Obvious: a Meta-Toolkit to Encapsulate Information Visualization Toolkits One Toolkit to Bind Them All

Thomas Baudel* IBM
Jean-Daniel Fekete† INRIA
Pierre-Luc Hémery‡ INRIA
Jo Wood§ City University

ABSTRACT

This article describes "Obvious": a meta-toolkit that abstracts and encapsulates information visualization toolkits implemented in the Java language. It intends to unify their use and postpone the choice of which concrete toolkit(s) to use later-on in the development of visual analytics applications. We also report on the lessons we have learned when wrapping popular toolkits with Obvious, namely Prefuse, the InfoVis Toolkit, partly Improvise, JUNG and other data management libraries. We show several examples on the uses of Obvious, how the different toolkits can be combined, for instance sharing their data models. We also show how Weka, a popular machine-learning toolkits, has been wrapped with Obvious and can be used directly with all the other wrapped toolkits.

We expect Obvious to start a co-evolution process: Obvious is meant to evolve when more components of information visualization systems will become consensual. It is also designed to help information visualization systems adhere to the best practices to provide a higher level of interoperability and leverage the domain of visual analytics.

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 few years, several information visualization toolkits have flourished in various languages such as Java [14, 21, 22, 26, 3], C++ [1, 13], Flash [17] or JavaScript [19, 7, 10] to name a few. When starting a Visual Analytics project, the choice of the toolkit is a major initial decision and this proliferation of toolkits can be confusing for visual analytics software developers who know that an inappropriate choice can lead to unanticipated limitations during the development of the application.

Historically, this proliferation of toolkits can be explained by several factors: each created toolkit addresses a specific set of problems, is designed with a specific application domain in mind or simply offers different tradeoffs. However, it results in a dispersion in terms of capabilities since each toolkit has unique and useful techniques for visualization and interaction. For example, the Prefuse [21] and JUNG [22] toolkits offer several graph layout algorithms whereas Improvise [26] supports very sophisticated coordinated views with limited graph capabilities.

The choice of the information visualization toolkit should be made early because it imposes not only the visualization techniques but also the data structure to work with. For an application dealing with small quantities of data, copying data from one structure to another is possible but not for visual analytic applications that usually manage data sets too large to be duplicated at all. Therefore, most data-management and analysis will be made on data structures compatible with the visualization and tied to the visualization toolkit.

Once the choice is made, any missing components have to be added specifically to the toolkit: if a special data manager is required (e.g. reading a particular data format), it has to be implemented specifically for the data structure managed by the toolkit. Analysis not supported by the toolkit requires the authoring or adaptation of analytical toolkit components. Likewise, if visualization techniques are required that are not supported by the chosen toolkit, they must be added, creating a strong dependency that may prevent change of toolkit later-on in development.

The effort required by one application to implement the missing components cannot easily be reused in other applications using another toolkit. Therefore, important resources are wasted reimplementing data converters, analysis modules and visualization techniques.

To address this early proliferation problem, this article introduces *Obvious*: a meta-toolkit that abstracts and encapsulates information visualization toolkits implemented in the Java language 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 promote sharing of data managers, converters or algorithms not restricted to one toolkit.

This article presents three contributions:

1. it describes the design and implementations of Obvious,
2. it reports some lessons learned when wrapping existing toolkits with Obvious,
3. it presents rationales for the social process we started and want to follow for the future of Obvious.

The article is organized as follows: after the related work section, we describe the design of Obvious. Section 3 reports on the wrapping of several toolkits and components with Obvious. Section 4 shows examples of Obvious in action to assess its usefulness. Section 5 discusses the social process we have used and how we envision the evolution of Obvious before concluding.

2 RELATED WORK

Obvious is a set of interfaces and extension classes for wrapping around existing information visualization toolkits. It generalizes and extends the standard architecture as defined in the Information Visualization reference model to try to abstract all the existing implementations. In this section, we list some major existing toolkits and explain what they share and how they differ. In the second section, we describe the most common standardization processes for software systems.

2.1 Visualization Toolkits

Pretty much all existing information visualization toolkits follow the InfoVis reference model initially specified by Ed Chi and refined by Card, Mackinlay and Shneiderman [12, 11] and has been described as a design pattern in [20]. The model defines three stages: *DataSet* or *Data Tables*, *Visualization* or *Visual Structure*

*e-mail:baudelth@fr.ibm.com
†e-mail:Jean-Daniel.Fekete@inria.fr
‡e-mail:Pierre-Luc.Hemery@inria.fr
§e-mail:jwo@soi.city.ac.uk

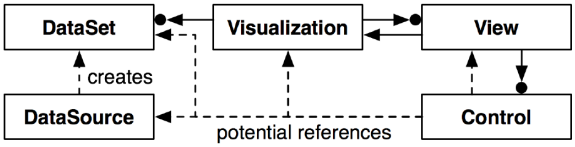

Figure 1: The Information Visualization Reference Model [20]

and *View* (Figure 1). One of its main benefits is that it explicitly represents interaction, in contrast to older visualization models. Several articles have described the concrete design of an information visualization toolkit. We report here on the common and the specific parts.

