

# An Extensible Framework for Information Visualization and Collection

William Luebke

Michael Richmond      Jacob Somervell  
Computer Science Department  
Virginia Polytechnic Institute and State University  
Blacksburg, VA 24061-0106, USA  
{wluebke, mrichmon, jsomerve, mccricks}@vt.edu

D. Scott McCrickard

## ABSTRACT

Developing successful information visualization experiments, principles, and applications requires iterative refinement of ideas and prototypes. Oftentimes realizing these prototypes involves a great deal of programming effort. Clearly, minimizing this effort permits research at a more accelerated pace due to shorter prototype turnaround time. The authors developed an extensible and flexible system along these lines that enables programmers and researchers to update and interchange data visualization and collection techniques with little effort. This system is discussed along with its relevant design patterns in the greater context of software orthogonality. Finally, the system is utilized to develop a computer supported cooperative work application for a large screen display.

## Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures – patterns, domain specific architectures.

## General Terms

Design, Experimentation, Human Factors.

## Keywords

Orthogonality, frameworks, large screen displays, computer supported collaborative work.

## 1. INTRODUCTION

As computer and Internet use increases over time, the amount of information our civilization collectively gathers grows exponentially [7]. Consequently, people are confronted with more information and expected to digest it in less time. For this reason, information needs to be filtered, summarized, and displayed, enabling a person to obtain information relevant to them in a way that can be understood intuitively or with little effort.

The broad range of information sources and communication mechanisms complicates the creation of software to address the

myriad of different configuration options for various interfaces, visualizations, and information repositories. All too often, a resulting program, in an effort to tie together a wide variety of disparate components, will be difficult to modify and reuse at a future time. Our approach is to generate an extensible framework that will alleviate many of these issues.

As humans are primarily visually oriented creatures [1], creating information visualization techniques is a very important area of research within the human-computer interaction community. Information visualization has extremely broad applicability due to the virtually limitless information mediums, information contexts, and combinations thereof. For example, different visualization techniques need to be employed when peripherally displaying stock quotes using a ticker on television than when displaying a company's profits on a presentation during a quarterly meeting.

However, many information visualization prototypes use similar data, and others use similar visualization techniques to display different data. For example, several different methods are employed to display image data in different scenarios, PhotoMesa [3] and the Data Mountain [12] being two examples. This illustrates the former case. The latter case can be seen in the myriad familiar statistical diagrams, as bar charts, scatter plots, etc. can display data that comes from vastly different locations. In both of these cases, there is an overlap in functionality between the two programs: programs rewrite either data collection mechanisms or information visualization implementations. The benefits of software component reuse are clear—much time and effort could have been saved if duplicate functionality were not implemented more than once.

North's Snap-Together Visualization [10] addresses many of these issues, allowing users to coordinate multiple data visualization techniques to increase their understanding of an information space. However, it focuses on databases as data sources and visualizations confined to a standard computer monitor. To address the needs of the mobile user with changing needs, a system must be highly adaptable. Successful information visualization techniques could also benefit from experimentation in other information contexts. For example, how could Shneiderman's Starfield display [2] be adapted (if necessary) to work well on the screen of a cellular telephone?

Issues such as these drove our desire for a common, unifying, extensible framework to serve as a laboratory for information visualization and collection. The framework's architecture should provide an effortless way to extend its information collection, filtering, and visualization capabilities. These capabilities should be easy to author, with very little programming overhead dedicated to interfacing with the framework itself. The framework

should also support dynamic reconfiguration, allowing experimenters to swap information collection, filtering, or visualization capabilities without changing any code or even stopping the program. As they would all follow a minimal common interface, they could also be easily shared with other researchers and tried out with their own implementations in novel ways not necessarily thought of by the original author. A side effect of this type of architecture (and modularization in general) is that these capabilities are orthogonal to the framework itself, confining bugs and other issues to the modules where they belong, and generally simplifying code all around. Cohesion increases as the independent parts are decoupled.

