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 toolkit, 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 [15, 23, 26, 31, 3], C++ [1, 14], Flash/Flex [21, 19] or JavaScript/HTML5 [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 dispersion in terms of capabilities since each toolkit has unique and useful techniques for visualization and interaction. For example, the Prefuse [23] and JUNG [26] toolkits offer several graph layout algorithms whereas Improvise [31] 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 re-implementing 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 main benefits offered by Obvious are:

1. it defers the choice of concrete toolkits to a later stage of the VA development;
2. it enforces a better separation of concerns in VA applications so that the data models can be specified independently of the visualizations and views;
3. it improves the interoperability of code, data models and visualizations;
4. it improves the reusability of code and components;
5. clarifies issues with notification and allows scaling up++;
6. it specifies a stable vocabulary which simplifies learning.

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

*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

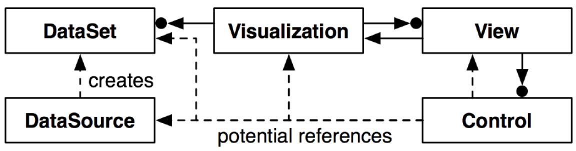

Figure 1: The Information Visualization Reference Model [22]

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 [22]. The model defines three stages: *DataSet* or *Data Tables*, *Visualization* or *Visual Structure* 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 [15] 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 [23] 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 structure 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 [23], Heer and Agrawala [22] have derived software design patterns that are common to information visualization applications and toolkits.

Improvise [31] 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 alternatives. Combining the best possible features for a next-generation toolkit might be tempting but there are still tradeoffs 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 tradeoff 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. [27, 8].) An early version of Piccolo has also been used as graphics engine for the Cytoscape graph visualization system [28] 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 attributes and even geometry from data attributes. For example, the position of an item using a scatter-plot visualization is computed using a simple affine transformation the data attributes using for the X and Y dimensions. 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 terms 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 [26] or the Boost Graph Library (BGL) [29], as well as for the graph library used by Cytoscape [28].

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 accelerating pace of software development taking place nowadays.

According to Wikipedia: "The goals of standardization can be to help with independence of single suppliers (commoditization), compatibility, interoperability, safety, repeatability, or quality." The goals raised in this article are well among them: compatibility, interoperability, and quality.

Standardization follow roughly four models:

1. Specified by national and international organization such as the International Organization for Standardization (e.g. ISO, ASCII),

2. Specified by a private or public consortium (e.g. the Unicode Consortium, the OMG, the World Wide Web Consortium (W3C). Closer to the information visualization community, "The Open Geospatial Consortium (OGC)[1], which is an international industry consortium of companies, government agencies and universities participating in a consensus process to develop publicly available interface standards for geospatial data.

3. Community-driven: looser groups can be faster and allow for more experiments than formal standardization bodies or consortia. Communities, such as the *Boost Community*[2] — designing libraries for the ISO C++ language — experiment, develop and document software that sometimes become part of formal ISO standards. The *Java Community Process*[3] plays a similar role for the Java language and programming environment.

   Standard specified by established organizations go through a formal process that take substantial time; usually years. On the other side, ad-hoc organizations such as consortia can issue standards or recommendations faster. In particular, the W3C or community-drive consortia define stages for their "recommendations" (the name for standard issued by the W3C) before they are considered final. In all cases, these organizations establish steering committees to control the processes and require substantial involvements from many organizations to achieve standards.

4. *De facto*: at the other extreme are the application domains where one system becomes the standard. For example, in Scientific Visualization, VTK [24] has become, in the latest years, the *de facto* standard toolkit: it is used by researchers and practitioners, and newer solutions are getting integrated quickly into VTK. This is possible when either a system reaches a certain level of popularity — such as Microsoft Word for Word Processors — or when the quality and features of the system are unmatched — such as VTK.

At the current stage, Information Visualization toolkits are not understood well-enough to start a formal standardization process. It seems that one toolkit will not become a *de facto* standard due to their stretch in scope and capabilities. Two models remain: consortium-driven or community driven. Obvious has started as a community-driven initiative. It is up to the community to decide how it wants to coordinate its software development for better compatibility, interoperability and quality.