The InfoVis Toolkit [14] is based on an *in-memory database manager* where data is organized in columns — contrary to most persistent relational databases — to improve the memory footprint and allow addition of new attributes that are needed to manage the interaction (e.g. selection or filtering) and to hold attributes computed on demand. The main challenge being the support of interactive performance for rendering and dynamic queries with a small memory footprint. The visual structure is managed using a *monolithic* architecture [6]: each visualization technique is implemented as a specific class (e.g. ScatterplotVisualization, ParallelCoordinatesVisualization, TreeVisualization) that performs the mapping between the data set and the graphics items to render. Finally, the view component is the same for each of the visual structures and takes care of scrolling, zooming, overlaying magic lenses (e.g. Fisheye or Magic Lenses). A *notification mechanism* implements the communication between the data tables and the visual structure: each time a data table is modified, it notifies all the registered handlers of the details of the modification. The interaction is managed by *Interactor* objects that are associated with the visual structures; the views are generic and forward interaction managements to the Interactors. One specific feature provided by the InfoVis Toolkit is layering: visualization can be composed on top of each others. Composite visualizations are useful to build complex visualization by breaking them into simple parts. For example, node-link diagrams are split into links managed as a layer and nodes as another. Magic lenses and Fisheyes are also managed as layers on top of other visualizations.

Prefuse [21] also relies on an in-memory database with notification but implements the visual structure using an extension of the data model (a visual table derives from a data table). It then transforms the data into a *polylithic* graphic structures whereas all the other toolkits use a *monolithic* architecture. In a polylithic architecture, there is only one component in charge of all the visual structures. A visualization object is responsible of managing a visual structure: it contains visual tables that augment data tables with graphic attributes (shape, color, etc.) Visualizations are in charge of computing the layout (assigning a position and shape to visual items), the graphic attributes and animations. Visualizations use a *Renderer* object to actually display visual items. Users can control which renderer is used depending on the visualization and the object itself. In Prefuse, data managers, visual managers and views are generic, offering a very clean interface to the application programmer. However, as noted by Bederson at al. [6], polylithic toolkits have a steeper learning curve than monolithic ones because the polylithic components do not work out of the box, they always need to be configured. To address this issue, Prefuse comes with code samples that simplify the initial setup.

Building upon their experience in the Prefuse toolkit [21], Heer et Agrawala [20] have derived software design patterns that are common to information visualization applications and toolkits.

Improvise [26] relies on an in-memory database with notification that is row-oriented and its visual structures are monolithic. The main characteristic of Improvise lies in its management of coordinated views. To this aim, it relies on several design patterns not supported by Prefuse; compared to the other information visualization toolkits, it adds a coordination component that is central and extends the notification mechanism implemented by the InfoVis Toolkit or Prefuse.

Discovery [4, 3, 5] shares most of its characteristics with Prefuse: it uses an in-memory, column-oriented database and a polylithic graphic model. Its two main features are 1) the absence of a scene graph, replaced by a dataflow pipeline made of short operations called *functors* that renders directly from the data-model, and 2) a deferred notification strategy to allow data editing.

Other information visualization toolkits can mostly be described using the four toolkits above, even if they use a different programming language. Tulip [1] is a graph-oriented toolkit programmed in C++ that uses data tables for vertices and edges, like the InfoVis Toolkit and Prefuse. It implements several complex graph layout algorithms and uses OpenGL for its rendering but the conceptual architecture is table-based and monolithic. Therefore, information visualization toolkits share a global organization, they all implement an in-memory database with two variants (row-based or column-based), a visual structure with two variants (monolithic or polylithic) and several specific features. Even if some choices made by toolkits designers were carefully decided, other were probably made without being aware of the alternatives. Combining the best possible features for a next-generation toolkit might be tempting but there are still trade-offs that cannot be solved. For example, the power of coordinated and linked views offered by Improvise comes at the cost of maintaining caches that should be flushed when the data change so there seems to be a trade-off there that still needs research to be solved.

There are also lower-level toolkits that can be used to build visual analytics applications. Two popular families are graphics libraries and graph libraries.

2.2 Graphics Libraries

Visual analytics applications can manage their own data structure and take care of the mapping from data to visualization on their own. At this point, they can use *scene-graphs* or *direct-graphics* libraries.

Scene-Graph toolkits can manage the visual structure and view as described in the reference model. They are focused on computer graphics and interaction: they only deal with the visual structure and view. Piccolo and Jazz [6] are popular 2D scene-graph managers that have been used to create several information visualization applications (e.g. [23, 8].) An early version of Piccolo has also been used as graphics engine for the Cytoscape graph visualization system [24] but dropped for performance reasons.

High-performance information visualization applications use scene-graph optimization techniques to speed-up the rendering of scenes. Tulip [1] and Gephi [2] maintain a spatial indexing structure to avoid rendering objects that are not visible.

Although scene-graph technologies are mature and used in a wide variety of graphics applications such as games, virtual-reality applications and scientific visualization systems, they are not always adequate for information visualization systems because they require the explicit specification of geometry and graphic attributes for each displayed objects. Very often, information visualization can quickly compute graphic attribute and even geometry from data attributes. For example, the position of an item using a scatterplot visualization is computed using a simple affine transformation from two data attributes. There is no need to store the computed values when computing them on the fly is very cheap. The same is true for color etc. Copying and storing this information is costly in term of

time and memory.

