entities – node and link. These are the key data structures (or entities) of the code. Without these key data structure, a graph layout algorithm can never be implemented.

To wrap the visualization code into a function call, a straightforward solution is to provide these key data structure types (e.g., class definition) to users so that they can use these data types to create objects which can then be passed to the function call directly. These objects are called *library objects*. Before calling the function calls, library objects must be created by users. In this process, users are responsible for filling the objects from data source. This process can only be done by programming.

The solution structure of this binding method is illustrated in Fig. 4. In the figure, the data from data source must be transformed and filled into the library objects and then these objects are used to invoke the function call. On the right hand side of the solution structure, we show the conceptual objects inside the visualization library. A visual model can contains several visual entities or composite visual entities (i.e., composite design pattern instance). Take graph visualization for example again, a set of nodes and links can form a sub-graph, which is a composite visual entity. A sub-graph may place additional constraints on layout algorithms to handle the nodes and links inside the sub-graph. Finally, layout algorithms are often included in the functional calls to work on these visual entities. Bindings occurs when you transform your data to fill the library objects. This binding method is accomplished by programming.

Example:

One of the popular visualization libraries that adopt this binding method is *igraph*[5]. Actually, igraph is not a pure visualization library. The main purpose of igraph is to provide graph algorithms to help programmers solve the computation problems of graphs. To visualize your data in igraph, you need to use the data types provided by the library. Fig. 5 is a sample code of using igraph. First, you need to create an (empty) graph using data type igraph_t, which is provided by the library. Next, you need to fill the data into the object. *igraph* can handle directed and undirected graphs. The igraph graph are multiset of ordered (if directed) or unordered (if undirected) labeled pairs. A directed graph can be imaged like:

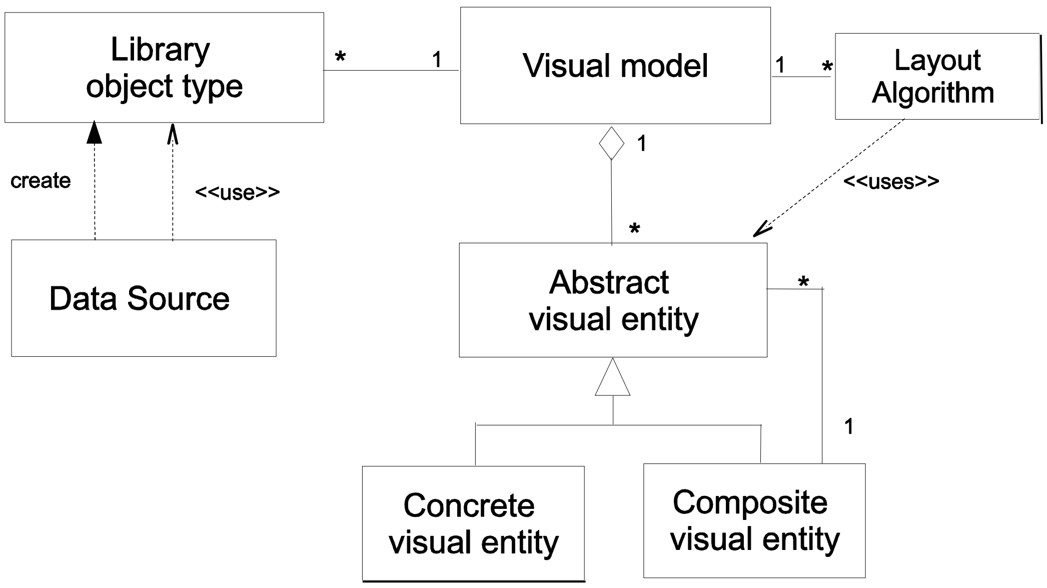


Figure 4. Library Object binding design pattern.

