Visualizations Everywhere: A Multiplatform Infrastructure for Linked Visualizations

Danyel Fisher, Steven M. Drucker, Roland Fernandez, and Scott Ruble

Figure 1: The WebCharts framework accommodates multiple types of visualizations, embedded in multiple types of applications. In this figure, four different host applications (around the outside) each dynamically add any WebCharts enabled visualization available on the web (center).

Abstract—In order to use new visualizations, most toolkits require application developers to rebuild their applications and distribute new versions to users. The WebCharts Framework take a different approach by hosting Javascript from within an application and providing a standard data and events interchange.. In this way, applications can be extended dynamically, with a wide variety of visualizations. We discuss the benefits of this architectural approach, contrast it to existing techniques, and give a variety of examples and extensions of the basic system.

Index Terms—Visualization systems, toolkit design, data transformation and representation.

1 Introduction

New visualization techniques are constantly being invented: for example, the InfoVis 2009 conference featured over twenty novel visualizations that addressed established problems and introduced new domain-specific opportunities. We can divide the potential users of these visualizations into three (often overlapping) audiences: end-users who want to apply the visualizations to their own data; visualization writers who wish to create new visualizations, and application developers that want to enable others to use the new visualizations with their applications. End-users often have no choice but to become adept at moving data between applications or wait for updates to their favourite applications that may or may not include the new capabilities. Visualization developers often need to create the visualization from scratch or they may build upon visualization toolkits (such as Prefuse [12] or the Information Visualization Toolkit [9]). In order to use the toolkit, they are forced to use the language and data structures that the toolkit requires and their visualization, in turn, will only be useful to people who are using those toolkits. Finally, application developers, hoping to incorporate one of the visualizations themselves, either must adopt the toolkits or translate the visualization code into their own data structures and rendering systems.

We have chosen a different route, which is to add visualizations to applications dynamically, without requiring recompilation of the existing application. The WebCharts framework enables this functionality by allowing applications to host arbitrary visualizations. Any application that uses this infrastructure can add any compatible visualization, dynamically and at runtime. The application developer does not need to know the types of visualizations that might be applied to their application; the visualization designer does not need to know what applications might serve as host.

As a result, application developers now have a much easier task: they must build infrastructure only once, in the language of their choice, and add a mechanism to select particular visualizations. Visualization developers write their visualization, using their favorite visualization framework or toolkit, and add support for the WebCharts interface, and post them on the internet. Other developers can even write a thin WebChart adapter to translate existing visualizations into our interface. Finally, end-users can use the WebCharts-enabled application of their choice without waiting for an update that incorporates the latest visualizations.

WebCharts takes advantage of the growing popularity of Javascript as a language to communicate between components; and of the ability to embed web browsers within client applications, to support visualizations in rich client applications. The host application provides a generic drawing surface, and translates both tabular data and commands from the host application into a standard Javascript

- *Danyel Fisher, Steven Drucker, and Roland Fernandez are with Microsoft Research. {danyelf, sdrucker, rfernand}@microsoft.com*
- *Scott Ruble is with Microsoft Corporation. sruble@microsoft.com*

Manuscript received 31 March 2010; accepted 1 August 2010; posted online 24 October 2010; mailed on 16 October 2010.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

format. The charting client, written in a web-based language such as Silverlight, Flash, or Javascript, responds to these commands, and renders the visualization. By storing the web-based code locally, visualizations can be generated offline, without web access.

This paper discusses the WebCharts framework. It looks at the problem of extensibility and different ways that it has been approached. It then discusses the system design of WebCharts. Last, it provides several use cases in different environments, and hosting a broad variety of visualizations.

2 Extensibility for Visualizations

WebCharts allows users to assemble visualization components within an application, using their data from that application. Tools that support a set of different visualizations are common: tools like Tableau [21] and Spotfire [20] and Dashiki [17] allow users to assemble dashboards using a fixed suite of visualizations. These applications provide mechanisms for importing data, choosing visualization from a fixed set, aggregating values, and assigning data columns to aspects of the visualization. These tools do not, however, accommodate extensibility. There have been a variety of research and commercial projects meant to accommodate the developing needs of data visualization. In this section, we first outline a variety of approaches to visualization extensibility, and then compare our approach to other web-based architectures.

