Obvious: a Meta-toolkit for information visualization toolkits used in Visual Analytics

Thomas Baudel* IBM
Jean-Daniel Fekete† INRIA
Pierre-Luc Hemery‡ INRIA

ABSTRACT

Put an abstract here.

Index Terms: K.6.1 [Management of Computing and Information Systems]: Project and People Management—Life Cycle; K.7.m [The Computing Profession]: Miscellaneous—Ethics

1 INTRODUCTION

Over the past several years, we have seen the development of a wide variety of information visualization toolkits [4, 3, 6, 8]. The choice of one of this toolkits is a major initial step in the development of a visual analytics application and this variety can be confusing for visual analytics software developers.

Historically, this proliferation of toolkits can be explained by the fact that each created toolkit addresses a specific problem and/or is designed with a specific application domain in mind. Thus, there is a dispersion in terms of capabilities since each toolkit has unique and useful visualization and interaction techniques. For example, the Prefuse and JUNG toolkits offer several graph layouts algorithms [3] whereas Improvise supports very sophisticated coordinated views without graph capabilities [8].

That is why visual analytics developers are almost immediately confronted with a crucial decision: the choice of the information visualization toolkit to use. Since, this choice imposes the data structure and then this data structure imposes capabilities and techniques designed for it. Thus, if a developer needs a technique available in other(s) toolkit(s), he has to implement it from scratch: it is a waste of time and a source of errors.

To address this proliferation problem, we introduce Obvious - a meta-toolkit that abstracts and encapsulates information visualisation toolkits implemented in Java - as a way to unify their use and postpone the choice of which concrete toolkit(s) to use later-on in the development process. Obvious is mainly targeted at Visual Analytics software developers but also at library or toolkits developers if they want to provide algorithms or data converters not restricted to one toolkit.

To provide evidence of the usefulness of Obvious, we have developed Obvious binding modules for several toolkits (Infovis toolkit [4], Prefuse [3] and Jung [6]); they implement the interfaces and abstractions defined in the specification. These bindings have been used to build some proof-of-concepts examples combining different toolkits and also to create complete systems used by ongoing research projects such as [7]. During the development of bindings, we have seen important design questions emerge regarding the interpretation of the reference model; we report them here to help clarify the InfoVis reference model and trade-offs in its implementation.

In addition, Obvious allows developers to eliminate the crucial choice of the toolkit and to avoid rewriting existing functionalities such as file import and export modules, as well as analytical algorithms. The following use case shows it is now possible to combine toolkits. For example, they can choose a data model from JUNG toolkit for a graph, then query it with Prefuse predicates, use a layout introduced in Infovis toolkit to display it and still used network algorithms introduced in JUNG.With obvious, there are no more design restrictions imposed by an initial choice for developer.

*e-mail:baudelth@fr.ibm.com
†e-mail:Jean-Daniel.Fekete@inria.fr
‡e-mail:Pierre-Luc.Hemery@inria.fr

1.1 Goals and Social Process

Obvious is not another toolkit, it is a set of interfaces abstracting services provided by information visualization toolkits following the InfoVis reference model [2]. It has been specified during a workshop gathering several major authors of toolkits [1] based on the level on consensus reached at that time among the developers.

A typical scenario of Obvious would be the design of VizTree [5], a visualization for monitoring massive time-series. The authors of VizTree encode very long time-series of a continuous value as a suffix tree. Describing the details of this encoding is beyond the scope of the paragraph; the point is that the associated tree visualization has been implemented by specialists of data-mining and leaves room for improvements in term of visual mapping and interaction. Using Obvious, the authors would first connect their computed data structure to the data model of Obvious. There are two ways of doing that: use the Obvious data-model directly or use the native data-model implemented for mining the time-series and wrap it with an implementation of the Obvious data-model. Both are possible and will be chosen according to the amount of work and flexibility offered by one option or the other. Once an Obvious data-model is available, the authors of VisTree can start exploring which toolkit will provide them the best support for their visualization. They can choose among the InfoVis Toolkit, Prefuse and JUNG to visualize tree data. Once the best one has been chosen, the interaction can be crafted either on top of the abstraction provided by Obvious - to keep the option of switching the final implementation - or using the native toolkit controls to keep a tighter control of the interface. If desired, the interface can also be improved by adding other visualizations associated with the computation of the prefix tree or of statistics associated with the data. If multiple-coordinated views are required for that, Improvise visualization and views can be added to the interface using the same data model. In that scenario, Obvious has enabled data-mining researchers to focus on their skills and to use state-of-the-art visualization components at a later stage of the development of their application.

Another scenario [DDupe]

1.2 Targeted uses

Scenario A user wants to implement a VA application starting from scratch.

