# A High-Level Architecture For The User Interface Prototyping Infrastructure: Lessons [Not] Learned From DENIM

*Jennifer Petrie*
*Department of Computer Science*
*University of Saskatchewan*
*jennifer.petrie@usask.ca*

**Abstract**

This paper investigates architectural issues for prototyping infrastructures. DENIM's architecture is studied because it has many features in common with the new prototyping infrastructure. This paper finds that DENIM's concrete architecture is not well designed and, as a result, cannot easily be extended or reused. The paper then presents a proposed architecture for the new prototyping infrastructure. The proposed architecture, although preliminary, shows promise for being reusable and extendable.

## 1 INTRODUCTION

This paper is a continuation of work presented in [8], which outlined an infrastructure for supporting prototyping. Specifically, this paper explores design issues for the prototyping infrastructure at the architectural level. Achieving a reusable and extendable architecture is key.

DENIM is a prototyping tool for supporting the early stages of web design through informal sketching [5]. It is one of the few prototyping tools that support low-fidelity stages of design. The infrastructure for prototyping presented in [8] can be considered an extension of DENIM as it aims to support the low-fidelity features of DENIM while supporting higher fidelity features and tying into software design models.

The architecture of DENIM is explored in order to assess the viability of extending DENIM rather than starting from scratch with developing this new prototyping infrastructure. Also, DENIM is studied in order to get a better understanding of what an architecture should and should not be when dealing with the domain of prototyping. This paper finds that DENIM has a poorly designed architecture that is neither easily extended nor reused. As such, a new well-designed architecture for the prototyping infrastructure is proposed. The goals of this architecture are for it to be easy to extend and ultimately reuse.

### 1.1 On Reuse

The benefits of reuse are clear. Reuse has potential to save substantial time, effort, and money in a software project. For these reasons, reuse is an important part of computer science research.

Reuse can take place at different levels in the software process, as pointed out in [1]. Reuse can occur at the language level. For example, iterations, methods, parameters, and classes all allow for reuse of a group of instructions in varying contexts and conditions. Reuse also can happen at the code level. Code libraries such as math utilities and user interface widgets are a form of reuse at the code level. Furthermore, reuse is being considered at the design or architecture level. Component-based architectures or frameworks are attempts at reuse at the design level. JavaBeans is an example of one component-based architecture. Parnas' work on modularization and information hiding was revolutionary in getting software engineers to think about reuse at the design level [7].

## 2 DENIM: AN EXISTING PROTOTYPING INFRASTRUCTURE

DENIM, or Design Environment for Navigation and Information Models, is an extension of SILK [4], which was one of the first sketch-based user interface prototyping tools. DENIM is specifically intended for web site design; however, DENIM's features are generally applicable to the design of any graphical user interface. DENIM provides these key features: informal pen-based sketching, design at different levels of granularity (from sitemap to storyboards to individual pages), zooming between these levels, creation of reusable components, a run mode to interact with a prototype web site, as well as gestures and pie menus for invoking commands. As these features suggest, DENIM may be useful for low-fidelity and

medium-fidelity prototyping. On the other hand, DENIM provides no support for transitioning to high-fidelity prototypes or implementation of the final product. A snapshot of DENIM is shown in Figure 1.

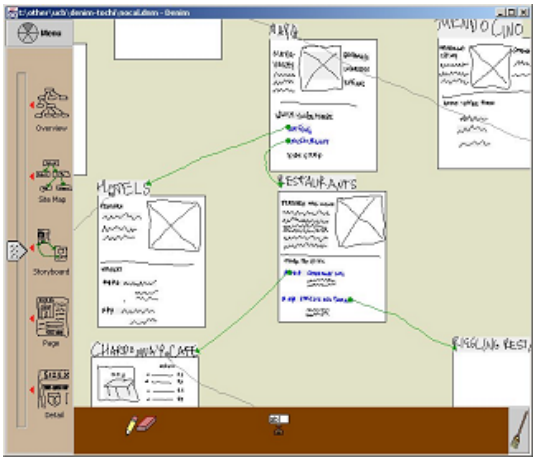


Fig. 1: A Snapshot of DENIM [5]

DENIM is built on top of SATIN [3], a toolkit for building applications that use informal pen-based interaction techniques such as sketching and gestures. Several useful pen-based interaction techniques have been developed over the years. However, these techniques are yet to be commonly used in desktop applications because they are difficult to implement. SATIN research tries to address this problem by focusing on developing a generalized architecture that supports reuse and extension of sketching and gesturing. SATIN provides these features: mechanisms for manipulating, handling, and interpreting strokes, zooming and rotating objects, switching between multiple views of an object, and specialized pen-based widgets such as pie menus. This list of features shows that most of DENIM's features originate at the SATIN level.