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 [17, 9, 16], 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 across toolkits,
3. creating a unified toolkit from scratch was out of reach due to varying needs and design tradeoffs.

Based on those observations, an attempt was made for a new approach of defining a "meta-toolkit" that would allow 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, visualization and view. Additionally, it provides utility classes in

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

[2] http://www.boost.org

[3] http://www.jcp.org/

the "util" package. Next, efforts were focused on designing a consensual data model. For now, the data model is the most elaborated and successful part of the framework.

Resting on these foundation modules (data, visualization, view), some actual service packages have been developed, such as data readers, writers, conversions to provide immediate utility to both the Obvious users and the toolkit designers.

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 [16] as consensus has emerged, tediously but rapidly on its central and annex features.

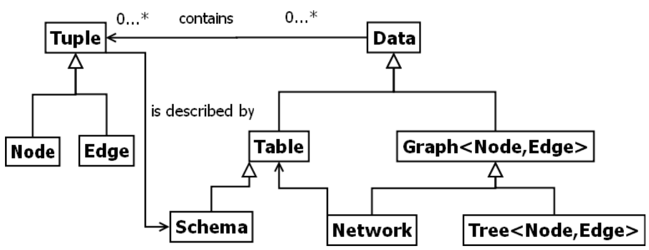

Figure 2: Class diagram of the data model

The Obvious data model is centered on the proxy tuple design pattern exposed in [22]. Obvious adopts this design pattern to offer high extensibility and good usability. Among all the patterns introduced in [22], 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 allow 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*, *batch editing* 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 explicitly and allow manipulating it as a full-fledged object. As an improvement over [22], 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 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 persistence, undo/redo, and universal object editors.

4.2 Batch Editing

Batch editing means that one or many cells in a data model may be edited at the same time. This happens when the toolkit manages analytical columns (e.g. computing the centrality of each vertex in a network), with selection and dynamic queries if their effect is reported to a data column or simply if a user wants to change values interactively, either in the data table or through a visualization by direct manipulation [5].

4.3 Notification

All the popular information visualization toolkits (e.g. [23, 15, 26, 4]) implement notification using the "Observer" pattern from [18] to propagate information about changes affecting the data model. This pattern specifies two roles: Observable and Observer; in our case, data models are Observables meaning that the allow Observers to register and be notified when they are changed. During the design of Obvious, we realized there were some variants in the way toolkits implemented this pattern. This is why the notification system introduced in Obvious is designed to support a wide variety of notification models, even those not currently implemented in current toolkits but that will be required to scale. The notification system in Obvious is also based on the Observer design pattern with extensions to supports transaction and batch techniques usually found in database system.

Combining Notification and batch editing raises a challenge: since one operation can affect a large amount of data, a flow of notifications concerning the same action will be generated. If each change is managed in isolation, the application can spend a large amount of time updating visual structures, e.g. recomputing a layout for each modified item. This typically leads to the application being unresponsive for a long time.

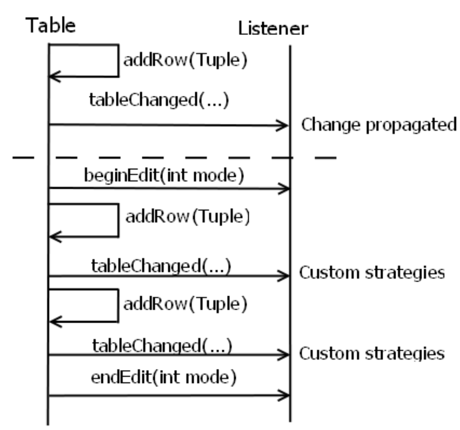

Figure 3: Sequence diagram for the notification system

Thus, Obvious introduces a method to control the management of batch notifications: the *beginEdit/endEdit* mechanism. Figure 3 shows the sequence diagrams of the notification manager. Each time a data table is changed, the change is transmitted to the Observer with a call to the *tableChanged* method. This method takes several arguments describing the current event (affected Table, rows, columns and operation type): this is the typical Observer pattern.