Direct-graphics libraries such as *Processing* or *OpenGL* can also be used to implement the visualization technique while drawing for rapid prototyping or high-performance reasons.

[Add a section on Processing]

Still, when separating the data-model from the visual model, scene-graph managers offer more flexibility than information visualization systems for complex graphics and sophisticated interaction. This is why several information visualization systems still use them.

2.3 Graph Libraries

While most table-based visualization toolkits rely on an in-memory database, several graph-based visualization systems manage their data-structures using a model inspired from graph-theory where topology is the main focus and data associated with graph entities is less important. This is the case for the JUNG library [22] or the Boost Graph Library (BGL) [25], as well as for the graph library used by Cytoscape [24].

These libraries support graphs as set of vertices and edges (the topological entities) that can be associated with arbitrary data. This data is just stored by the graph entities as a convenience for the application: the library does not implement any integrity check between data and graph entities. In contrast, the InfoVis Toolkit, Prefuse and Tulip maintain a close consistency between graphs and data tables: removing a data table entry associated with a graph entity (vertex or edge) also removes the entity from the graph structure.

Thus, there is no clear consensus on how a graph data structure should be managed internally; the design choices are quite different depending on the communities such as graph theory, information visualization, database and semantic web.

2.4 Standardization Processes

Standardization is a well established habit in the software community; several standardization models have been used in the past and these models tend to evolve due to the growing pace of software development taking place nowadays.

Standards have been specified by national and international organization such as the International Organization for Standardization (e.g. ISO, ASCII), non-profit organizations (e.g. the Unicode Consortium or OMG), consortia of public or private organizations (e.g. the World Wide Web Consortium (W3C)). Closer to the information visualization community, "The Open Geospatial Consortium (OGC)[1] is an international industry consortium of 423 companies, government agencies and universities participating in a consensus process to develop publicly available interface standards."

[JDF: More to come..]

3 Design Overview

Beyond proposing a unifying design, perhaps the most novel approach of Obvious is the *process* carried to obtain this design. The project started through a sequence of Information Visualization Infrastructure workshops [16, 9, 15], during which consensus was reached that:

1. many common traits were shared among toolkits, often in slightly incompatible ways.

2. much mundane work was needlessly repeated accross toolkits.

3. creating a unified toolkit from scratch was out of reach due to varying needs and design tradeoffs

[1] http://www.opengeospatial.org

Based on those observations, consensus was reached to try the new approach of defining a "meta-toolkit" that would allow at first sharing and implementing cross-compatible services (such as data readers), then design and implement, one by one, the components on which common consensus could be reached for a unified design.

For this reason, Obvious is organized according to the Information Visualization Reference Model in three main packages: data, visualisation and view. Additionally, it provides utility classes in the util package. Next, efforts were focused on designing a consensual data model. To this day, the data model is the most elaborated and successful part of the framework. Subsequent efforts shall focus on the next modules of the Visualization Reference Model, even though simple stubs for these packages have already been successfully designed.

Resting on these fundation modules (data, visualisation, view), some actual service packages have been developed, such as data readers, or shall be developed, such as dimension to scalar functors, to provide immediate utility to both the Obvious users and the toolkit designers.

JO SAYS: It seems to me that there were two categories of contributors to the consensus. Firstly there were the developers of existing comparatively generic vis toolkits (InfoVis Toolkit, Prefuse, Improvise). While they had different approaches to their architecture (as detailed in the previous section), they are all relatively application-agnostic. This has understandably had the main influence on the design of Obvious since they are sufficiently generic to offer wide applicability to other toolkits. Then there developers who worked with specific types of data or application (Jung, Cytoscape, LandSerf [27], Mondrian). These data/applications have particular character that shaped the design of the Obvious interface (e.g. need for robust graph handling; need for geospatial raster handling; statistical graphics). Sometimes these application areas fit well with the initial Obvious interface (e.g. JUNG graphs), but some others were a little more problematic (e.g. large geospatial rasters can be modelled as tables, but not particularly efficiently. This raises the design question as to what extent Obvious should accommodate these application areas and to what extent should they adapt their internal architectures to accommodate a common Obvious framework? I think this would lead nicely into the next section that provides details on the Obvious data model.

4 Data Model

This section describes the data model used in Obvious to represent and manipulate data structures. This model has been specified for the most part during the workshop [15] as consensus has emerged, tediously but rapidly on its central and annex features.

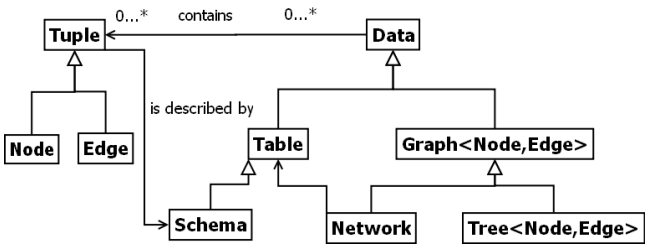

Figure 2: Class diagram of the data model

The Obvious data model is centered on the proxy tuple design pattern exposed in [20]. Obvious adopts this design pattern to offer high extendability and good usability. Among all the patterns introduced in [20], the proxy tuple pattern enables both as it encompasses graphs in an object-oriented manner - many developers are

used to manipulation of *object oriented graphs* - and as it unifies the data model around the same standard structure (tuples and tables). In our data model, tuples are standards elements of all structures: tables are composed of tuples and graphs/trees are implemented as networks i.e. graphs built around two tables one for the nodes and the other for the edges.