Drawing upon design patterns and other tried and true software engineering strategies, as well as some new ideas, we have made progress in achieving the aforementioned goals with our implementation of a framework, and have had much preliminary success with it. Our initial project, described in this paper, involved studying effective visualizations for shared large screen displays in a laboratory environment. First, we present details of the framework...

## 2. ORTHOGONALITY AND EXTENSIBLE ARCHITECTURES

### 2.1 Orthogonality

Orthogonality describes a decoupling situation where unrelated software functionality is confined to separate modules. The term is borrowed from mathematics, where in a Cartesian space two vectors are orthogonal if the angle between them is a right angle. A change in one software module will not affect other orthogonal software modules, since the modules are interdependent [11].

A system with a high level of orthogonality has many benefits. First, bugs and other defects are isolated to a module instead of existing throughout the entire system, making them easier to track down and fix. Second, a change in one module will not affect the rest of the system, making adding enhancements more straightforward. Finally, the flexibility of the system overall is increased, as one module can be swapped out for another one if it implements the same interface, and the other modular components will not need to change. This can even be accomplished at run-time to enable extensible frameworks that lend themselves to on-the-fly configuration and plugability.

Such flexibility is achieved through the proper implementation of interfaces that allow the abstraction of dynamic software components. (This is called the “Interface” design pattern [5].) Interfaces control not only what functionality a given implementation will have, but also dictate how interaction with the implementation will occur via the function names contained in the interface. This allows software components to communicate in precisely the same fashion regardless of what implementation a software component is using, and enables a system to remove one implementation of a component and replace it with another that implements the same interface—even at run time (using something similar to the “Dynamic Linkage” design pattern [11]).

```
//This is an example of the interface that defines what the
//required behavior of some object is.
public interface SomeInterface
{
    //...
}

//This class is an implementation of SomeInterface that
//performs the required behavior.
public class SomeImplementation implements SomeInterface
{
    //...
}

//This is another implementation of SomeInterface that
//performs the required behavior, just in a different way.
public class AnotherImplementation implements SomeInterface
{
    //...
}

//This class houses an instance of SomeInterface, but by
//referring to it as a SomeInterface instead of a
//SomeImplementation, different implementations are able to
//be swapped out so long as they implement the functionality
//required of SomeInterface.
public class ExtensibleObject
{
    //Here is the reference to SomeInterface:
    protected SomeInterface _interface;

    public void createSomeInterface(String className)
        throws Exception
    {
        //Retrieve the class information about the class
        //named by className, and call the other method
        //signature:
        createSomeInterface(Class.forName(className));
    }

    public void createSomeInterface(Class cls)
        throws Exception
    {
        //Make sure that the class isn't just an interface
        //(since we cannot instantiate interfaces)
        if (cls.isInterface())
        {
            throw new IllegalArgumentException("The "+
                "class cannot be an interface.");
        }

        //Make sure that the class isn't an abstract
        //class. This part is a bit tricky, but the
        //11th bit of the modifiers is set if the class is
        //abstract. (See the JVM Specification, Table 4.1
        //for details.)
        if (cls.getModifiers() & 0x0400 > 0)
        {
            throw new IllegalArgumentException("The "+
                "class cannot be abstract.");
        }

        //Make sure that the class has a default
        //constructor so that we can create it without
        //passing any arguments:
        try
        {
            cls.getConstructor(new Class[]{});
        }
        catch (NoSuchMethodException e)
        {
            throw new IllegalArgumentException("The "+
                "class must have a default "+
                "constructor.");
        }

        //Finally, create the object:
        _interface = cls.newInstance();
    }
}
```

Figure 1. Example orthogonal/extension model implementation in Java.

Figure 1 provides an example Java<sup>1</sup> implementation of a class that allows extensible, orthogonal plugability. In the example, `ExtensibleObject` is a class that requires a certain orthogonal functionality defined by `SomeInterface` in order to perform its function in the program. Two implementations of `SomeInterface` exist—`SomeImplementation` and `AnotherImplementation`. The `createSomeInterface` method is called to initialize (or reinitialize) the `_interface` field of `ExtensibleObject` where the class name of the desired implementation is passed in.