DENIM and SATIN are implemented in Java v1.3. Basic metrics are shown in Table 1. As the actual source line of code metrics indicate, DENIM is a rather small application. However, it has a fairly large number of classes and packages, considering many of its key features are at least partially implemented within SATIN.

Tab. 1: Basic Metrics of DENIM and SATIN

| | DENIM | SATIN | TOTAL |
|---|---|---|---|
| Source Files | 119 | 184 | 303 |
| Source LOC | 11,000 | 20,000 | 31,000 |
| Total LOC | 30,000 | 58,000 | 88,000 |
| Classes | 222 | 240 | 462 |
| Packages | 13 | 21 | 34 |

## 2.1 The Architecture

A conceptual architecture depicts the high-level structure of a system from the designers' perspective while abstracting away implementation details. A conceptual architecture is often very revealing about the user interface features and key functionality of an application. As such, the initial plan was to uncover the conceptual architecture of DENIM and SATIN. However, due to the lack of design documentation and access to the original designers, a concrete architecture was extracted for study instead of a conceptual architecture.

A concrete architecture is a more detailed view of the system, taking into account actual implementation details. The concrete architectures of DENIM and SATIN were extracted using a commercial tool called Headway reView [2] as well as Source3D [6]. Specifically, package-level diagrams as well as class-level diagrams were explored in order to arrive at the complete concrete architecture diagram, presented in Figure 2.

Figure 2 clearly indicates that DENIM has a poorly designed concrete architecture. As mentioned, DENIM is not a very large or complex application in terms of source lines of code so this architecture is unnecessary complicated. The architecture is decomposed into many fine-grained packages, making it very difficult to understand the overall system. For example, if one wanted to view all classes related to an individual web page sketch, it is unclear from the packaging where to find such details. Also, DENIM's architecture breaks the basic object-oriented design principle of low coupling; DENIM's design is very highly coupled, with most packages using several other packages. Moreover, interaction cycles exist between various packages. This tight coupling makes DENIM's architecture difficult to modify because a change to one class is likely to have side effects on several classes in several other packages. Consequently, all of these problems with DENIM's architecture result in a system that cannot easily be maintained, extended, or ultimately reused. Note that all of these problems with DENIM's architecture also exist in SATIN's architecture, not surprisingly as the same group of researchers were involved in