This model is instantiated via factories, that allows cross-toolkit interoperable data structure instantiation. 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...). This also provides 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 the source and target node columns for a graph in an edge table.

In addition to data access, our data model allows providing 3 optional, interoperable, features: introspection, modifiabilit and notification. Those features are not found in all target toolkit implementations and thus sometimes had to be emulated.

4.1 Introspection

Introspection means the capability of a program to inspect its own content. In the context of the data model it means mostly that objects expose their own schema explicitely and allow manipulating it as a full fledged object. As an improvement over [20], our data model uses a meta-circular schema design (the schema is itself a table) instead of a column object, that does not exist in Obvious. Schemas have been introduced, first because they are an efficient an elegant mean to gather all *meta-data* for the columns of a table in one unique structure, allowing easy table and network instantiations with a factory. The main use of introspection in a toolkit, though, is to enable generic implementation of a variety of side services as varied as generic persistancy, undo/redo, universal object editors...

4.2 Modifiability

Modifiability means that objects in a data model may be edited. This service is not a must-have in an information visualization context, and is actually not even considered in many of the existing toolkits: after all, in many cases, the data may be considered a static object to be analyzed. Yet new and important usage patterns for combining visualization and data editing have emerged that make this service important [5].

4.3 Notification

While most toolkit need to provide a unified notification system to propagate information about changes affecting the data model, it is also not a must-have in the context of an information visualization data-model. Stil,l notification is also an important feature to provide most advanced visualization services. During the development of Obvious implementations, notification systems included in toolkits appeared to be widely different in terms of syntax and functionalities. That is why the notification system introduced in Obvious is designed to support a large variety of existing notification techniques even those not currently implemented in toolkits. Yet, as in many toolkits [21, 14, 22, 4], the notification system in Obvious is based on listeners placed on data structures that propagates changes to listening objects. In addition, it supports transaction and batch techniques usually found in database system, as will be seen later.

4.4 Combining Notification and Modifiability

Combining Notification and Modifiability however raises a challenge. Since one operation can affect a large amout of data, flow of notifications concerning the same action could be generated. This flow can be a problem, since it can increase response time of the application.

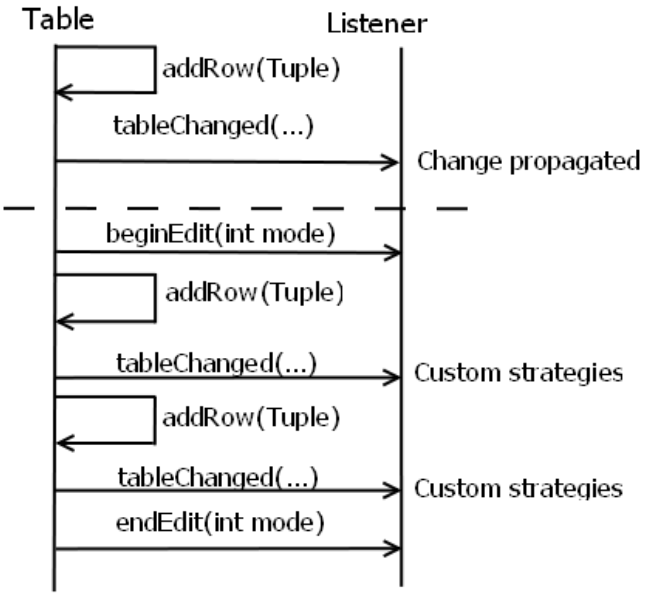

Figure 3: Sequence diagram for the notification system

Thus, we introduce a method to control the number of emitted notifications: the *beginEdit/endEdit mechanism*. As shown in the figure 3, when no mechanism is triggered the change is transmitted to the Listener with *tableChanged* method. This method have several arguments describing the current event (affected Table, rows, columns and operation type). When the *beginEdit* mechanism is enabled, changes are also propagated but different strategies can be applied. It is possible to select a specific strategy with the *mode* parameter. The following strategies have already been developed in one or more Obvious implementations:

Lazy strategy the listener do nothing until it is awake by the endEdit method

Transaction strategy the listener only commits after the *endEdit* if all data changes occuring between the *beginEdit* and *endEdit* have not violated defined invariants (for example, no null value for a specific field)

Batch strategy if the batch mode is enabled, all data changes occuring between the *beginEdit* and *endEdit* are placed in a batch, executed after the *endEdit*

4.5 Data services

To leverage our core implementation and offer some immediately useful services to Ovious users, we have defined a 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 the creation of gateways between the Obvious data model and common data formats such as CSV and GraphML. It provides 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 work 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 quick creation of 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 copying of data when using more than one data model.

5 Visualization and View models

Unlike the data model, during the workshop [15], no consensus emerged concerning the Visualization and View models. 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 workshops may address this problem.