2.1 Extensible Toolkits

A number of visualization toolkits have been developed to help developers more rapidly create visualizations. Most toolkits support building visualizations that are compiled into applications. For example, Pad++ [3] and later Piccolo & Jazz [4][5] provide libraries for common tasks such as zooming and object models. The Infovis Toolkit (IVTK) [9], Borner's XML Infovis Toolkit [1], and Prefuse [12] are Java toolkits that provide base classes and package several common visualizations. All of these systems offer additional libraries that help the application collect or manage data, communicate with external sources, or take care of common tasks. Flare [10], written in Actionscript, provides similar functionality. In contrast to these systems, the WebCharts framework encourages developers to create visualizations in whatever system they are most comfortable, with the caveat that it must be embeddable within a web browser.

Other visualization systems, such as Processing [19] and Protovis [6], are primarily languages for expressing visualizations, without a containing application. Protovis is written in Javascript, and so is optimized for deployment on the internet. Protovis, given its focus on web technology, is particularly well suited to being incorporated into the WebCharts framework.

Several of these systems provide functionality over the visualizations that are worth considering. IVTK provides a shared "magic lens" system, for example, which applies across any visualization. Weaver [23] provides a shared selection and interaction layer. WebCharts supports brushing and linking between disparate visualizations in a similar fashion.

In each of these systems (with the exception of Protovis), a developer adds relevant libraries to their application as part of the development process, and then compiles their visualization. In Protovis, visualizations are controlled by the website developer who has embedded the visualization. Our focus is extending application dynamically, at run-time, so that end-users can choose what visualizations they will need.

2.2 Visualization on the Web

WebCharts leverage deployment across the web, and uses internet protocols as a communication mechanism. Increasingly, the internet is being used as a delivery mechanism for visualizations.

There are several common architectures for constructing visualizations on the web. Visualizations can be generated as image files that can be embedded in a webpage (e.g. [8]). Alternatively, client-side JavaScript code, Flash objects, and Java objects have been used to actually compute the visualization on the user's computer. For example, Many Eyes [22] uses this mechanism: the Java applet that runs in the user's browser connects to the website, downloads the data, and renders the visualization. WebChart visualization can use either strategy (server or client rendering), but visualizations that support offline visualizations must support client rendering.

Several tools allow site designers to build visualizations easily. VisGets [7] and Exhibit [13] provide infrastructure that allow a web developer to assemble a coordinated series of visualizations that can be embedded in a web page. The web developer can choose from a variety of provided visualizations; the end-user can switch between views, select and filter data, and explore the resultant data.

Perhaps the closest analogue to WebCharts is the Google Visualizations [11]. Google Visualizations are componentized visualizations packaged in Javascript. All Google Visualizations share a small set of Javascript commands, including an ability to set data and a callback to monitor selection. A web developer can incorporate several Google Visualizations into a webpage by placing their data into a single table, and applying the table to the visualizations. In addition, Google Spreadsheet allows users to add arbitrary Google Visualizations to their spreadsheets. Google Visualization is a client-only solution: it does not have an ability to accommodate visualizations in desktop applications, nor can it work when a user does not have an internet connection. The WebCharts framework includes an application-side component, which accommodates aggregation, small multiples, and remapping data between columns.

3 System Description of WebCharts

WebCharts enables visualizations to be dynamically added to applications. In order to do so, both the visualization and the application must support the WebCharts interface.

Figure 2 shows a system schematic of WebCharts. The top half describes the design of the visualization; the bottom half describes the design of the client application, which we refer to as the 'Host.'

In general, our scheme is to have the application embed a web browser control. Many web browsers at this point come in embeddable form; applications with embedded web browsers can typically send Javascript commands to the browser. The browser control is used to contain the visualization, which is implemented in HTML, or any language that can be embedded in a web browser, such as Silverlight, Javascript, Java, or Flash. The host application sends data and commands to the chart; the chart sends back events.