For illustrative purposes, the code in Figure 1 can be represented diagrammatically as shown in Figure 2. Objects are labeled boxes, and interfaces are a specific styles of “plugs” attached to the left side of objects that implement them. Extensible objects that use orthogonal interfaces are designated with a plug insert of the appropriate type on their right side.

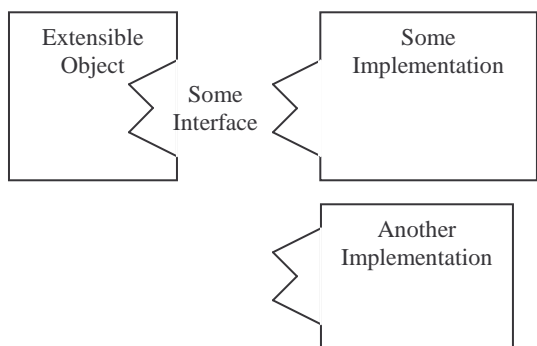


Figure 2. Diagram of Figure 1 classes and interfaces.

## 2.2 Adapter Extension

One can see that the orthogonality/extension model can be extended in other ways, creating a highly customizable system. For example, a structure similar to the adapter design pattern can be implemented, as depicted in Figure 3. We term this an “Adapter Extension.”

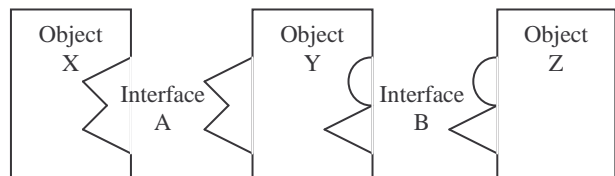


Figure 3. Example Adapter Extension.

<sup>1</sup> Java supports this activity easier than a non-dynamically typed programming language. For example, in C++, a programmer would be unable to easily perform these tests or create instances of classes just with a string containing the class name, as this sort of functionality is not available in the language natively. Java provides a reflection API, which is a vital part of the orthogonality/extension model, as it allows runtime querying of the methods and inheritance model of a class, and provides methods for dynamically loading and instantiating classes.

## 2.3 Multiple-Delegation Extension

As shown in Figure 4, there is no reason why an extensible object cannot delegate to multiple other implementations, should that behavior make sense for the interface in question. This is called a “Multiple-Delegation Extension.”

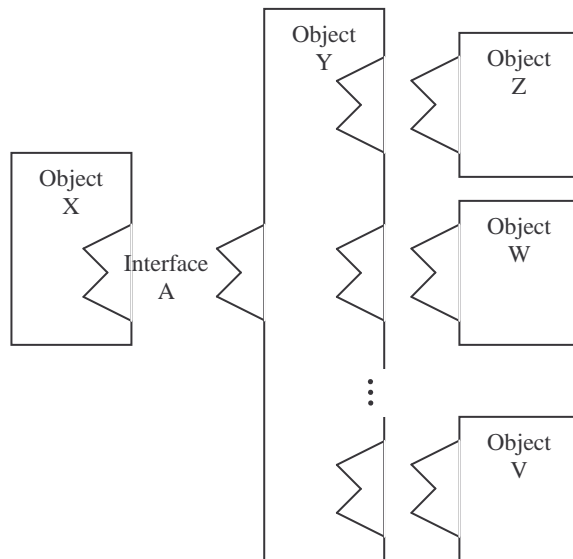


Figure 4. Example Multiple-Delegation Extension.

## 2.4 Non-trivial Extension

The orthogonality/extension model can be used to develop complex, robust systems in other non-trivial ways. Figure 5 provides a simple example.