However, we design 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 configuration of 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 the choice of an implementation toolkit to create it, then this new visualization will be compatible with all data models. Our solution does not generate extra costs of development.

JO SAYS: Can we sure about the last sentence above?. The discussion of Prefuse in 6.1 is an example of this, but requires the user of Prefuse to develop new visualization components and wrap them in Obvious. This seems like an 'extra cost of development' to me. [Pierre-Luc: Wrap them in Obvious is pretty easy (about ten lines of code). So yes, there is a tiny extra cost of development.]

An Obvious visualization works with all implementation of Obvious data structure. For existing implementations such as Obvious Prefuse or Obvious JUNG, Visualization component uses obviousx package to wrap Obvious table to native data structures of original toolkit of the Visualization. This mechanism avoids copying data from one structure to another an important point for visual analytics. However, Obvious also provides a default mechanism to quickly translate and synchronize one Obvious data structure from a targeted toolkit to an Obvious structure for another toolkit when no wrapper has been defined in obviousx for a specific toolkit.

Concerning the view, we choose to implement a simplified version of the camera pattern introduced in [20] to support standard operations such as zoom and pan. The black box concept is still present : a view simply wraps a view component of a targeted toolkit completing our monolithic pipeline for the visualization.

JO SAYS: The assertion made in this section suggests that the mono/poly-lithic approaches that vary between existing toolkits have less of an impact on data models than on view models. We should probably make some reference to this distinction in Section 2 so that it does not look like we simply ran out of time in the workshop to consider standardization of view models.

6 Implementations

This section points out encountered difficulties, raised problems and learned lessons during implementations of Obvious. Each implementation has its own particularities and allows to better identify gaps in existing design patterns and models. Thus, we start by describing 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 its design patterns are very close to those of Obvious. The binding implements all abstractions described in the core Obvious interfaces for all data model structures, visualizations, views.

Prefuse is currently the only polylithic Information Visualization toolkit used with Obvious. However, Obvious does not currently offer a visualization abstraction for polylithic components since no consensus has emerged yet. Thus, we choose to provide some prebuilt 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. Prefuse have also been used to introduce predicate support for all existing implementations of the Obvious data model. Since an Obvious tuple can easily be wrapped into a Prefuse one, the first Obvious predicate implementation is based on Prefuse predicate engine and its syntax.

This demonstrates that creating an implementation of Obvious allows other implementations to benefit of new features. It is one of the main goal of Obvious: binding toolkits to extend their capabilities.

6.2 Infovis Toolkit

Since the Infovis Toolkit is monolithic and follows the reference Information Visualization model, this implementation can realize all interfaces for the data model, visualization and view introduced in Obvious.

That is why unlike the Obvious implementation for Prefuse, to create Obvious visualization component, it is simply needed to wrap existing monolithic component of Infovis Toolkit. However, the data model of Obvious and the one of Infovis Toolkit are different: both models do not share the same object relationships -for example, in Infovis Toolkit the Graph interface is not a superinterface for the Tree interface-. In addition, data model classes are more specialized in Infovis Toolkit and do not offer the same granularity for method than in Obvious: for instance, tables can be described as static tables or dynamic tables. Thus, to implement the Obvious data model for Infovis Toolkit, some complementary code that plays with existing methods in the 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 cannot directly bind Improvise visualization components, since the toolkit does not expose its visualization pipeline publicly. 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 implementation.

Thus, this implementation shows that *coevolution* between Obvious and existing toolkits is needed to apply abstractions.

6.4 JDBC

This implementation is built to prove that Obvious can handle a large variety of data model and not only models coming from information visualization toolkit. JDBC was chosen in this purpose since databases are often used as a data source for applications and since the others toolkits we use to create implementation do not support functionalities such as transactions to test the notification model introduced formerly. Logically, this implementation only supports the data model of Obvious.

Concretely, this implementation 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. Queries used are written in "'stan-

dard"' SQL and the implementation has been tested (with tests introduced in 6.6) and several applications have been written on different DBMS such as MySQL and Oracle. In addition, for the notification system, table listeners compatible with transaction and batch strategies presented in 4.4 have been developped to validate it.

Finally, this implementation shows one interest of Obvious: its abstraction can be applied to different fields than information visualization and it is another way to extend capabilities of all Obvious implementations.

6.5 JUNG

The motivation to create an Obvious implementation based on JUNG is that others implementations concern applications and data structures agnostic toolkits (Prefuse and Infovis Toolkit). Thus, JUNG, a toolkit dedicated to graph structure, is a good candidate to test abstractions introduced in Obvious on another kind of toolkit.

Concretely, this implementation realizes all interfaces defined in Obvious, except for tables and schemas, since those notions do not exist in JUNG. Schemas are mandatory in Obvious, so this implementation uses predefined schema implementation from obvious core package. Network structures of Obvious are close enough to JUNG graph that is why the data model of JUNG was easy to wrap in Obvious. Concerning visualization and view parts, Since JUNG can easily provides monolithic visualizations, this implementation simply binds existing JUNG visualization components to Obvious ones.