We have provided two libraries to help developers add WebCharts support to the applications and visualizations. The **HostLib** library is used on the host application, and provides communication with the visualizations. The **ChartLib** handles communications for a visualization, and also includes an optional lightweight visualization

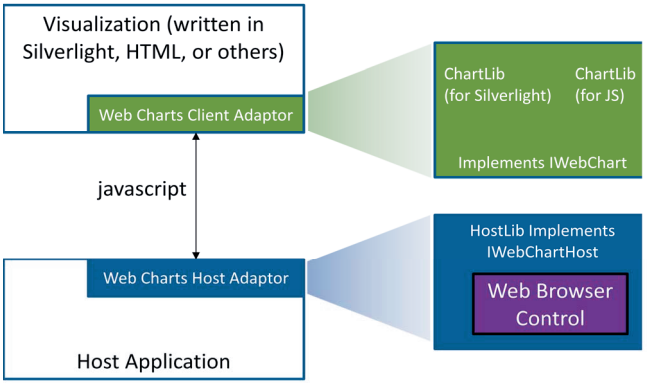

Figure 2: System Diagram of WebCharts. Host applications contain a web browser control in which a visualization is rendered.

framework. Our current implementations are written in C# (for hosts) and in Silverlight or Javascript (for visualizations); however, there is nothing in our design that would preclude interoperability with applications and visualizations written in any other language or framework.

We have implemented an extension to Microsoft Excel 2007 using the standard add-in mechanisms and we will use that to illustrate many examples. We will illustrate other host applications of the system in later sections.

In the following sections, we discuss the design of **visualizations** and the design of the **host.** Next, we discuss the **interaction mechanisms** that allow the visualization to send back events to the host. We discuss **aggregation** and **column selection**, which allows visualizations to show appropriate subsets of data. We discuss **styling and theming**, which allows visualizations to share a common visual appearance. Last, we discuss **persistence and security,** which allows host applications to store and save visualizations, and address data privacy issues.

3.1 Defining a visualization

In our framework, a visualization can be thought of as a web page which receives data from a host application. Visualizations maintain a very simple interface (Figure 3): they must accept data. In addition, they may be able to communicate a selection, take color and theme updates, and share custom parameters. Visualizations communicate entirely through a Javascript bridge; the Javascript calls these commands on the chart.

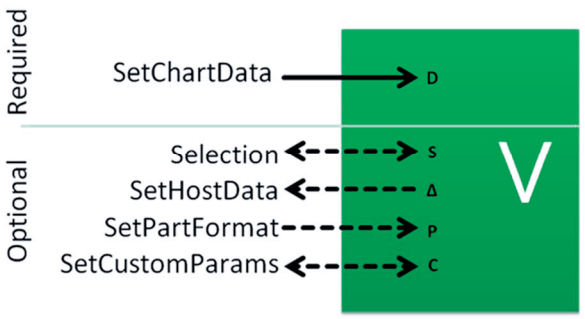

Figure 3: Host/Visualization communication

The communication mechanism is managed by **HostLib**, a library which runs on the host side, and **ChartLib**, which is available on the chart side. The host application communicates with the visualization through Hostlib; the visualization, in turn, receives all its messages through ChartLib commands. HostLib and ChartLib are responsible for serializing and deserializing data (respectively), and for packaging other commands.

From a hosting perspective, an application developer includes a reference to HostLib, and adds a ChartHost control to their application. Hostlib manages the web browser component, the network connection, and the Javascript communication. The Host can make the calls in Figure 3, and can register for callbacks.

Figure 4 : Excel spreadsheet showing some visualizations using the WebCharts Framework. Tag Clouds, Maps, TreeMaps, and a Focus + Context visualization have all been added to a spreadsheet. At the top, a palette offers more visualizations.

Similarly, from a chart's perspective, a visualization developer includes Chartlib, which is implemented in both Silverlight and Javascript. They can register for events from the ChartLib, which will handle incoming messages to the visualization. The Silverlight version of ChartLib is useful for building custom visualizations, while the Javascript version is useful for wrapping prewritten visualizations.

3.1.1 Basic ChartLib Commands

SetChartData is the basic call of WebCharts, and must be implemented by every visualization. This command takes a dataset (in XML), consisting of one or more tables as a parameter. A data table consists of a series of named and typed columns and a number of rows. Most data structures can be represented in tabular form readily (see section 3.2.1); the **HostLib** and **ChartLib** libraries provide support for translating a dataset to and from XML.

In addition to this command, every Chart and Host can choose to support additional commands which provide a richer experience.

SetChartSelection and **SetHostSelection** are sent from the host to the visualization and back (respectively) to communicate the rows and columns that are logically selected by the host, or other visualizations This forms the basis for brushing and linking support as well as the ability for visualizations to filter data based on the selection (see Section 3.3).