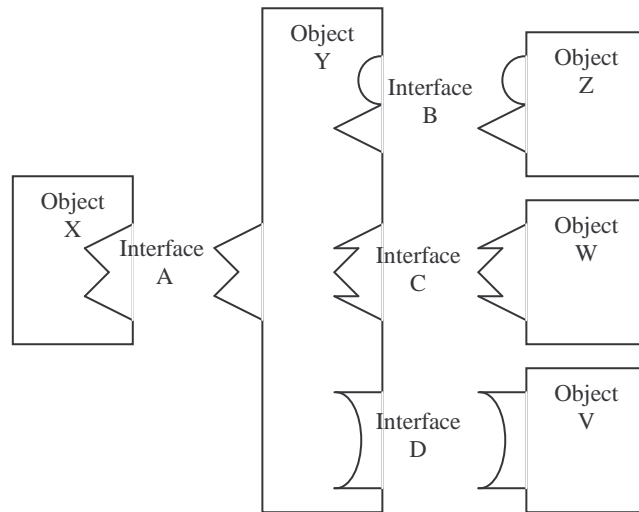
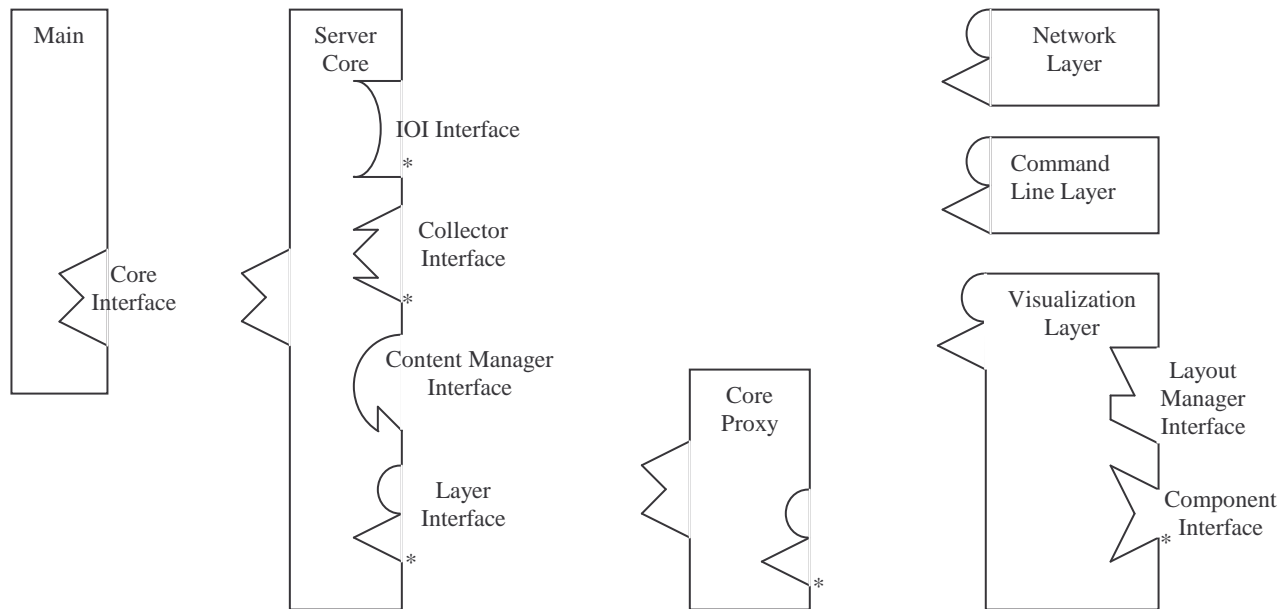


Figure 5. Example Non-trivial Extension.

## 3. ARCHITECTURE

The architecture of the system was designed to be extensible, flexible, and scalable, by providing a set of interfaces (and in some cases, default implementations) for information representation, collection, and visualization. Figure 6 provides a graphical representation of the system architecture.

The `Main` object sits at the root of the extensibility framework; its role is to instantiate and hold an implementation of the `Core`



**Figure 6. Extensible information visualization and collection architecture. (Asterisks denote zero or more of a type of interface.)**

interface. The core implementation is intended to be the “heart” of the system—it holds other four orthogonal components of the system. First, information is abstractly represented and stored in the system through the “item of interest (IOI)” interface. The IOI interface is generic enough to represent nearly any form of information, as there are no restrictions on the details of the implementing class. For example, it is possible to author textual IOIs, IOIs containing image data, sound bytes, or video streams. We have also implemented a “news IOI” which uses multiple-delegation extension to combine a textual IOI and an image IOI to represent a news article gathered from World Wide Web sources.