When the *beginEdit* method is called on an Obvious data table, the Observer's *beginEdit* method is called to start a batch editing transaction. Different strategies can be applied by the observer. A *mode* parameter allows the observer to select a specific strategy depending on the type of transaction (atomic or batched). Note that the observer will still receive a tableChanged call for each tuple modified. The observer is in charge of implementing a strategy to optimize atomic/batch edition. If it does not, the standard behavior will happen and batch editing will flood the observer — which is acceptable in some cases. The following strategies have already been developed in one or more Obvious implementations:

Lazy strategy after a beginEdit, the observer ignores all the tableChanged calls until the endEdit method is called; then, the observer's actions are performed e.g. a layout is recomputed.

Batch strategy after beginEdit, the observer buffers the information sent by tableChanged. When endEdit is called, the actions are performed on each of the buffered items. Note that buffering can be complicated when items are created, deleted or changed many times. The burden is left to the observer since this management can be heavily optimized depending on the action to perform.

Transaction strategy after beginEdit, the observer buffers the information sent by tableChanged. When endEdit is called, it first checks structural invariants (for example, no null value for a specific field) before performing its actions on the modified items.

The *beginEdit/endEdit* mechanism has been added to support batch editing and also database transactions. When the data model implementation relies on a transactional database, atomic and batch transactions will occur and notifications (e.g. implemented as database triggers) will arrive in batches. The semantic of Obvious notification handles this case correctly but the observer should be aware that the tableChanged method can be called much later than when the table is actually changed. This is the case for database atomic transactions: the actual notification is propagated after the end of the transaction when the database engine has done all the integrity checks.

In practice, the three strategies we have described have been sufficient so far to handle all the cases required inside toolkits. The impact on performance can be substantial, in particular to manage dynamic data, a very standard situation in visual analytics applications that has not been well addressed in information visualization so far.

4.4 Other services

To leverage our core implementation and offer some immediately useful services to Obvious users, we have defined a utility package "obviousx", named in the same way as the Java extension package "javax". This package provides different kinds of utility classes for the Obvious data model. First, we have defined 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 provides compatibility classes to use with standard Java components such as a Java Table Model that allows the creation of a JTable from an Obvious table. Finally, obviousx also provides wrappers to *map* obvious data structures into common existing data structures (e.g. for Prefuse, the InfoVis Toolkit, and Jung) to share data structures when using more than one data model.

5 Visualization and View models

Unlike the data model, no consensus emerged concerning the Visualization and View models during the workshop [16]; the main reason being the different approaches chosen among toolkits. One important issue is the monolithic vs. polylithic approach. Another one is related to tables vs. objects: some toolkits keep the visualization data in tables (e.g. Prefuse, the InfoVis Toolkit and Tulip) whereas others create objects for displaying (e.g. Improvise, Cytoscape) or nothing at all when there is a pipeline as in Discovery. So, more work is needed to design the abstractions required to wrap the different implementations. Further discussions and workshops will address the problem.

Still, Obvious provides a solution: it wraps visualizations into a black box with a small set of methods and, for the creation of these visualizations, it relies on a *Factory* design pattern [18]. For example, creating a scatter-plot visualization from a data table requires the following lines:

Listing 1: Creating a visualization using a Factory

```
Map params = new HashMap();
params.put("x", "id");
params.put("y", "age");
Visualization vis = VisualizationFactory.getInstance()
    .createVisualization(table, null, "scatterplot", params);
```

The variable "param" contains parameters to configure the visualization, here to specify which attribute will be used for the X and Y axes.

With this mechanism, it looks as if Obvious were a monolithic toolkit but the actual implementation of the Obvious wrapper for a polylithic toolkit can easily translate a monolithic specification into a dedicated configuration for the underlying polylithic component. The code above will work for the Prefuse toolkit and return a polylithic component wrapped as an Obvious visualization and configured as a scatter-plot visualization.

If a developer needs a visualization component that does not exist in the default implementation (e.g. an InfoVis Toolkit time-series with a Prefuse-wrapped data table), the visualization can be created directly from a specified factory or from the Obvious class:

```
new IvtkTimeSeriesVis(table, null, "timeseries", params)
```

An Obvious visualization works with any Obvious data model. The data model will be either wrapped to become compatible with the native model if the underlying implementations are different or unwrapped when the visualization and the data model are from the same implementation (e.g. Prefuse).