This implementation was the easiest to create since Obvious and JUNG share common hypotheses: for their data model (Obvious network and JUNG graph are equivalent) and JUNG and Obvious are both compatible with the monolithic approach. 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 consensus to create unit tests for this part of the *meta-toolkit*. 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. These tests have been systematically run for each new Obvious data model development: all presented implementations successfully passed those tests.

6.7 Lessons learned

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

- the need for a specification of a clear semantic for notifications that can support simple models and more advanced ones (with batches and transactions for example)
- the need for support for a polylithic approach to define an abstraction model for visualizations
- the more abstractions defined the more unit tests we can define to validate new implementations
- toolkits need 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

JO SAYS: The last itemized point above seems like an important one, and one we should revisit in the conclusion. It exposes the issue of the degree to which existing toolkits need to adapt to allow interoperability through Obvious. It is good that we are honest about this, but it does seem to be a significant weakness with the approach, especially if we wish to support VA that use data structures that don't fit well into those modelled in Obvious (e.g. geospatial Rasters, fuzzy sets). ←

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 shows how Obvious can implement common applications in information visualization such as the creation of a scatterplot or of a graph visualization. These examples explain how to combine obvious component to build an application, how to create data structure and spot patterns to use. The first usecase concerns the coding of a graph visualization with Obvious InfoVis Toolkit implementation and the second one based on the coding of a scatterplot by combining component from different Obvious implementations.

For both examples and in fact every creation of an Obvious application, developpers have to follow the same steps:

- creation of an Obvious data structure, three ways exist:
  1. wrapping an existing data structure from a targeted toolkit as shown in first example
  2. using an Obvious reader to load an Obvious structure from a well known file format (CSV, GraphML...) as shown in the second example
  3. using Obvious methods to directly manipulate the data structure (addRow, addNode, addEdge...), this is not shown in both examples
- creation of an Obvious visualization with the created data structure and a map of parameters
  1. as shown in the second example, it is possible to combine data structure from one Obvious implementation with visualization from another Obvious implementation
  2. the map parameter allows developpers to customize the Obvious monolithic component (in the second example, the map indicates columns used for X and Y axis of the scatterplot)
- creation of an Obvious view with the created visualization

Listing 1: Visualizing a graph with Obvious

```
// Creates the graph structure.
infovis.Graph g = Algorithms.getGridGraph(10, 10);
Network network = new IvtkObviousNetwork(g);

// Creates the associated visualization using the
// factory for visualization.
Visualization vis = new IvtkVisualizationFactory()
    .createVisualization(network, null, "network", null);

// Creates the view.
View view = new IvtkObviousView(vis, null, null, null);
// Standard Java window creation
JFrame frame = new JFrame();
JScrollPane panel = new JScrollPane(
    view.getViewJComponent());
frame.add(panel);
frame.pack();
frame.setVisible(true);
```

Listing 2: Combining different Obvious implementations to display a scatterplot

```
// Defining the data factory to use,
// obvious-prefuse will be used for the data structures.
System.setProperty("obvious.DataFactory",
    "obvious.prefuse.PrefuseDataFactory");
// Creating an Obvious CSV reader and loading an
// Obvious table
CSVImport csv = new CSVImport(new File(
    "src//main//resources//articlecombinedexample.csv"),
    ",");
Table table = csv.loadTable();

// Creating the parameter map for the monolithic object.
Map<String, Object> param = new HashMap<String, Object>();
param.put(PrefuseScatterPlotViz.X_AXIS, "id"); // xfield
param.put(PrefuseScatterPlotViz.Y_AXIS, "age"); // yfield

// Creating the visualization then the view...
Visualization vis = new IvtkScatterPlotVis(table, null,
    null, param);

View view = new IvtkObviousView(vis, null, "scatterplot", null);
// Standard Java window creation
...
```

7.2 Example using Weka

Weka [18] is a suite of machine learning software widely used to design applications particularly for visual analytics [Pierre-Luc: I should put a reference here to justify this affirmation. Can you suggest one?]. Thus, Obvious in its obviousx package supports mechanisms to build Instances (main data structure of Weka) from Obvious Table. Several methods exist to link an Obvious structure with a Weka one:

- an Obvious table can be loaded into a Weka Instances. Since "Instances" is a data structure specially optimized for fast processing with clustering and machine learning algorithms, with this approach the developer benefits from Weka optimizations in terms of execution time.
- an Obvious table can wrap a Weka Instances: Obvious tables translates its methods to Weka ones. With this approach, data are not duplicated in memory.

Both methods are equivalent in terms of lines of code and can be completed with the same machine learning algorithms from Weka. For instance, to wrap an existing table into a weka Instances, developpers simply need to add the following line to the code sample 2 :

Listing 3: Wrapping an Obvious Table into Weka Instances

```
Instances inst = new ObviousWekaInstances(
    table, "Instances", new FastVector(), table.getRowCount());
```

To wrap the Obvious structure, the constructor simply needs as argument the Obvious table, a name for the weka Instances, a weka FastVector and the number of existing tuple in the table. Then, developpers can apply to this new Instances all machine learning algorithms introduced in Weka. Creating this wrapper takes less than a week.

This example demonstrates an important gain of Obvious: when a toolkit adopts Obvious it can immediately benefit from existing wrappers to complementaries functionalities. Thus, with the obviousx.weka package, all Obvious implementations can now provide advanced machine learning capabilities to their users.