Second, the `Collector` interface represents a mechanism that collects information and creates IOIs from it. A collector can conceivably be implemented<sup>2</sup> to collect information from just about any source; it could read from a database, watch the stock market, ask users for input, or parse data from the Internet.

Third, the `ContentManager` interface provides a mechanism to determine what information is of most importance to the system at the current time. This information can be used to start or stop various collectors or to filter information.

Finally, the `Layer` interface provides access to the core, so external programs or users can interact with it. There are three main implementations of the layer interface. First, the command line layer allows control over the system to an administrator through the command line. Second, a network layer enables other programs to communicate with the core through the network. The

<sup>2</sup> Although the collector implementation must be written in Java, Java can interface with any other language, so the collection mechanism is not limited by this. In fact, Perl was used in one of our implementations to monitor news sites on the Internet such as Yahoo! News.

framework supports a client/server paradigm by allowing two copies of the framework to run—one using a server core, and the other using a core proxy that interfaces with the server core through its network layer. In this manner, all data storage and collection, potentially processor intensive tasks, are performed by the server, while clients launch visualization layers. This also minimizes the work necessary to add additional clients. Lastly, the visualization layer displays IOIs stored in the core to users.

The visualization layer is another example of a non-trivial extension as it provides extension interfaces of its own for a layout manager and components. The `Component` interface utilizes adapter extension to display an IOI in a meaningful way, separating potentially non-orthogonal information and visualization. The `LayoutManager` implementation then positions components to produce a visualization. For example, we have implemented a layout manager that displays the components associated with news IOIs arranged into a grid to form a grid visualization. Other better-known visualizations could easily fit in this model. For example, a hyperbolic browser [8] could assign a hierarchy to components and display the resulting tree using the hyperbolic layout algorithm.

It should be clear that implementations of the `Layer` interface need not be purely visual in nature. For example, implementations could be written to play sounds or otherwise affect the environment using real world displays [9].

The framework adheres to the aforementioned orthogonality model, which results in a robust flexible framework for information visualization and collection. This framework seems of particular interest to research where the ease of addition, combination, and or removal of systems components or implementations makes testing a large number of possible implementations or combination of components relatively effortless. For example, the orthogonality between the collectors and the display makes it possible to change what type of display is

being used or how the display is implemented without having to change how or what information is collected and vice versa; making it easy to test a large number of visualizations without the headache of reconfiguring a large system.

#### 4. RESULTS

The extensible/orthogonal framework has enabled our visualization system to evolve rapidly. We have successfully used it to develop a system called the “Photo News Board.” Its architecture is presented in Figure 7.

The Photo News Board is an application built upon the previously described framework that collects news headlines and photos, and displays them on a large screen display. Intended to run in a shared environment, such as a common meeting room, the Photo News Board displays news stories and photos according to the

common interests of the people in the presence of the display. Newer stories are displayed in the center of the display and older stories towards the edge. As new photos appear in the display, existing photos shift outwards, away from the center. A simple highlighting technique is used to show the details of the story associated with the photo (See Figure 8).

The Photo News Board derives its cores from the default core provided by the architecture to add in user profile functionality, and provides implementations for the necessary interfaces. For example, the `DefaultContentManager` implementation takes into account users’ interests (found in their profiles) to determine which collectors to launch. Text, image, and news IOIs were developed, as well as collectors that gather this information. Finally, the visualization layer’s functionality is completed by our `GridLayout` and `GridComponent` implementations.