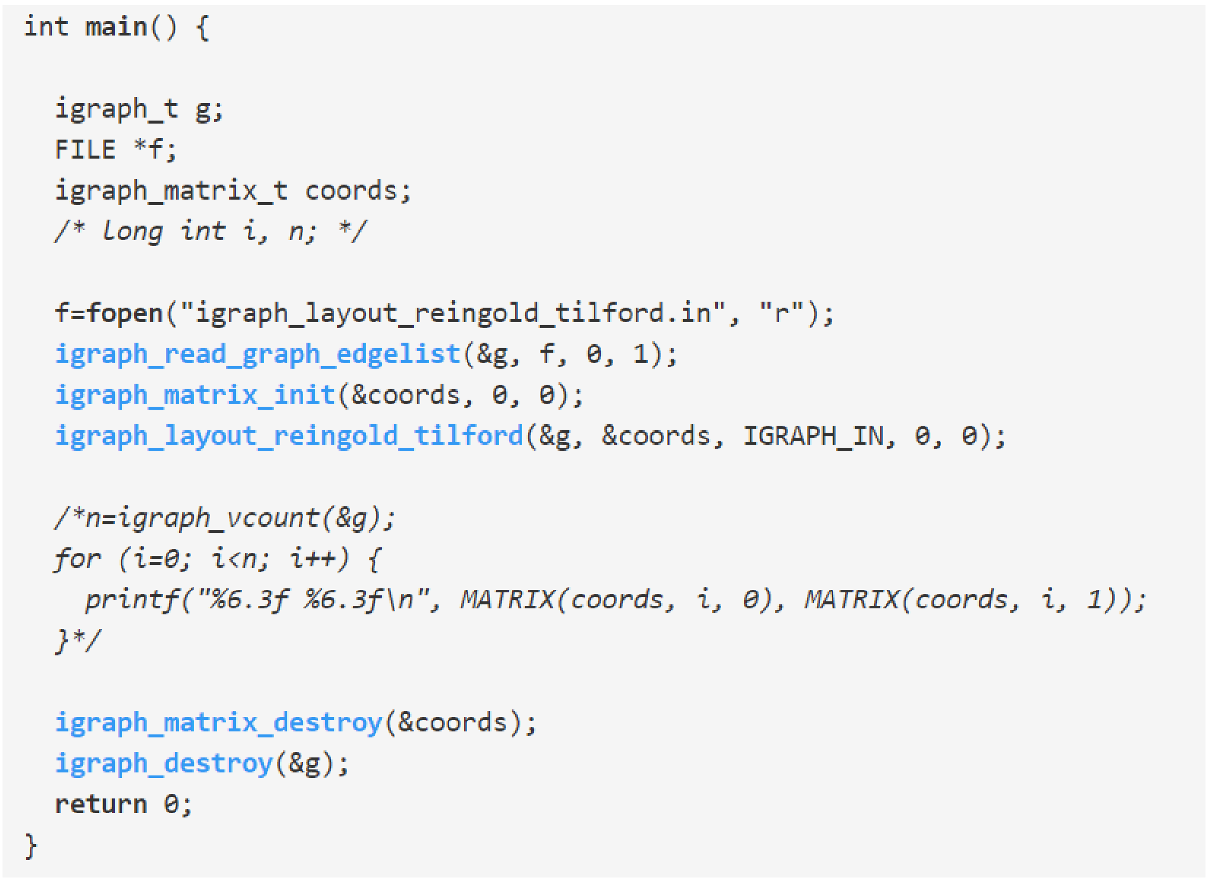
```
(vertices: 6
    directed: yes
    {
    (0,2),
    (2,2),
    (2,3),
    (3,3),
    (3,4),
    (3,4),
    (4,1)
    }
  )
```

To use *igraph*, you must abstract your data into the form above, where the nodes are always numbered from 0 to *n-1* and the edges are ordered pairs of vertex ids.

Consequences:

The visualization libraries adopting such a solution structure can be reused at the level of programming; that is, it requires programming to use the library. Porting to different programming languages is straightforward and the overhead is the cost of rewriting code by another language or providing a cross language API wrapper.

Users must handcraft some code to transform data to fill the library objects. In this process, relations between original data items and the elements in the library objects can be lost. For example, in the case of *igraph*, a vertex id is limited to *0,..n* . It could be non-straightforward to link a vertex *i* back to an original data item.



```
int main() {

  igraph_t g;
  FILE *f;
  igraph_matrix_t coords;
  /* long int i, n; */

  f=fopen("igraph_layout_reingold_tilford.in", "r");
  igraph_read_graph_edgelist(&g, f, 0, 1);
  igraph_matrix_init(&coords, 0, 0);
  igraph_layout_reingold_tilford(&g, &coords, IGRAPH_IN, 0, 0);

  /*n=igraph_vcount(&g);
  for (i=0; i<n; i++) {
    printf("%6.3f %6.3f\n", MATRIX(coords, i, 0), MATRIX(coords, i, 1));
  }*/

  igraph_matrix_destroy(&coords);
  igraph_destroy(&g);
  return 0;
}
```

Figure 5. iGraph sample code.