This mechanism avoids copying data from one structure to another, which is a crucial point for visual analytics. Alternatively, Obvious also provides a default mechanism to quickly copy and synchronize data models when no wrapper has been defined for a specific toolkit. Current wrappers are lightweight, adding very little overhead to the system.

At this point, the application developer can choose one of the existing visualizations from one of the wrapped toolkits or decide to create a new one which can derive from one of the wrapped toolkits or be implemented from scratch. Obvious substantially increases the number of possible visualizations and toolkits to use and does not limit the developer in any way at this stage.

A *View* is simply specified as a black box implementing a simplified version of the camera pattern introduced in [22] to support standard operations such as zoom and pan. Like the visualization interface, future workshops should enrich it when a more consensuses are reached.

6 Implementations

This section describes the implementation of Obvious and points the lessons learned during implementations binding Obvious interfaces to wrappers around concrete toolkits. Each toolkit has its own design choices that are discussed in articles but some of the implications came to light when implementing the bindings, for example differences of interpretations of design patterns. We briefly describe the most important lessons here.

The core of Obvious is a small (40Kb) Java project[4] managed by Maven [13] to facilitate its deployment. It consists in 44 java files, 23 interfaces and 21 utility classes organized in 5 top-level packages: data, viz, view, util and impl. It has been designed to be small and lightweight. Therefore, the substantial parts are in each concrete toolkit binding.

6.1 Prefuse

Prefuse was the first binding implemented because its architecture is, by design, very close to Obvious. The binding implements all the abstractions described in the core Obvious interfaces for all data models, visualizations and views.

For the visualization, Prefuse is currently the only polylithic Information Visualization toolkit with a binding for Obvious. As explained in 5, Obvious does not offer a visualization abstraction for polylithic components. Thus, Obvious provides components preconfigured for well know visualization techniques such as scatterplots or force directed graphs. Currently, if a software developer wants visualize an Obvious table using a Prefuse visualization not offered in the Obvious visualization factory, the only requirement is to convert the data model to a Prefuse data table using an obviousx component.

Several interfaces defined by Obvious are based on a Prefuse concrete class. Therefore, Prefuse was used as a complete implementation of Obvious to check its model and syntax.

This binding is composed of 19 classes in three packages (data, viz and view). The size of the resulting JAR is 120KB and the one of the Prefuse core is 3.3 MB.

6.2 InfoVis Toolkit

Since the InfoVis Toolkit is monolithic and follows the reference Information Visualization model, its Obvious binding realizes all the interfaces for the data model, visualization and view introduced in Obvious. The InfoVis Toolkit has monolithic visualizations, providing them simply consists in wrapping an Obvious visualization around them class and implementing a factory to create them by name.

However, the data model of the InfoVis Toolkit differs from Obvious for trees and networks: in the InfoVis Toolkit, the Graph interface is not a super-interface of the Tree interface. In addition, some data model classes are more specialized in the InfoVis Toolkit than in Obvious. For example, tables can be described as static tables or dynamic tables. The binding was therefore complicated by these mismatches that needed a more complicated code than for Prefuse.

Nevertheless, the InfoVis Toolkit binding is operational and reliable. This binding is composed of 12 classes in three packages (data, viz and view). The size of the resulting JAR is 115KB and the one of the Infovis toolkit core is 10MB.

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, Obvious cannot directly bind Improvise visualization components because the toolkit does not expose its visualization pipeline publicly: Improvise components are intended to be complete black boxes. Addressing this problem would require some changes inside the current version of Improvise. In addition, Improvise does not support well dynamic data, which is a functionality intended to be in every obvious implementation.

Currently, Improvise can use a data table from an Obvious data table but the rest of the Improvise pipeline is hidden from Obvious. Providing a complete binding for Improvise in Obvious would require some changes in Improvise.