SetHostData allows the visualization to request that the host change the data based on actions in the visualization. (see section 3.1.4)

SetPartFormat allows the host to send styles and themes to the visualization. These are described in section 3.4

SetCustomParams allows the visualizations to accept custom parameters. Host applications that support customizable user interfaces can send parameters to the visualization. Conversely, when parameters are changed from the visualization (such as when a map is moved or an axis is altered), the new values are sent back through this command. These custom parameters are also persisted with the visualizations by the host (see section 3.5)

3.1.2 Declaring Custom Parameters

Each visualization may have its own custom parameters; and each visualization has its own capabilities. If a visualization requires, for example, a field of type 'ordinal', than mapping it to a string value may not be useful. Visualizations may declare their interfaces by providing an optional manifest file, called **WebCharts.xml.** This file specifies the capabilities of the visualization, and which columns of data it can accept. In addition, it can describe any custom parameters that it supports and can announce any other relevant metadata. By providing a separate capabilities file, users can search for certain capabilities amongst a large number of charts stored on a web site without needing to invoke and examine each of them.

3.1.3 Updating and Modifying Data

One advantage of embedding visualizations into a host application is that there can be a tight integration between the user editing and updating their data and the visualization. If the visualization is embedded in a spreadsheet, the user need not switch context from the visualization application to a spreadsheet to correct data errors, then switch back to update the visualization. Instead, the user can interactively modify or correct data, and see it instantly updated in the visualization or manipulate the data in the visualization and have it update the data in the host. One way of accomplishing this is for the host application to detect changes to the data and automatically send the data to the visualization.

Data updating can work in both directions: visualizations can choose to call SetHostData to update data on the host, perhaps as a result of the user manipulating shapes within the visualization. In this case, the host application must provide a protection mechanism to help prevent accidental data changes by the user.

3.2 Aggregating Data and Choosing Columns

Two different approaches to datasets seem to be common among visualization packages. First, the package can visualize input data without transformation, as is done with Microsoft's Excel. In Excel, two columns of data are understood as x and y axes. In other packages, such as Tableau (and Excel's PivotTables), visualizations implicitly contain aggregation: a user chooses a dimension and measure; the application adds, averages, or counts the measure in order to visualize the values. Since the goal of WebCharts is to conveniently allow the embedding of a variety of visualizations, we wanted to make aggregation capabilities available without requiring support from, the visualization. Of course, some visualization, such as histograms, prefer to do their own aggregations; this is an optional aspect of the system.

Aggregation is supported on the host, within HostLib. HostLib can transform a data table, given a few parameters. These parameters are similar to those that drive VizQL [21]: columns of the data table can be used as aggregation values, filters, and sort orders. The aggregated values then can be sent to the visualization; columns of the aggregation can be split off to enable small multiples. The host maintains mapping objects, which maintain a mapping from unaggregated rows to their aggregated results. These allow the host to subsequently map selection rows from the visualization to the host domain, or from the host to the visualization domain (between the pre/post query data rows).

The aggregation is performed by the host rather than the visualization for two reasons. First, this allows aggregation to be implemented for any visualization, including those that are unable to scale to large datasets. Second, the host level can also provide significant optimization, since the individual visualizations do not need to be sent or store the entire datatable, but only the data that is actually visualized. A huge dataset can be reduced to only few datapoints in some of the aggregations. Alternatively, there are already visualizations that aggregate or optimize the renderings of data based on the view (for instance, a scatterplot with dense areas can choose not to render every data instance). This is still supported by sending all the data to the visualization.

3.2.1 Non-Rectangular and Multi-Table Data

The SetChartData command sends data via a data table, which represents columnar data easily. Other common data structures can be readily normalized into tabular form. The Host is responsible for normalizing its native datastructures into a tables that the visualization can use.

For instance, hierarchical data can be represented as name, parent, metadata as seen in Table 1.

Table 1: Hierarchical data represented in tabular form. Used for TreeMap visualization.

Name	Parent	Metadata (hierarchy level)
Bill Gates	None	1
Steve Ballmer	Bill Gates	2
Steven Sinofsky	Steve Ballmer	3
Stephen Elop	Steve Ballmer	3

Network models can be represented with two tables, one each for nodes, the other for edges. Table 2 illustrates a sample graph format.