2 RELATED WORK

2.1 Existing toolkits

2.2 Standardization processes

ISO, Internet, SQL

3 Design Overview

a) generic implementation of the InfoVis ref. model

b) social process involving workshop to build consensus

c) selecting emerging consensual patterns

Obvious is organized according to the Information Visualization Reference Model in three main packages: data, visu and view. Additionally, it provides utility classes in the util package.

3.1 Obvious Design patterns

4 Data Model

This section describes the data model used in Obvious and the way to create data structures. It also exposes the util package for the data model. The data model used in Obvious has been largely specified during the workshop: a consensus has been found among all participants.

The data model introduced in Obvious derived from the proxy tuple design pattern exposed in [2]. However, originally, this model was containing table, graph and tree (an extension of graph) classes. Obvious was using data the relational graph pattern described in [2]. Nevertheless, if this patterns improves extendability, graphs can not be manipulated in an object oriented way and as many developers are used to manipulate object oriented graphs, it was not satisfying to only use the relational graph pattern . That is why, we decide to offer possibility to use proxy tuple pattern, a design pattern combining benefits of the relational graph pattern and of object oriented graphs.

In addition, a major difference exists between our patterns and those introduced in [2]. Our patterns used schema to describe the columns of a table (type, name and default value) instead of a column object, that does not exist in Obvious. Schemas have been introduced, since it is efficient to gather all meta-data for the columns of a table in one unique structure and since it facilitates tables and networks instantiations with a factory.

Thus, the data model used in every implementations [described in the corresponding section] is built around tuples: abstractly tables are composed of tuples and graphs are described as networks i.e. a combination of two tables one made of nodes and the other of edges (both classes derived from tuples).

Also, this model is completed by factories, that allows data structure instantiations. Logically, they used the well known factory pattern. With those factories, it is possible to instantiate tables and networks from a schema or from an existing object from a targeted Obvious implementation (e.g. a Prefuse table or a JUNG graph...). We also give the possibility to use parameters to provide more arguments used in targeted toolkits. For example, in the Prefuse implementation of Obvious, parameters are used to specify for a graph its source and its target node columns in the edge table.

Finally, we have defined an utility package obviousx, named in the same way as javax. This package provides different kinds of utility classes for the Obvious data model. First, we have defined in obviousx, reader and writer interfaces allowing to create gateways between the Obvious data model and common data formats such as CSV and GraphML. It is useful since it gives to software developers a standard way to import and export data in Obvious whatever the underlying implementation of the data model is. In addition, for data providers, it simplifies their works because they only have to develop one reader and one writer to be compatible with a large number of toolkits. With the same logic, obviousx furnishes a Java TableModel compatible with the Obvious one: it allows to quickly create a JTable from an Obvious table. Finally, obviousx also provides wrappers to transform obvious data structures into common existing data structures (Prefuse, Ivtk, Jung, more to develop) in order to avoid when using a visualization to copy data from one model to another.

5 Visualization and View models

Unlike the data model, during the workshop, no consensus emerged concerning the Visualization and View models in Obvious. The main reason is that currently different approaches exist among toolkits : the monolithic and the polylithic approaches. So, for the moment, there is no way to create an abstract layer for visualization as we did for the data model. Further discussions and workshop may address this problem.

However, we propose a solution to address this problem for monolithic toolkits. We propose to wrap monolithic components in a black box fed by an Obvious data structure and a map of parameters allowing to configure the component. For example, it is possible to indicate the X and Y axis columns for a scatter-plot. If a developer needs a non existing visualization component, it simply adds to choose an implementation toolkit and to create, then this new visualization will be compatible with all data models: our solution does not generate extra costs of development.

Concerning the view, we choose to implement a simplified version of the camera pattern introduced in [2]. The black box concept is still present : a view simply wraps a view component of a targeted toolkit.

6 Implementations

In this section, we point out encountered difficulties, raised problems and learned lessons during implementations of Obvious. Each implementation has its own particularities and allows us to better identify gaps in existing design patterns and models. Thus, we start by describing every specific points in realized implementations and then we conclude with lessons learned during the whole step of implementation.

6.1 Prefuse

Prefuse was the first targeted implementation of Obvious since their design patterns are very close. The binding implements all abstractions described in the core Obvious interfaces for all data model structures, visualizations, views and predicates. [to be continued...]

Prefuse is currently the only polylithic Information Visualization toolkit [ref]. However, Obvious does not currently offer a visualization abstraction for polylithic components since. Thus, we choose to provide some pre-built monolithic components based on Prefuse for well know visualization techniques such as scatter plots, force directed graphs, radial graphs... Currently, if a software developer wants to use a non existing technique, he has to write it with Prefuse as a component and wrap it with the Obvious visualization interface.