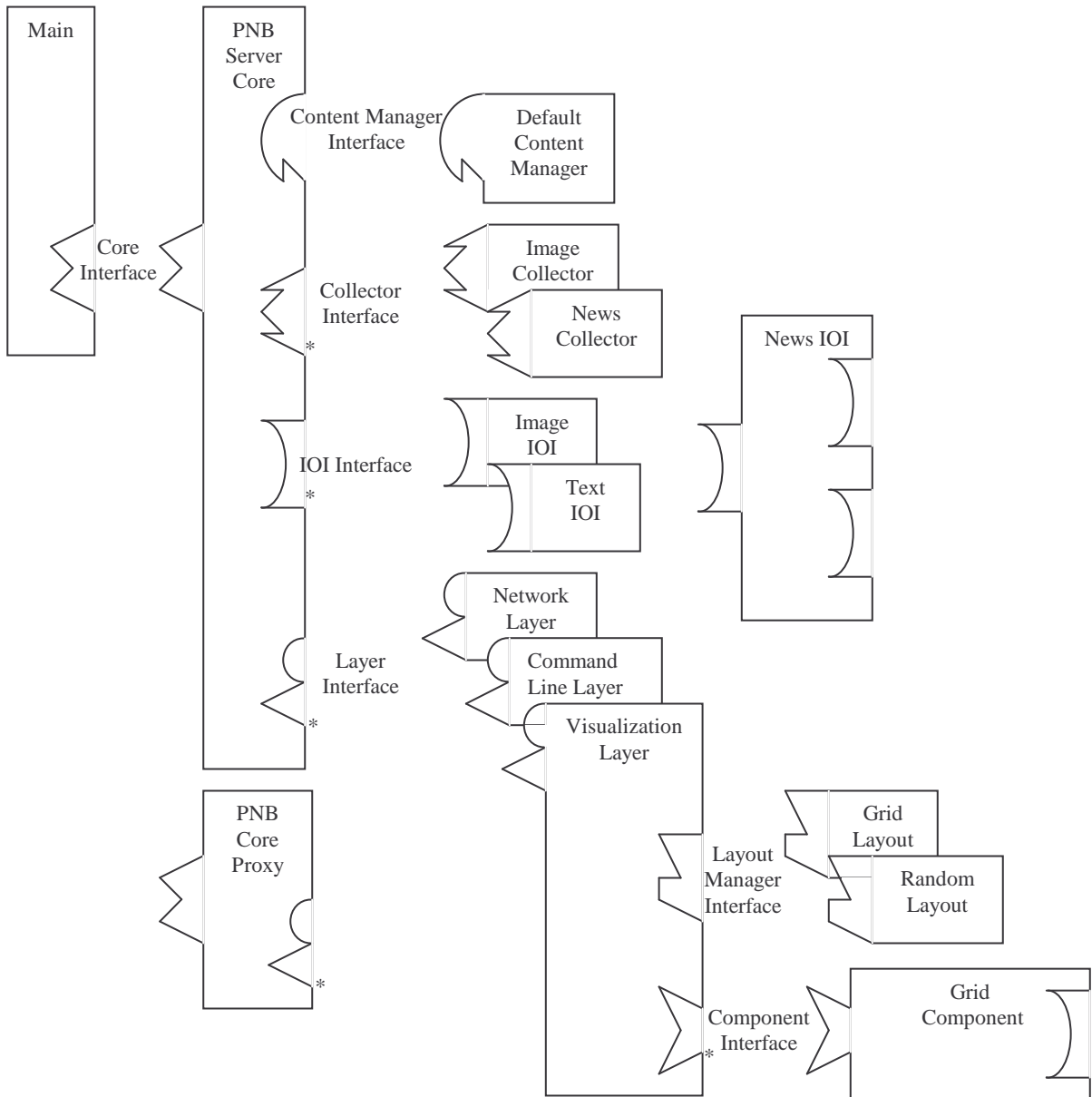
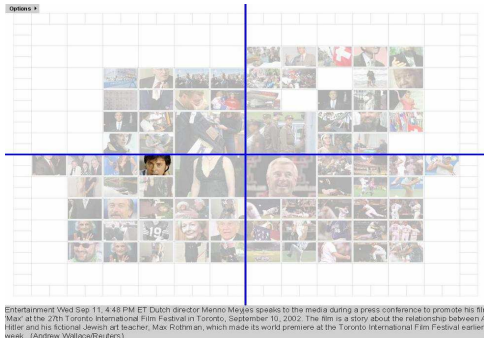


Figure 7. Photo News Board architecture. (Asterisks denote zero or more of a type of interface.)