7.3 EdiDuplicate (a DDupe-Like application)

INRIA maintains a database for the publication of its member. Researchers regularly fill this database, called HALINRIA, with their new publications. However, mistakes often appears during this process: duplicated authors, institutions or papers. That is why Obvious has been used to create a DDupe like application to detect duplicated authors in the database.

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.

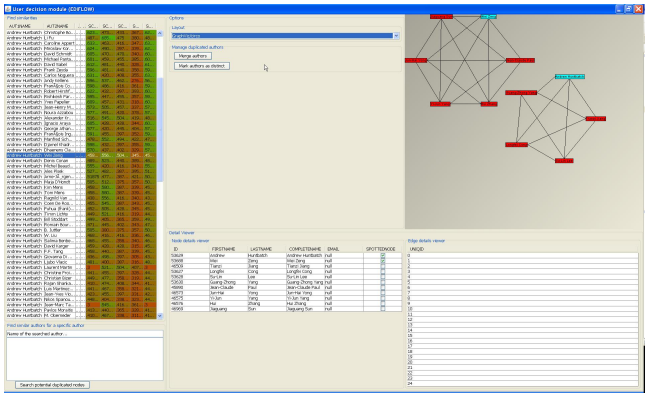

Figure 4: A screenshot of Ediduplicate

Our application EdiDuplicate offers the same possibilities but with extensions to cover specific needs. 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.

7.4 Implementing a cross-toolkit layout component

Some of us have tested a potentially important usage scenario: devising a novel layout algorithm and using the Ovious toolkit to make it available in a variety of toolkits. This layout component is a generalized treemap algorithm. Its interface makes it easy to port as this algorithm takes as input a data model and renders using a Visitor design pattern accross a renderer, making it very convenient to implement accross polyglyphic toolkits like prefuse or Discovery. Considering the current visualization model is mostly targeted at enabling monoglyphic patterns, Obvious in its current state turns out to be of limited use for our purpose.

Still, we have found that the existing data model and utilities have made developing our layout algorithm on top of Obvious worthwhile: we could implement very easily a simple monoglyphic visualization and view instances, and relying on the default data model already saves us time in the development of our prototype, while we have the assurance that only minimal work may be needed to port our method to the toolkits targeted by Obvious.

8 Future work

8.1 Extending the feature set

As said before, Obvious aims at covering all the features of an infomation visualization toolkit for which a consensual interface can be drawn. Progress in this respect is actually more advanced conceptually than in the implementation. Indeed, a few specific services have already been described that could giverise to a shared implementation:

1. value set to scalar mappings: this feature is used accross visualization definitions, to map screen coordinates, color gradients and many other visual dimensions onto data dimensions. Because the API of such a feature is somewhat small and well known, consensus appears reachable.
2. selection representation (to implement cross-toolkit brushing)
3. some interaction techniques such as zoom and pan, selection...
4. scales
5. ancilliary panels, such as object editors...

While most of those features would be of high value and we feel consensus can be reached, we have not, as of yet, proposed unifiying designs. The main reason is that while the core structure of those services is consensual, they rely on parts which are not yet consensual, such as the application bus, and more elaborated view, visualization and interaction models.

8.2 Adding new toolkits and new languages

A true test of the generalizability and unifying power Obvious would be post-hoc integration of a new toolkit. Discovery [3] is one such example being under consideration, eventhough the toolkit is proprietary toolkit, thereby limiting the potential impact of such an integration.

Another wished extension would consist in porting the toolkit to other languages and platforms, such as C++. This endeavour raises some new challenges: each language and platforms supposes some specific idioms that are hardly translatable in concepts of the other languages. Java has a generic collection type, for instance, that does not map to a standard equivalent in C++. In translating the design verbatim from a language to another, we would insure some level of compatibility, but at the expense of idosyncrasies in our library, which would preclude widespread adoption in our target languages. Conversely, adopting the target language's idioms would preclude interoperability of the Obvious platform accross languages.

8.3 Reinforcing the consensus building process

Perhaps the most novel experience we retain from the Obvious experiment is the community-driven process to reach consensus and realize a reference implementation. In many respects, this process is akin to a standardization process, only it lacks the industry incentive and backing. Like open source projects, we shall only count on voluntary contributions (aside partial of funding from public research grants), only, in the present state of toolkits, it is much more tempting to devise and expand one's own toolkit than contribute and make compromise to use a shared design which still lacks serious adoption.

We intend to experiment on various means to carry this "freeform" consensus building, to see how it can help consolidate acquired experience in the art of toolkit design. As an afterthought, we find such consolidation effort is rarely found in research domains, and yet should surely help make the field more visible and readable from an outsider perspective.