Table 2: Two tables used to represent graphical data. The first represents edges while the second represents node data.

Node1	Node2	Metadata (Distance, in miles)
Boston	New York	190
Boston	San Francisco	2708
Seattle	San Francisco	679
Seattle	Boston	2484

Node	Metadata (population, in people)
Boston	620535
Seattle	602000
New York	8214000
San Francisco	809000

3.3 Interactivity: Brushing and Linking WebCharts

There has been a great deal of research showing the benefits of linking together multiple visualizations via selection and highlighting behaviors. Becker [2] proposed early brushing behavior in scatterplots, and the systems proposed by both North and Weaver [18][23] both support coordinated views. North provides a taxonomy of types of actions that can be propagated across brushing and linking, including selecting subsets of data, providing detail on a particular item, and restricting and filtering data. WebCharts implicitly supports all of these by providing a universal selection mechanism.

In the WebCharts framework, a selection is created by the user in either the host application or in a visualization. If the user has made a selection within a visualization, that selection is sent back to the host as the set of data rows which correspond to the selection. **Hostlib** provides infrastructure for translating the selection back to the original dataset. (In the case of aggregation, for instance, this means computing which rows of original data correspond to the aggregated values). With the selection now in the domain of the original data, the host can now both reflect the selection in its own user interface, if appropriate, and then send the selection to other visualizations that refer to the data.

Each visualization can respond to the selection as is most appropriate. In many visualizations, for example, selection commands correspond to highlighting particular shape objects within the display. However, we have also implemented a 'details on demand' visualization that merely prints all values from selected columns, and resolves image URLs as pictures (Figure 4).

3.4 Styling and Theming

A host can choose to support styling and theming for its hosted visualizations. A style is a collection of typefaces, text sizes, and colors. Styles specify the color and typeface for standard parts of a chart, such as the background color, the color of foreground objects, and the typeface for captions. A theme is a collection of styles that, together, describe the way that a chart is rendered. Using styles and themes, a host can ensure that all charts use the same color palette and similar typefaces. Implementation of styles for a visualization is optional; a chart implementation can choose to act on or ignore any portion of these commands. A theme typically includes color selections for both selected and unselected items, for multiple series of data.

Styles and themes are also used to enable small multiples and other grouped visualizations. In a small multiple, it is desirable to hide axis captions and legends that are shared between the several charts. The style information includes a "hidden" label, which allows the host to request that the visualization hide its unneeded axis labels.

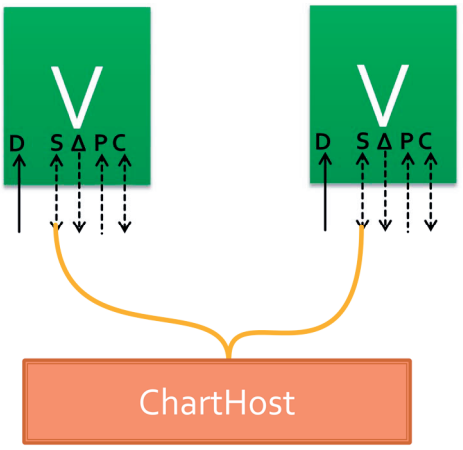

Figure 5: Brushing and linking within WebCharts. All communication channels work as usual, except that selection is routed through the ChartHost.

3.5 Persistence: Save-Load and Copy-Paste

It is useful for applications often provide persistence for operations such as saving and loading. When the embedding application is saved, the visualization should be saved with it; users should be able to copy and paste visualizations into other applications. Hostlib provides a mechanism for persisting (and reloading) a visualization. It includes the original URL, a reference to the source data, and any parameter settings that the user has set, and its current style settings. Optionally, the persisted version can include an offline executable version of the visualization code (the HTML page and related resources). Last, the persistent form of the visualization can include a bitmap snapshot of the visualization, to be used as a fallback when executing the visualization is not possible.

3.6 Offline Use, Privacy, and Security

Two great concerns that are associated with web based visualizations is the ability to use them when not connected to the internet and privacy/security issues. Visualizations that do not require the internet can be used offline, while some cannot. Offline use is possible for those visualizations that support disconnected operation from the internet. This means that once acquired via the net, the visualization will continue to function if all the code is available and the visualization does not require external data.