**Figure 8. A screenshot of the Photo News Board.**

Throughout the Photo News Board development, many parts of the system were iteratively refined or replaced. We easily added and removed new collectors, IOIs, visualizations, layout managers, and components, especially early on in development in order to try different information visualization techniques. For example, we were initially using a standard flat, two-dimensional visualization implementation, but later replaced it with an implementation written with the Jazz Zoomable Interface Library [4]. These changes had no impact on the rest of the system, compressing development time considerably.

A simple usability evaluation in a controlled lab environment was conducted to show the system to users and get feedback about some of the design elements used in the display. From this pilot study, we learned that the display was effective at providing shared information on a public display. Users liked the access to the news stories and thought the highlighting reflecting the interests of the room occupants was a useful thing for promoting interaction.

Perhaps most telling as to the success of our framework is the fact that a developer not involved in the creation of the framework wrote several layouts that were easily plugged into the Photo News Board system. The layouts used animation and visualization techniques in conveying textual and graphical information, and worked on both desktop and large screen displays.

In the future, we hope to utilize the extensibility of the system in order to promote further experimentation and understanding of various layout and updating mechanisms. The true test of the utility of our framework will take place when other external developers create layers, components, and layouts for the Photo News Board, and when other systems are developed that make use of our framework. Our initial experiences suggest that it will successfully promote rapid extensibility and software reuse.

## 5. CONCLUSION

Both creating information visualization and collection applications and conducting usability experiments requires

iterative refinement of ideas and prototypes. A system architecture designed with orthogonality, flexibility, extensibility, and plugability in mind shortens the amount of time between these iterations and stands to make research and development in the HCI community faster.

## 6. REFERENCES

- [1] Abrams, A. Multimedia magic: Exploring the power of multimedia production. Boston: Allyn and Bacon, 1996.
- [2] Ahlberg, C. and Shneiderman, B. Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays. In *Proceedings of ACM CHI'94* (April 1994), pp. 313-317.
- [3] Bederson, B. B. PhotoMesa: A Zoomable Image Browser Using Quantum Treemaps and Bubblemaps. In *Proceedings of ACM UIST 2001*, pp. 71-80.
- [4] Bederson, B. B., Meyer, J., and Good, L. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. In *Proceedings of ACM UIST 2000*, pp.171-180.
- [5] Gamma, E., Helm, R., Johnson R., and Vlissides, J. Design Patterns: Elements of Reusable Object Oriented Software. Addison Wesley, 1995.
- [6] Grand, M. Patterns in Java, Volume 1. New York: Wiley Computer Publishing.
- [7] Internet statistics, Merit Network, Inc. (1995). <ftp://nic.merit.edu/nsfnet/statistics>.
- [8] Lamping, John, Rao, Ramana, Pirolli, Peter. A Focus + Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In *Proceedings of ACM CHI '95*, May 1995.
- [9] McCrickard, D. Scott, Wrighton, David, and Bussert, Dillon. Supporting the Construction of Real World Interfaces. Tech note in *Proceedings of IEEE HCC 2002*, Arlington VA, September 2002.
- [10] North, C., Shneiderman, B. Snap-Together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata. In *Proceedings of ACM Advanced Visual Interfaces 2000*, pp. 128-135.
- [11] Smaragdakis, Y., Batory, D. Implementing Reusable Object Oriented Components. In *Proceedings of ICSR '98*.
- [12] Robertson, George, Czerwinski, Mary, Larson, Kevin, Robbins, Daniel C., Thiel, David, van Dantzich, Maarten. Data Mountain: Using Spatial Memory for Document Management. In *Proceedings of ACM UIST '98*. (San Francisco, CA, November 1998), ACM Press, 153-162.