[4] `http://code.google.com/p/obvious`

6.4 JDBC

JDBC is the standard Java interface to standard SQL databases. We wrapped JDBC in an Obvious data table to prove that Obvious can support a large variety of data model, not only models coming from information visualization toolkits. JDBC was chosen because databases are frequently used as data sources for applications and since JDBC provides additional features not available in the toolkits data tables such as atomic and batched transactions. We used it to test the notification model introduced formerly. As expected, this implementation only supports the data model of Obvious.

Concretely, this implementation translates obvious methods into SQL queries. For example, the data table “get” methods are implemented as SELECT queries, the “set” methods as UPDATE queries, the “add” methods as INSERT queries, and the “remove” methods as DELETE queries. Queries are written in standard SQL and several applications have been written to work with different DBMS such as MySQL and Oracle. In addition, for the notification system, table listeners compatible with transaction and batch strategies presented in 4.3 have been developed and help validate the Obvious notification model.

6.5 JUNG

JUNG is a graph library in Java that mainly manages the graph topology but associates arbitrary attributes with vertices and edges. Concretely, this implementation realizes all interfaces defined in Obvious, except for tables and schemas since these notions do not exist in JUNG. Schemas are mandatory in Obvious, so this implementation uses a default schema implementation from the obvious core package. The network structure of Obvious is similar to JUNG’s graph; therefore, the data model of JUNG was easy to wrap as an Obvious Network. Concerning the visualizations and views, JUNG provides monolithic visualizations. The Obvious implementation simply binds existing JUNG visualization components to Obvious visualizations.

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.

6.6 Units tests

Obvious is specified using Java interfaces and some comments in the implementation files but without any formal specification of the precise behavior of the defined interfaces. To verify that all the implementations behave correctly and consistently, we have implemented *Unit Tests*: a suite of classes and methods aimed at testing all the methods of all the classes.

Currently, the tests are only defined on the data model of Obvious. The level of specification of Obvious visualizations and views is not sufficient to perform useful tests.

Unit tests need an implementation to work; they cannot test abstract classes or interfaces. Due to the similarities of Obvious and Prefuse, the Prefuse binding has been used to set up the unit tests for the data model of Obvious. They have then been moved to the core Obvious module to be usable by all the bindings. 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 in their own module to perform more advanced ones for their binding.

Concretely, unit tests have been defined with JUnit [30] for the following interfaces: Schema (14 tests), Table (11 tests), Network (13 tests) and Tree (8 tests); all part of the Obvious core package. These tests have been systematically run for each new Obvious data model development: all presented implementations successfully passed those tests.

With the exception of Improvise, all the toolkits we wanted to support are now available with Obvious. The extra code required for binding a concrete toolkit is quite modest compared to the toolkit itself and the overhead in time has always been negligible. Table 1 summarizes the footprint of the implemented Obvious modules and the services they support. At this point, Obvious is usable with a wide range of concrete implementations and we report on its uses in the next section.

Bindings	obvious.data	obvious.vis	obvious.view	Size
Prefuse	10 classes *1235*	7 classes *446*	2 classes *94*	120 kB
InfoVis Toolkit	6 classes *1404*	5 classes *460*	1 class *105*	115 kB
Improvise	3 classes *519*			27 kB
JDBC	8 classes *1936*			75 kB
JUNG	3 classes *714*	1 class *139*	2 classes *114*	93 kB

Table 1: Footprint of Obvious bindings. Numbers in italics refer to the number of lines of Java code.

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 [15] 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 [23] – 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 scatter-plot or of a network visualization. These examples explain how to combine Obvious components to build an application, how to create data structure and spot patterns to use. The first use-case concerns the coding of a network visualization with the Obvious-InfoVis Toolkit implementation and the second based on the coding of a scatter-plot by combining component from different Obvious implementations.