6.2 Infovis Toolkit

The Obvious implementation based on Infovis Toolkit realizes all introduced design patterns. Since it is a monolithic toolkit, we do not encounter problems seen with the Prefuse implementation. We simply bind Infovis Toolkit components to Obvious ones.

Implementing the data model was a bit more tricky because Infovis Toolkit owns a lot of different data structures (DynamicTable, Table, Graph, Tree) and super interfaces for tables and graphs do not provide equivalent to methods defined in Obvious interfaces. That is why we can not wrap Infovis toolkits component directly: some complementary code that plays with existing methods in Infovis Toolkit was needed and so complicates the development and the maintenance of the implementation.

6.3 Improvise

Currently, the Obvious implementation based on Improvise only implements the data model part - for tables -.

Even if Improvise is a monolithic toolkit, we can not directly bind Improvise visualization components, since the toolkit does not expose publicly its visualization pipeline. Components are intended

to be totally monolithic from scratch. To address those problems, a solution to expose the visualization pipeline is to create in Improvise a specific visualization interface making public needed methods.

In addition, Improvise does not provide dynamic data support: a functionality intended to be in every obvious implementations.

6.4 JDBC

The Obvious implementation based on JDBC only uses the data model of Obvious. It was designed to prove that our data model can be applied to a large variety of existing data models or sources. In addition, this implementation is useful since databases are often used as data sources for visualization. That is why it is interesting to directly link our model to an SQL database with JDBC.

Concretely, it translates obvious methods into SQL queries. For example, getters are implemented as SELECT queries, setters as UPDATE ones, add as INSERT queries and remove as DELETE queries. Also, this original implementation passes all predefined unit tests.

In addition, we use it to experiment advanced notification model that does not exist in current toolkits. The tested model includes support for transaction and batch for modifications: this functionalities are based on mechanisms provided by the database. Those techniques are useful for scalability since large data modifications often occur when working with a visualization.

6.5 JUNG

JUNG implementation of Obvious realizes Visualization and View patterns and the Network abstraction, because JUNG already follows this pattern and does not have the notion of table. Since JUNG can easily provides monolithic visualizations , we simply bind existing JUNG visualization components to Obvious ones. As JUNG does not offer schema class, we use our predefined schema implementation from obvious core package to respect our patterns.

From all existing implementations, this one was the easiest to create, since it fits well in Obvious assumptions (network pattern for the data model and easily providing monolithic components). Thus, it demonstrates when an existing toolkit and obvious have the same (or close) hypotheses, it greatly facilitates implementation.

6.6 Units tests

Since, we have defined a consensual abstraction for the data model, we take advantage of this to create unit tests for this abstraction. If such a consensus emerges for the visualization and/or the view part, the same approach could be adopted.

Due to the similarities of Obvious and Prefuse, the binding has been used to set up unit tests for the data model in Obvious. Unit tests allow authors of Obvious bindings to automatically test whether their implementation behaves in conformance with the intended semantics of Obvious. Also, authors are able to extend these existing tests to perform more advanced features for their binding.

Concretely , unit tests have been defined with JUnit for Schema (14 tests) , Table (11 tests), Network (13 tests) and Tree interfaces (8 tests) in the Obvious core package.

6.7 Lessons learned

We have learned several lessons during the development of those implementations: mainly concerning lacks of precision in the Information Visualization reference model. We can list the following lessons:

- it was needed to specify a clear semantic for notifications that can support simple models and more advanced ones (with batches and transactions for example)
- it is needed to support the polylithic approach to define an abstraction model for visualizations
- toolkits needs to adopt patterns close to Obvious ones in order to enhance code quality of the implementations and to facilitate the sharing of functionalities among existing toolkits

7 Evaluation

Formally evaluating the effectiveness of a meta-toolkit for visual analytics is complex. Arguably the most convincing method would require two groups of programmers of equivalent skills to implement the same set of visual analytics programs with and without Obvious. Then, a judgment could be made from the time spent and the quality of the results. This methodology has been used to assess the InfoVis Toolkit [4] with students but is impractical for real Visual Analytics applications that are more complex and would not fit the scope of student projects.

Another method, used to validate Prefuse [3] would be to re-implement complex Visual Analytics applications using Obvious and assess the results, again in term of time and quality. This is what we have done and we report on our results here.

7.1 Coding applications with Obvious

This section will show how obvious can implement well know examples of information visualizations techniques introduced in several toolkits such as prefuse and Infovis toolkit by combining different Obvious components.