4.2 Meta Representation

Context:

Although a visualization library can be provided as a function call, it is programming language dependent and code reuse must be achieved at the level of programming, making visualization reuse limited to programmers of a specific programming language.

Problem:

You want your visualization library to be used beyond programming and can be easily integrated by other applications.

Force:

Independence of programming language: Visualizations can be reused beyond programming without linking the visualization code as a function call or library.

Solution:

A meta representation is defined by the library. This meta representation can be a file format, a string format, or any representations which are independent of programming languages. Fig. 1 shows the Meta Representation binding design pattern. The right hand side of the pattern is pretty much the same. In the left hand side, a meta representation is defined equally for each visual model. It is also possible that a meta representation can be shared by multiple visual models.

A meta representation parser should be built to parse the meta representation and then construct internal visual entities for visual models. Bindings actually occur when user data is transformed into a meta representation.

Examples:

One popular visualization tool that adopts this binding design pattern is Graphviz [8]. Graphviz defines a file format called DOT (see Fig. 7) to describe a directed or an undirected graph. Graphviz must then read a DOT file to produce a graph visualization as in Fig. 7.

Consequences:

- *Benefit:* Independence of programming language: As the visualization code is hidden behind the meta representation, the visualization libraries can be reused without programming. However, users must handcraft the code to transform data source to the meta representation, which requires the understanding of the meta representation. The relation between a data set item and a visual can be kept if transformation code is implemented properly.
- *Performance and instant update:* Independence of programming language: As the visualization code is hidden behind the meta representation, the direct memory linkage is lost and performance can be compromised by the file accessing and parsing. When data set is changed, visualization may not immediately reflect the changes at the speed of memory accessing.

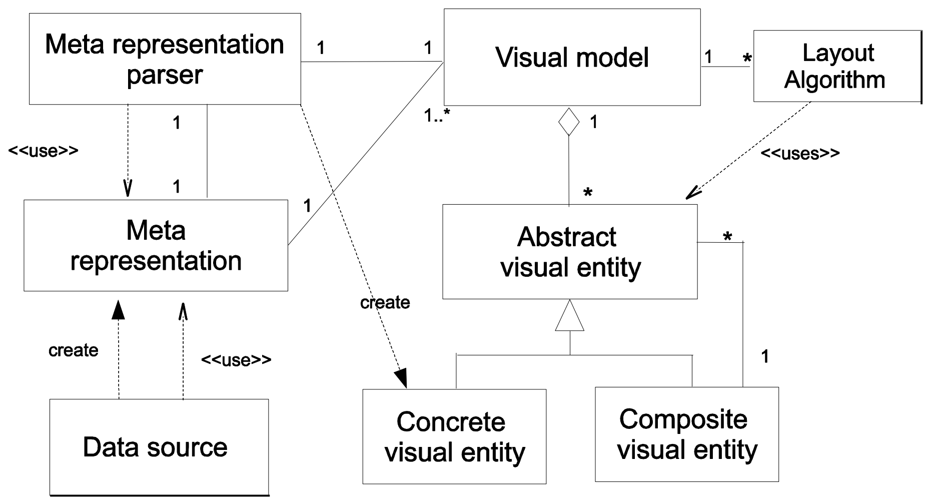


Figure 6. Meta representation binding design pattern.

```
digraph G {
     main -> parse -> execute;
     main -> init;
     main -> cleanup;
     execute -> make_string;
     execute -> printf
     init -> make_string;
     main -> printf;
     execute -> compare;
}
```

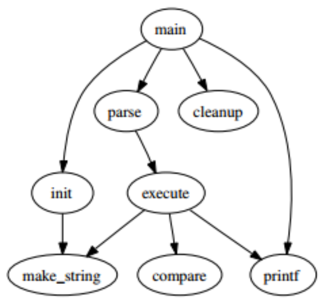


Figure 7. An example of DOT meta format in Graphviz and the visualization of the DOT example.

4.3 Standard Data Format

Context:

In principle, data can be stored and represented in arbitrary format, media, or memory. In some popular application domain, particularly web programming, data source are often stored as standard formats such as XML and JSON. The binding method become a problem on how to read data from these standard formats and then feed the data into visualization code. In these programming environments, there is no need to worry about how to parse these standard format files. Manipulating the objects in these standard formats is a default functionality in the programming environment. Standard formats can promote data exchange from applications to applications, which becomes a popular trend in information visualization.

Problem:

When data are stored in standard format you want your visualization code to read data from these standard formats.