For both examples and more generally for every creation of an Obvious application, developers have to follow the following steps:

- Creation of an Obvious data structure, either directly with a standard constructor or through a factory. Three ways exist to fill the data structure:
    1. wrapping an existing data structure from a targeted toolkit as shown in the 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...); an example would be too long for this article.
- Creation of an Obvious visualization from the created data structure and additional parameters. This can be done directly with a class constructor or through a factory. The parameters allow customization of the Obvious monolithic components. As shown in the second example, it is possible to use the data structure from one Obvious implementation with a visualization from another.
- Creation of an Obvious view with the created visualization directly with a constructor or through a factory

Listing 2: Visualizing a graph with Obvious

```
// Creates the graph structure. First, set the factory to use (ivtk).
// Then load the native data structure, and get a factory instance.
// Finally, call the convenient getter of the factory.
System.setProperty("obvious.DataFactory",
    "obvious.ivtk.data.IvtkDataFactory");
infovis.Graph g = Algorithms.getGridGraph(10, 10);
DataFactory factory = DataFactory.getInstance()
Network network = factory.createGraph(g);

// Creates the associated visualization using the
// factory for visualization. No predicate and extra
// parameters are given to the constructor.
Visualization vis = new IvtkVisualizationFactory()
    .createVisualization(network, null, "network", null);

// Creates the view. No predicates and extra parameters are given to
// the constructor.
View view = new IvtkObviousView(vis, null, "graphview", null);
// Standard Java window creation
JFrame frame = new JFrame();
JScrollPane panel = new JScrollPane(view.getViewJComponent());
frame.add(panel);
frame.pack();
frame.setVisible(true);
```