We have investigated three types of visualizations:

- Visualizations that, once downloaded, can continue to function without an active connection.
- Visualizations that can run locally, but require external data. This is the case for the world map pictured above. When the user pans or zooms, other tiles are downloaded from the tile-server. In addition, this visualization uses an external service for turning city names into longitude and latitude coordinates. While data such as city names is sent to the network, most of the data stays resident on the local system.

These visualizations continue to work as long as no new external information is needed and the data is not sent out on the internet. The third case of visualizations cannot be moved offline:

- Visualizations that run externally. These are cases where the visualization is created on a separate server, and either a bitmap, or laid out graphical objects are sent to the client. These visualizations can only be run when connected.

Related to the online issue is privacy. Systems such as ManyEyes and Tableau Public require all data to be shared their data with the world. This can be inconvenient for sensitive information.

Visualizations of the first type do not share their data at all with the external world. Visualizations of the second type share some information, but the actual visualization is still computed and

rendered locally. Visualizations of the third type require sharing their visualization with the external service. We are now investigating effective UI indicators to help users understand which sort of visualization they are using, and what behavior they can expect both offline and with regards to their privacy.

Any system that loads programs from the web represents potential security concerns, both with the data that is being shared with it and to other data resident on local client. Since the visualization is embedded in a browser, the same security conditions that is used for general internet browsing are in place – that is the code runs in a sandbox and does not have access to data except that which is explicitly passed to it.

3.7 Performance

The WebCharts scheme requires using a hosted web browser, and communicating with it via Javascript calls. This means that all data must be flattened into a string form for Javascript, and then expanded outward back to a datatable in the visualization. As a result, WebCharts imposes some overhead above and beyond the needs of the visualization.

Performance issues can be broken down into 4 stages.

- *Capture*: is the time to capture data from the host application (in the case of the tests shown here, Excel)
- *Shape*: is time spent selecting columns and organizing the data into tabular form in order to prepare it for the chart
- *Marshal*: is the time spent converting the data to a string, sending to the visualization, and converting back to data
- *Build*: is the time spent by the visualization creating and laying out shapes.

In order to evaluate the impact of these phases, we evaluated the application against a moderate workload of 4200 rows. We computed the timing for generating a chart of all 4200 rows; the timing for generating an aggregated chart of just 4 rows, and of generating a trivial chart of one row. The following data is from a desktop PC, core i7 with 6GB of memory working on a typical sales spreadsheet with 4200 rows and 20 columns of data in msecs.

We note that *capture* appears to be proportionate to the number of rows processed, that *shape* currently dominates the run, and that *marshal* and *build* are proportionate to the number of rows transmitted. This suggests that larger aggregations can continue to scale, and that visualization designers can gain by optimizing their own code.

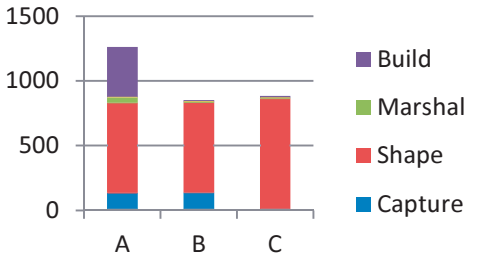

Figure 6: Timing graphs for each condition. Condition A: non-aggregated (4200 rows), Condition B: Aggregated (4 rows), Condition C: Single Row

Table 3: Performance numbers for communicating with a visualization in WebCharts.

Condition	Capture	Shape	Marshal	Build	Total
A: Non-aggregated (4200 rows)	136	696	40	390	1262
B: Aggregated (4 rows)	138	696	3	15	852
C: Single Row (1 row)	5	859	3	16	883

3.8 Versioning

Because WebCharts are deployed to and loaded from web URLs, any time a visualization is created or opened (or refreshed), by default, the latest version of the visualization will be downloaded and used. The benefits of this behavior include always running the latest (and presumably best) version of the visualization. The downsides include users having to deal with broken features and unexpected new behaviours, often at very inconvenient times.

We believe most users would like host applications to default to an automatic "version capture" mode, where the first time they download the visualization to their application, it captures that version (when possible) and keeps providing it to the user (via a web service, and local caching), until the user requests to move to another version. This can be supplemented by unobtrusive notifications of newer versions of visualizations becoming available.