Force:

Data exchange standardization: Instead of fitting data to visualization, visualization code should adapt to data sources represented in standard exchange formats to achieve more flexible data exchange between applications.

Solution:

In contrast to *Meta Representation*, the data in this pattern is already stored as standard format or standard data types and these standard formats are independent of visual models. In Fig. 1, the left hand side shows a different pattern for data source. You need to write code to extract data values from Open standard format or Standard data types and feed these data values to visual entities by yourself. Binding occurs when the code feeds the data into visual entities or standard data types provided by visualization library.

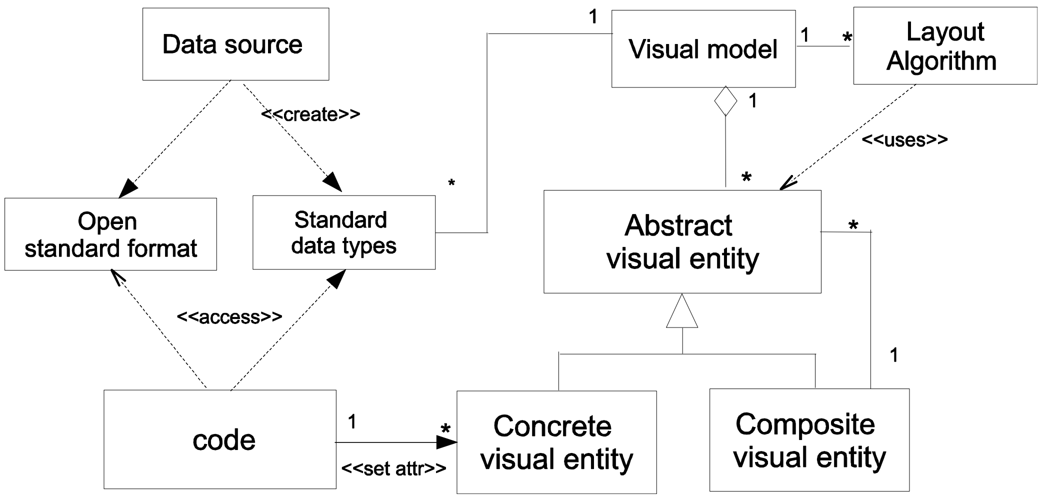


Figure 8:Standard Data Format binding design pattern.

Examples:

The most well-known approach that adopt this binding method is *D3* (Data Driven Documents)[2,9], which employs a declarative language to deal with mappings between data and visualization entities without worrying about much in drawing. D3 is based on Java script and SVG.

In principle, you can use D3 with basic SVG drawing commands to construct the visualization you want. However, if you choose a visualization library, in general, less code is written with the benefit of a declarative programming language. D3 let users focus on efficient manipulation of documents based on data rather than worry about drawing.

Consequences:

Adopting this binding method requires users to write additional code to manipulate data. The data are assumed to be stored in open standard formats so that accessing the items in these format is fully supported by the programming language and environment. That is, data can be manipulated in a much easier manner.

4.4 No visual models

Context:

In principle, there are arbitrary number of visualizations to be constructed and there are arbitrary data formats to be visualized. Once a visualization imposes the concept of a visual model, it also defines the constraints of data that can be applied to the visual model. For example, you can define a visual model which can only accept data organized as a matrix. If data are not organized as a matrix, a user must handcraft some code to transform data into a matrix, provided that meaningful semantics is kept.

On the other hand, given a set of data, it is unlikely that any fixed-mapping visualization will be available a priori to cover everything that might be of interest. A visual model is a fixed visualization, which can always be unsuitable for an arbitrary set of data.

When facing infinite construction choices, humans have learned to find common basic building blocks for all choices, set up the interfaces between the basic building blocks, and then establish engineering methods or frameworks to assemble these building blocks. This knowledge can be explained by a term – *composability*. Composability[1], in general, refers to a system design principle that deals with the inter-relationships of components. A highly composable system provides recombinant components that can be selected and assembled in various combinations to satisfy specific user requirements. A component may cooperate with other components, but dependent components are replaceable. So, to approach such a human knowledge in visualization, two key issues must be addressed: Composability and the use of basic build blocks.

Problem:

When fixed visualizations are not suitable for your data set, it is still possible to construct visualizations from existing visualizations.

Force:

Composability: Allow new visualizations to be composed from existing ones.