Listing 3: Combining different Obvious implementations to display a scatter-plot

```
// 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("example.csv"), ',');
Table table = csv.loadTable();

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

// Creating the visualization then the view. No predicates are given to
// the constructor.
Visualization vis = new IvtkScatterPlotVis(table, null, "plot", param);

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

7.2 Integration of Weka

Weka [20] is a suite of machine-learning algorithms and data structures widely used to design machine-learning applications. The obviousx package of Obvious supports two mechanisms to build the main data structure of Weka (called “Instances”) from an Obvious Table:

- an Obvious table can be copied into a Weka “Instances”, which is a data structure specially optimized for fast processing with clustering and machine learning algorithms. With this approach, running-time is optimized.

- an Obvious table can wrap a Weka "Instances": the Obvious table translates its methods into the Weka equivalents. With this approach, memory-consumption is optimized.

Both methods are equivalent in terms of lines of code and can be applied to the same machine learning algorithms from Weka. For example, wrapping the table from the code sample 3 into a Weka "Instances" requires the following line:

Listing 4: Wrapping an Obvious Table into Weka Instances

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

This "Instances" can be used by all the machine-learning algorithms defined in Weka. Creating this wrapper took about three days to one developer who knew Obvious well but was discovering Weka.

This example demonstrates an important gain of Obvious: a toolkit with a binding in Obvious can immediately benefit from a substantial set of additional features, such as Weka for advanced machine-learning capabilities and several format converters. Conversely, developers of new analysis algorithms could port them to use Obvious data structures to become usable by a substantial number of toolkits and application programmers to build visual analytics systems.

7.3 EdiDuplicate

EdiDuplicate is a system built with Obvious, designed to detect and merge duplicated entities in the HAL publication database. It is an adaptation of the D-Dupe software [25] written in .NET. To help resolve the entities, two types of information are visualized: the similarity measures between entities to resolve and the social network of these entities, as shown in Figure 4.

Each time a new entity is created in the HAL database, a module computes multiple similarity metrics from this entity to all the ones already in the database. This information is loaded on an Obvious table and displayed on the left pane using a standard Java table. Each row refers to one pair of names and the columns contain the multiple similarity measures with a green-red color coding; the table that can then be sorted according to any column order.

When pair is selected by clicking on a row, a network view is created on the right visualizing the neighborhood network of the pair of entities. This neighborhood is computed from publication data: for a target entity, it contains all the entities already connected to it through co-authorship relations. This information helps the user decide if the pair of entities has to be merged.

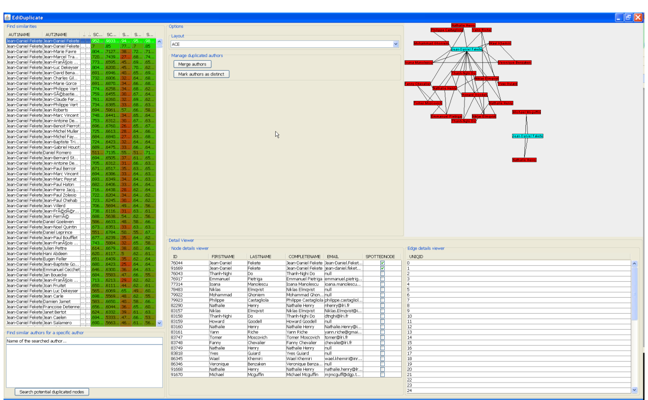

Figure 4: The EdiDuplicate application

EdiDuplicate offers the same features than D-Dupe but with extensions to cover needs specific the INRIA-HAL database[5]. The application mainly combines Obvious components and Swing components (derived from Obvious structures). For the data model, an Obvious Network is used using the InfoVis Toolkit implementation of Obvious; the visualization and the view parts are also provided by this implementation. Building this application took less than a week.

[5] http://hal.inria.fr

7.4 DBMS Caching Data Table

[JDF: Add description of EdiFlow cached data manager]

7.5 Large Wall Network Visualization

[JDF: Add description of implementing the visualization from scratch]

7.6 Implementing a Cross-Toolkit Layout Component

We have also tested a more advanced usage scenario: devising a novel layout algorithm and using the Obvious 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 to a renderer object, making it very convenient to implement across polylithic toolkits such as Prefuse or Discovery. Considering that the current visualization model is mostly targeted at enabling monolithic 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 monolithic 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.

7.7 Conclusion

The examples described in this section assess an important strength of Obvious: it allows a clear separation of concerns in the development of visual analytics applications with a small memory and performance footprint. The data model of the application can be chosen independently from the visualization components as long as all these components fit the Obvious model. From our experience, a large number of the visual analytics application fit the Obvious model and they will benefit from the meta-toolkit in term of richness and extensibility.

8 Future Work

8.1 Extending Obvious Supported Features

Obvious aims at covering all the features of an infomation visualization toolkit for which a consensual interface can be specified. Progress in this respect is actually more advanced conceptually than in the implementations. A few specific services have already been described that could give rise to a shared implementation:

1. value set to scalar mappings: this feature is used across 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. ancillary 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 unifying 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 Additional Toolkits and Languages

A true test of how generalizable and unifying Obvious would be post-hoc integration of a new toolkit. Discovery [3] is one such example being under consideration, even though 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 endeavor 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 idiosyncrasies 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 across languages.

8.3 Obvious and Other Visualizations

8.4 Community Building

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 re-implementing 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 information visualization 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 shall remain a work in progress by design, at least in the foreseeable future. All members of the visual analytics 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 visual analytics 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[6].

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 Francophone sur l'Interaction Homme-Machine)*, IHM '02, pages 161–168, New York, NY, USA, 2002. ACM. 1, 2, 9

[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. 2

[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. 2

[13] S. Company. *Maven: The Definitive Guide*. O'Reilly Media, Inc., 1 edition, Oct. 2008. 6

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

[15] 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, 7

[16] J.-D. Fekete. Vismaster workshop on visual analytics software architecture. http://code.google.com/p/obvious/wiki/Motivation, December 2008. 3, 4, 5

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

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

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. 4, 5

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

[20] 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

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

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

[23] 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, 7

[24] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-15-7, http://www.itk.org/ItkSoftwareGuide.pdf, second edition, 2005. 3

[25] H. Kang, L. Getoor, B. Shneiderman, M. Bilgic, and L. Licamele. Interactive Entity Resolution in Relational Data: A Visual Analytic Tool and Its Evaluation. *IEEE Transactions on Visualization and Computer Graphics*, 14:999–1014, 2008. 8

[26] 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

[27] 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

[28] 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

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

[30] P. D. Stotts, M. Lindsey, and A. Antley. An Informal Formal Method for Systematic JUnit Test Case Generation. In D. Wells and L. A. Williams, editors, *XP/Agile Universe*, volume 2418 of *Lecture Notes in Computer Science*, pages 131–143. Springer, 2002. 6

[31] 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