9 Conclusion

We have presented Obvious, a meta-toolkit whose goal is, ultimately, to consolidate the experience acquired by multiple toolkit designers and contributors into a single tool. Obvious provides concrete, immediate benefits to both toolkits designers and users: it dispenses the former from reimplementing various mechanisms which are outside their core concerns and helps them give visibility to their work by integrating it into a global context. It enables the later (users) to delay their choice of toolkit, be faced upfront with state of the art design patterns as well as a wealth of convenience features. The longer term benefits shall reach deeper prospects: possibly defragmenting a research area by providing bridges between tools rather than yet another, competing, toolkit. If this goal is successfully reached, then the reward shall benefit the whole infovis community, enabling it to provide its user communities an easier entry path. *note from tb: it's getting late, I'm not sure I can articulate my thoughts...*

Obvious whall remain a work in progress by design, at least in the foreseeable future. All members of the infoviz community are invited to contribute to its design, make it evolve, and possibly use its features. While Obvious may not yet be at a stage where it can be diffused much beyond, this shall come soon. We hope it will prove a valuable service to the infoviz community.

Acknowledgements

The authors wish to thanks the participants of the VisMaster Workshop on Visual Analytics Software Architecture, held Dec. 4-6th 2008 in Paris: Fanny Chevalier, Christophe Favart, Jeffrey Heer, Joshua O'Madadhain, Harald Piringer, Danyel Fisher, Giueseppe Santucci, Mike Smoot, Martin Theus, and Chris Weaver. This project has been partially funded by the European FET-Open Coordination Action project VisMaster[2].

References

[1] D. Auber. Tulip : A huge graph visualisation framework. In P. Mutzel and M. Jünger, editors, *Graph Drawing Softwares*, Mathematics and Visualization, pages 105–126. Springer-Verlag, 2003. 1, 2

[2] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *International AAAI Conference on Weblogs and Social Media*, 2009. 2

[3] T. Baudel. Visualisations compactes: une approche declarative pour la visualisation d'information. In *Proceedings of the 14th French-speaking conference on Human-computer interaction (Conference*

[2] http://www.vismaster.eu

Francophone sur l'Interaction Homme-Machine), IHM '02, pages 161–168, New York, NY, USA, 2002. ACM. 1, 2, 8

[4] T. Baudel. Browsing through an information visualization design space. In *CHI '04 extended abstracts on Human factors in computing systems*, CHI EA '04, pages 765–766, New York, NY, USA, 2004. ACM. 2, 4

[5] T. Baudel. From information visualization to direct manipulation: extending a generic visualization framework for the interactive editing of large datasets. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*, UIST '06, pages 67–76, New York, NY, USA, 2006. ACM. 2, 4

[6] B. B. Bederson, J. Grosjean, and J. Meyer. Toolkit design for interactive structured graphics. *IEEE Trans. Softw. Eng.*, 30:535–546, August 2004. 2

[7] N. G. Belmonte. Javascript infovis toolkit, Mar. 2011. 1

[8] A. Bezerianos, P. Dragicevic, J.-D. Fekete, J. Bae, and B. Watson. Geneaquilts: A system for exploring large genealogies. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1073–1081, Nov-Dec 2010. 2

[9] K. Borner, B. Herr, and J.-D. Fekete. 2007 workshop on information visualization software infrastructures. https://nwb.slis.indiana.edu/events/ivsi2007/, July 2007. 3

[10] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15:1121–1128, November 2009. 1

[11] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 1

[12] E. H.-h. Chi and J. Riedl. An operator interaction framework for visualization systems. In *Proceedings of the 1998 IEEE Symposium on Information Visualization*, pages 63–70, Washington, DC, USA, 1998. IEEE Computer Society. 1

[13] S. G. Eick. Visual discovery and analysis. *IEEE Transactions on Visualization and Computer Graphics*, 6:44–58, 2000. 1

[14] J.-D. Fekete. The infovis toolkit. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 167–174, Washington, DC, USA, 2004. IEEE Computer Society. 1, 2, 4

[15] J.-D. Fekete. Vismaster workshop on visual analytics software architecture. http://code.google.com/p/obvious/wiki/Motivation, December 2008. 3, 5

[16] J.-D. Fekete and K. Borner. 2004 workshop on information visualization software infrastructures. http://vw.indiana.edu/ivsi2004/, November 2004. 3

[17] T. Gonzales and M. VanDaniker. Axiis open source data visualization. web site, 2009-2011. 1

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009. 7

[19] J. Heer. flare data visualization for the web, Mar. 2011. 1

[20] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12:853–860, September 2006. 1, 2, 3, 4, 5

[21] J. Heer, S. K. Card, and J. A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '05, pages 421–430, New York, NY, USA, 2005. ACM. 1, 2, 4

[22] J. O'Madadhain, D. Fisher, S. White, and Y.-B. Boey. The jung (java universal graph/network) framework. Technical report, UCI-ICS, 2003. 1, 3, 4

[23] C. Plaisant, J. Grosjean, and B. B. Bederson. Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, INFOVIS '02, pages 57–, Washington, DC, USA, 2002. IEEE Computer Society. 2

[24] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504, Nov. 2003. 2, 3

[25] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002. 3

[26] C. Weaver. Building highly-coordinated visualizations in improvise. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 159–166, Washington, DC, USA, 2004. IEEE Computer Society. 1, 2

[27] J. Wood. Terrain parameterization in LandSerf. In T. Hengl and H. Reuter, editors, *Geomorphometry: Concepts, Software and Applications*, volume Ch. 14, pages 333–349. Elsevier, London, 2008. 3