4 Case Studies

In this section, we discuss several instances of how these elements have been brought together to add visualizations to different applications.

4.1 Wrapping existing visualizations

While we have implemented some new visualizations for use directly within the WebCharts framework (including TagClouds, and a special purpose Map Control), it is the ease with incorporating existing visualizations that sets WebCharts apart. We have written small wrappers that translate between the WebChart API and the Google Visualization API. In addition, we have written an adaptor for the Silverlight Toolkit, which enables TreeMaps, animated charts, and scatter-plots. Since Java applets, including those written in Processing can be embedded within a web page, we can potentially write adaptors for those as well, but we have not done so as yet.

4.1.1 Exhibit in Excel

Exhibit [13] is a multi-dimensional faceted browser implemented in pure Javascript. It creates curated visualizations based on web-based datasets. An author can easily select structured data, choose columns, and add visualizations. A user can switch between the visualizations, and can choose filters on the dataset. Exhibit requires datasets that are placed on the web.

We embedded Exhibit in Excel using WebCharts. Thus, Exhibit used Excel-based data, but maintained the curated set of visualizations. We wrote a very small wrapper which translated the SetData call's XML format into JSON. We then mapped by hand the column names with the exhibit attributes. The entire procedure took only about 15 minutes. Creating an explicit roles file within exhibit, where we can match columns from the data with roles in the exhibit would be a useful extension. We are also exploring the use of Dido [15] which allows modification of the data within the web page which could then reflect those changes back to the underlying data.

4.2 Alternate hosts 1: WebCharts in IronPython

IronPython is a scripting language that follows Python syntax, but is based on Microsoft's Common Language Runtime.CLR. We implemented an IronPython extension to WebCharts using IronPython as the host. This allows IronPython users to add visualizations inline within their code.

The interface for users is quite simple: they can invoke a visualization by creating a new Visualization object with a URL, and then can make calls against it

```
v=new Visualization( "http://server/vis.html" )
v.setData( dataTable )
```

The setData call (implemented within IronPython) uses Hostlib to translate the data in the table into the Javascript format that

WebCharts requires. This allows visualizations to be called from a read-eval-print loop enabling support for interactive exploration of data from a command line environment, similar to the Matlab and R.

4.3 Alternate hosts 2: Charts as Visual Debuggers

The Visual Studio IDE allows general extensions to be added anywhere within the user interface, including the debugger. By embedding the web-charts framework within a Visual Studio extension, we have enabled simple visualizations of data structures at debug time. See figure 7. While the demonstration at present is primarily a proof of concept, enabling visualization of memory allocation, processor utilization, values of variables over time could all aid in the debugging process enormously.

5 Conclusions

We have created a simple strategy and architecture by which visualizations can be plugged in to a variety of host applications. This empowers end-users, application designers, and visualization designers by allowing greater reuse of existing code. End-users do not have to wait for new revisions of existing applications to use the latest techniques; designers of domain specific visualizations can work on just the visualization and have them incorporated into a variety of different host applications. The strategy helps bridge the benefits of both the client-centric and web-centric worlds. Users can do local processing and visualizations on their own machine, yet obtain new visualizations from the web world where they can be updated more frequently and the 'long-tail' phenomena (small amounts of special purpose applications) are available.

Acknowledgments

The authors wish to thank David Karger, for his assistance with Exhibit.

References

[1] Baumgartner, Jason and Börner, Katy (2002). Towards an XML Toolkit for a Software Repository Supporting Information Visualization Education. IEEE Information Visualization Conference, Boston, MA, 2002. Interactive Poster.

[2] R.A. Becker and W.S. Cleveland, "Brushing Scatterplots," Technometrics, vol. 29, May. 1987, pp. 127-142. 1987.

[3] B.B. Bederson, J. Hollan, K. Perlin, J. Meyer, D. Bacon and G. Furnas Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics. Journal of Visual Languages and Computing, 7. 3-31. 1994.

[4] Benjamin B. Bederson, Jesse Grosjean, Jon Meyer, "Toolkit Design for Interactive Structured Graphics," IEEE Transactions on Software Engineering, pp. 535-546, August, 2004.