The two first examples use two different implementations (obvious-ivtk and obvious prefuse) to illustrate the same use case : display of a Network using the standard graph layout of the targeted implementation. The third example demonstrates how to create an application by combining component from different Obvious implementations.

The first step is to create an obvious data structure. We have chosen to illustrate two ways of creating an Obvious structure : one is to wrap an existing data structure into an Obvious one, another is to read a file using a common format (CSV, GraphML...). As shown, in the code sample, factories are provided for each implementation and they allow the developer to use the convenient strategy to set up the data structure.

Once the data is ready, the next step is to create the visualization. As explained in the Visualization section, to create the Obvious Visualization, it is needed to provide additional parameters such as the label column used to display nodes in the first two examples. For the scatterplot example, columns used for X and Y axis are indicated in the map of parameters.

Finally, created Visualizations are injected in a View component. Since all examples use Swing, they are added to Jframes and then those frames are displayed to the final user. It is also possible to add zoom and pan control to created views in order to add interactivity to those examples.

7.2 Example using Weka

Weka is a software suite of machine learning. It has been proven useful to design such applications, even in visual analytics. That is why in the obviousx package (utils), we have chosen to introduce mechanisms to support Weka. So, in practice, with one line of code, it is possible to create a Weka Instances (Weka data structure) from an Obvious table. Thus, each toolkit implementing Obvious can be easily linked with Weka.

7.3 EdiDuplicate (a DDupe-Like application)

During the development of Obvious, the meta-toolkit has been used to create piece of software used in scientific project such as Ediflow [1]. In this project, we have to check for duplicated authors in the INRIA database of publications with the Ediflow environment. Thus, we decide for our use case to create a DDupe equivalent in Java able to work with a database: EdiDuplicate.

DDupe is a software initially written in .NET dedicated to detect and merge duplicated nodes (often modeling people) in (social) network to facilitate data analysis. It uses similarity metric to compare each pair of authors and class results in descending order of similarity. In addition, DDupe allows the user to see the neighbourhood of the pair of nodes before merging them, in order to check if common nodes exist among their neighbors and then to confirm metric results.

Our application EdiDuplicate offers the same possibility. Concretely, we have implemented a loader to create an Obvious network from the database. This structure is then fed by an external application with metrics for each pair of authors. Then, those statistics are displayed in a JTable (automically created from the Obvious structure). When, the user clicks on a cell a view of the neighbourhood of the current nodes is created. Then, with this information, the user is able to decide if the potential duplicated authors has to be merged. In addition, it is possible to query the Obvious structure to only display pair of duplicates for a specific authors. The user can also change the layout used to display the neighbourhood network with a JList: all graph layouts introduced in Infovis Toolkit are available.

Concretely, the application mainly combines Obvious components and Swing components (derived from Obvious structure). For the data model, an Obvious Network is used (from the obvious-ivtk implementation), the visualization and the view part are also provided by this implementation. Building this application takes about less than a week.

8 Future work

a) new candidates for inclusion

b) adding new toolkits (Discovery) / evolving existing toolkits to comply better

c) social process

Will port it for other languages to stabilize the framework and facilitate learning.

9 Conclusion

Work in progress by design, contributions welcome

Service to the VA community

Among many concerns, alleviate at least the early choice of the toolkit

Acknowledgements

The authors wish to thank A, B, C. This work was supported in part by a grant from XYZ.

References

[1] Vismaster workshop on visual analytics software architecture. http://code.google.com/p/obvious/wiki/Motivation, December 2008.

[2] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE Trans. Vis. Comput. Graph.*, 12(5):853–860, 2006.

[3] J. Heer, S. K. Card, and J. A. Landay. prefuse: a toolkit for interactive information visualization. In *CHI*, pages 421–430, 2005.

[4] Jean-Daniel Fekete. The Infovis Toolkit. ACM Symposium on User Interface Software and Technology (UIST 2003) Conference Compendium, November 2003.

[5] J. Lin, E. J. Keogh, S. Lonardi, J. P. Lankford, and D. M. Nystrom. Visually mining and monitoring massive time series. In *KDD*, pages 460–469, 2004.

[6] J. O'Madadhain, D. Fisher, S. White, and Y.-B. Boey. The jung (java universal graph/network) framework. Technical report, UCI-ICS, 2003.

[7] Vronique Benzaken and Jean-Daniel Fekete and Pierre-Luc Hmery and Wael Khemiri and Ioana Manolescu. EdiFlow: data-intensive interactive workflows for visual analytics. In *International conference on Data Engineering (ICDE 2011)*, Hannover, Germany, 04 2011. IEEE. to appear.

[8] C. Weaver. Building highly-coordinated visualizations in improvise. In *INFOVIS*, pages 159–166, 2004.