Solution:

One way to solve the crux of the problem is to remove the concept of visual models. Fig. 9 illustrates this design pattern. In this binding design patterns, data are read into objects. Next, the visualization library allows binding object to be created. A binding object binds the attribute of an object to a property of a visual (aka, an existing visualization) or to a computation unit. A mapping is composed by arbitrary number of binding objects and computation units. Since no visual models are defined, a user needs to construct a visualization by composing existing visuals.

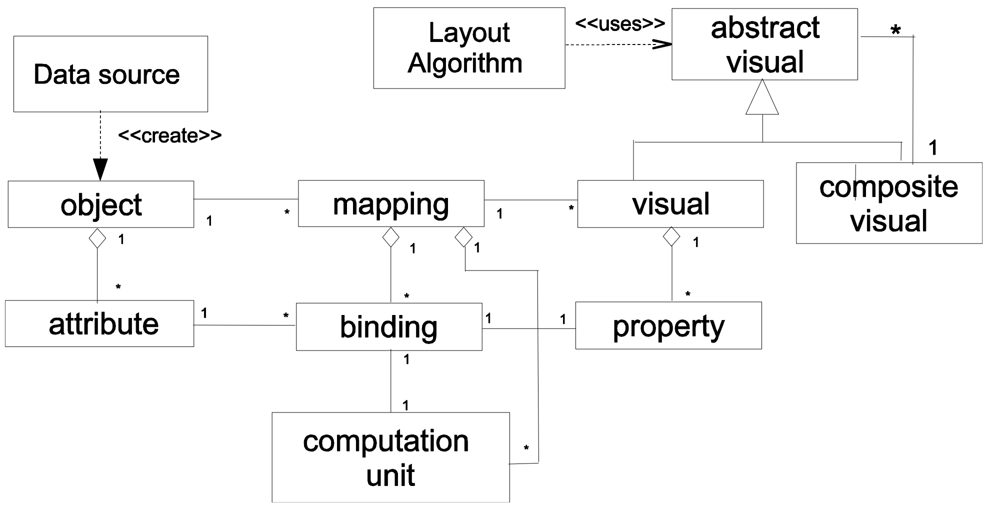


Figure 9. The NO-Visual Models binding design pattern.

Examples:

One visualization tool that adopt such a binding design pattern is xDIVA [3,4]. The design rationale of *x*DIVA originates from within common human knowledge described above. From the best of our knowledge, construction of visualization from basic blocks was first proposed by Cheng [3] and later Heer [9] proposed similar idea for the declarative approach.

To realize the binding objects and computation units in the design pattern, xDIVA proposes a visual programming language (VPL) called xDIVA-VML. The screen-shot of xDIVA-VML is shown in Fig. 10. It is comparatively much easier to start programming from these VPLs because these VPLs typically let users create programs by manipulating program elements graphically rather than by specifying them textually. However, VPLs are rarely used as general-purpose programming languages due to their limitations, particularly when problem complexity and scale are increased.