[5] Bederson, B. B., Meyer, J., and Good, L. 2000. Jazz: an extensible zoomable user interface graphics toolkit in Java. In Proceedings of the 13th Annual ACM Symposium on User interface Software and Technology (San Diego, California, United States, November 06 - 08, 2000). UIST '00. ACM, New York, NY, 171-180.

[6] Bostock, M.; Heer, J.; , "Protovis: A Graphical Toolkit for Visualization," Visualization and Computer Graphics, IEEE Transactions on , vol.15, no.6, pp.1121-1128, Nov.-Dec. 2009

[7] Dork, M.; Carpendale, S.; Collins, C.; Williamson, C.; , "VisGets: Coordinated Visualizations for Web-based Information Exploration and Discovery," Visualization and Computer Graphics, IEEE Transactions on , vol.14, no.6, pp.1205-1212, Nov.-Dec. 2008.]

[8] Dundas Chart for ASP.NET. Dundas Documentation: 2005-2009. http://support.dundas.com/Dashboard1.Documentation.ashx . Assessed 2010.

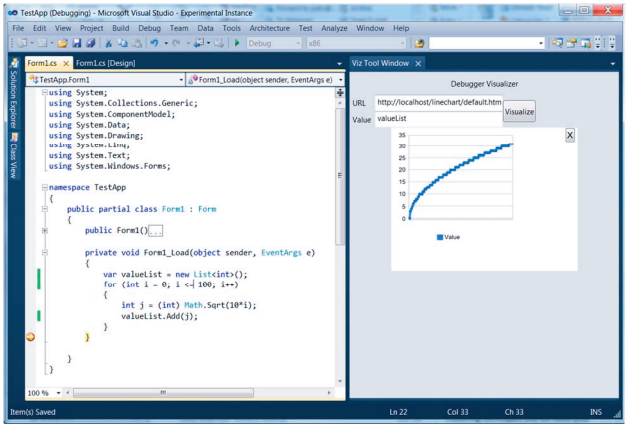

Figure 7: WebCharts embedded in a development environment (Visual Studio) and used for debugging.

[9] Fekete, J.-D., "The InfoVis Toolkit," Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on , vol., no., pp.167-174.

[10] Flare, http://flare.prefuse.org, March 2010.

[11] Google Visualization API: http://code.google.com/apis/visualization/interactive_charts.htm . Assessed 2010.

[12] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In CHI, 2005.

[13] D. F. Huynh, D. R. Karger, and R. C. Miller. Exhibit: Lightweight structured data publishing. In WWW '07: Proc. of the Int. World Wide Web Conf., pages 737–746. ACM Press, 2007.

[14] JSON, http://www.json.org. Online assessed 2010.

[15] Karger, D. R., Ostler, S., and Lee, R. 2009. The web page as a WYSIWYG end-user customizable database-backed information management application. In *Proceedings of the 22nd Annual ACM Symposium on User interface Software and Technology* (Victoria, BC, Canada, October 04 - 07, 2009). UIST '09. ACM, New York, NY, 257-260.

[16] Lienhard, A.; Kuhn, A.; Greevy, O.; , "Rapid Prototyping of Visualizations using Mondrian," Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on Software, vol., no., pp.67-70, 24-25 June 2007.

[17] McKeon, M.;"Harnessing the Information Ecosystem with Wiki-based Visualization Dashboards," Visualization and Computer Graphics, IEEE Transactions on , vol.15, no.6, pp.1081-1088, Nov.-Dec. 2009.

[18] North, C. and Shneiderman, B. 2000. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *Proceedings of the Working Conference on Advanced Visual interfaces* (Palermo, Italy). AVI '00. ACM, New York, NY, 128-135.

[19] Processing. http://processing.orghttp://processing.orghttp://processing.org, March 2010.

[20] TIBCO Spotfire. http://spotfire.tibco.com/. Assessed 2010.

[21] Chris Stolte, Diang Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multi-dimensional relational databases. Transactions on Visualization and Computer Graphics, 8(1):52–65, Jan 2002.

[22] F. Viégas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. Many eyes: A site for visualization at internet scale. IEEE Trans. On Visualization and Computer Graphics, 13(6):1121–1128, Nov/Dec 2007.

[23] Weaver, C.; "Building Highly-Coordinated Visualizations in Improvise," Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on , vol., no., pp.159-166. 2004.