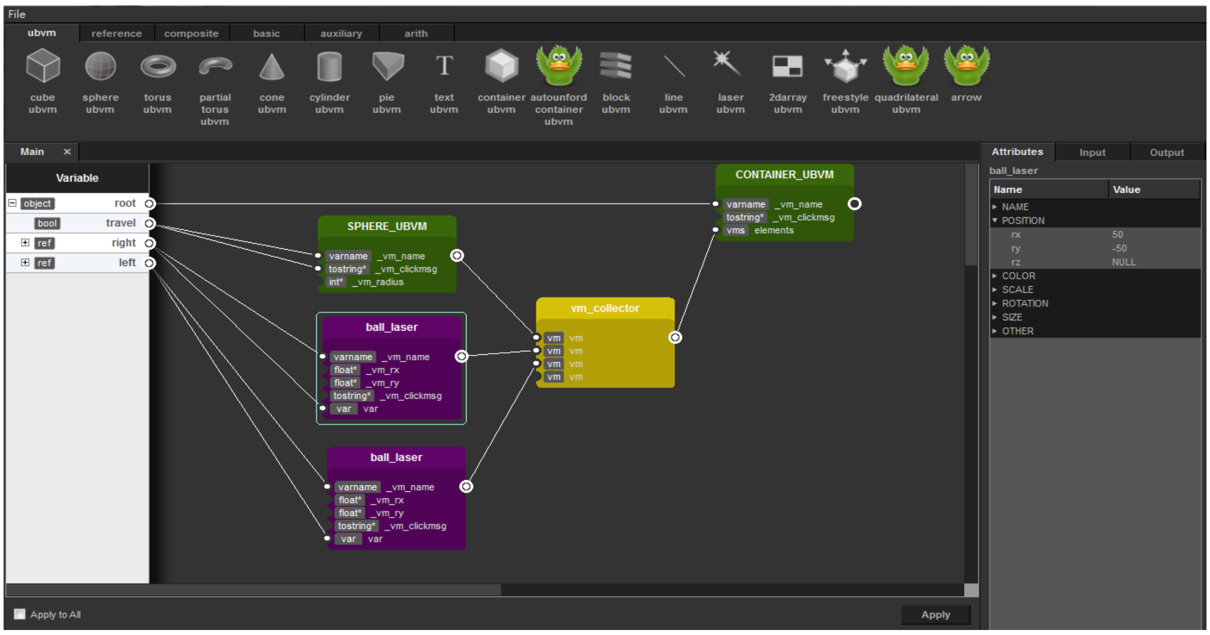


Figure 10:The screen shot of xDIVA-VML.

Consequences

The benefits of adopting this binding design pattern is the absence of visual models. A visualization library imposed by a visual model can be viewed as a fixed visualization that allows data to mapped to. However, given a set of data, it is unlikely that any fixed visualization will be available a priori to cover everything that might be of interest. When no fixed visualizations are available, this approach allows fast construction of visualization without textual programming.

The drawback of adopting this design pattern is that the cost of realization can be considerably high, as compared to library objects or meta representation. Second, fixed visualizations can be built to be efficiently executed and aesthetics can be fine tuned. That is, most work of a fixed visualization is already taken care of by the visualization library. On the other hand, building visualization from basic blocks can be more tedious in the beginning because users must have a visualization concept to be incubated at first. This problem could be alleviated if reusable visualizations are accumulated to some extent that a critical mass is reached.

Conclusions and Future Works

The discovery of these four binding design patterns originated from years of work in [3,4]. It is possible to have more binding design patterns that may not be categorized into the four design patterns. More surveys and studies are still ongoing.

So far, how to bind data to visuals is still a difficult problem. Comparatively, Library Object, Meta Representation, and Open Standard Format push most of the responsibility to users. However, bindings may be accomplished in a much more user friendly manner. Resolving the problem can therefore increase the reuse of visualization code. In addition, modern object-oriented programming languages have inheritance mechanism for code reuse but this mechanism apparently cannot be applied properly to visualization code reuse. There are more researches to be done in this direction.

References

[1] Composability. http://en.wikipedia.org/wiki/Composability. Accessed: 2014-09-18.

[2] D3 data-driven documents. http://d3js.org. Accessed: 2015-12-9.

[3] Yung-Pin Cheng, Jih-Feng Chen, Ming-Chieh Chiu, Nien-Wei Lai, and Chien-Chih Tseng. xdiva: a debugging visualization system with composable visualization metaphors. In Gail E. Harris, editor, Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA, pages 807–810. ACM, 2008.

[4] Yung-Pin Cheng, Han-Yi Tsai, Chih-Shun Wang, and Chien-Hsin Hsueh. xdiva: automatic animation between debugging break points. In Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26, 2010, pages 221–222. ACM, 2010.

[5] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. InterJournal, Complex Systems, 1695(5), 2006.

[6] Jean-Daniel Fekete. The infovis toolkit. In Ward and Munzner [14], pages 167–174.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patters: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional 1st Edition, 1995.

[8] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. IEEE Transactions on Software Engineering, 19(3):214–230, 1993.

[9] Jeffrey Heer and Michael Bostock. Declarative language design for interactive visualization. IEEE Trans. Vis. Comput. Graph., 16(6):1149–1156, 2010.

[10] Jeffrey Heer, Stuart K. Card, and James A. Landay. prefuse: a toolkit for interactive information visualization. In Gerrit C. van der Veer and Carolyn Gale, editors, Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2-7, 2005, pages 421–430. ACM, 2005.

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

[12] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. J. Object Oriented Program., 1(3):26–49, August 1988.

[13] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. IEEE Trans. Vis. Comput. Graph., 8(1):52–65, 2002.

[14] Matthew O. Ward and Tamara Munzner, editors. 10th IEEE Symposium on Information Visualization (InfoVis 2004), 10-12 October 2004, Austin, TX, USA. IEEE Computer Society, 2004.

[15] Chris Weaver. Bui lding highly-coordinated visualizations in improvise. In Ward and Munzner [14], pages 159–166.