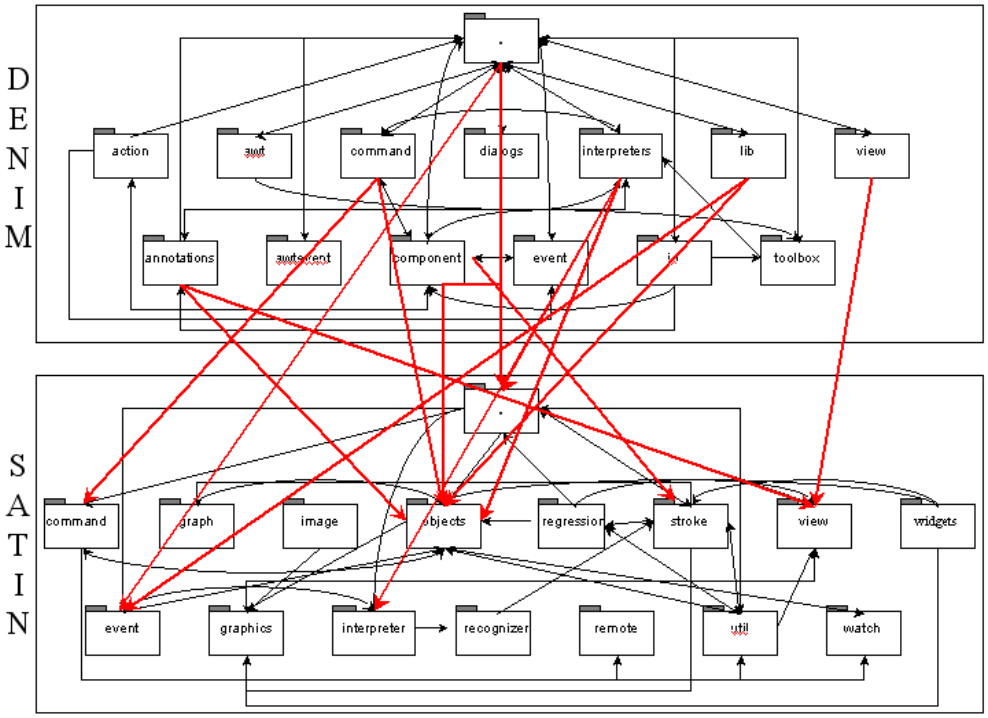


Fig. 2: Concrete Architecture Diagram of DENIM and SATIN

both.

A further problem is evidenced in how the two layers interact. DENIM makes over 3000 calls to SATIN. These calls are not simply made from a couple of classes in DENIM to a couple of classes in SATIN. Rather, these calls area spread over numerous classes and packages, making the coupling between the layers very high. A well-designed architecture for SATIN should only require minimal calls to it from the application layer. The way SATIN is designed, developers must be intimately familiar with the implementation details in order to reuse any pieces of code. This shows that SATIN has failed at its goals of being reusable and extendible.

## 3 Architectural Styles

Architectural styles describe a set of components and interactions among these components, called connectors, as well as constraints on how these components

and connectors can be composed. Thus an architectural style describes a class of systems according to its structural pattern. Shaw and Garlan [9] set out several commonly used architectural styles including Pipe-and-Filter, Data Abstraction, Hierarchical Layers, and Event-Based, Implicit Invocation. These styles, along with Chiron-2, will be discussed in this section because they all promote reusability and extendibility as well as have some applicability to the domain at hand.

## 3.1 Pipe-and-Filter Style

The Pipe-and-Filter Style consists of computational components, called filters, which perform local transformations on a set of inputs and then output a set of results. The connectors, known as pipes, hook the components together by transmitting the outputs from one filter to the inputs of another filter. In this style, the filters are independent and unaware of what filters precede or follow it. Consequently, this style is also very extendable and reusable since existing filters can easily be modified, used in other systems, or new ones added to the current system without affecting other filters. With this style, it is easy to understand an overall system by looking at the system as a simple composition of all the parts, which further promotes extension and reuse.

## 3.2 Data Abstraction

Data Abstraction is a style based on the principles of object-oriented design. In this style, objects are the components and method invocations are what connect these objects. The key characteristics of this style are: an object's representation is hidden from other objects and each object must ensure integrity of its underlying representation. As a result of these principles, systems that follow this style should be easy to reuse or extend because adding new objects to existing systems or using existing objects in a new system will not affect the representation of other objects. Objects should ultimately be interchangeable.

## 3.3 Hierarchical Layers

In the Hierarchical Layers Style, the layers are the components and typically procedure calls between the layers are the connectors. Each layer provides services to the layer above it. Thus interaction is limited to adjacent layers only. For this reason, systems following this style are easily extended because modifying or adding

a new layer at most affects two other layers. Furthermore, layers are easily reused in other systems assuming that they have appropriate interfaces.

## 3.4 Event-Based, Implicit Invocation

The Event-Based Style consists of independent processes or modules that are connected via messages. In this style, messages are sent implicitly rather than through explicit method invocations. In other words, the key aspect of this style is implicit invocation, where messages are announced and all the components that have a registered interest in that announcement receive notification. Thus a component that announces an event is independent from the components that receive the notification. This promotes reuse and extension because a new component can easily be added into a system simply by registering for events of the system.

## 3.5 C2 Style

Chiron-2, commonly referred to as C2, is an architectural style intended to support larger grained reuse and flexible composition of software [10]. Furthermore, this style also aims to allow for components to be programmed in different languages, ran in distributed, concurrent environments, interchanged at runtime, as well as to provide for multiple toolkits to be used. This style will be discussed in more detail than the other styles because it is somewhat more complicated and likely less familiar to the reader than the above commonly used styles.

Figure 3 provides an illustration of the C2 Style. The style has a network of independent modules as the components, which are connected via message routing devices. This style generally resembles the Event-Based, Implicit Invocation Style, but with further constraints on how the structure can be composed. Specifically, components are only allowed to communicate through connectors. Thus every component must be connected to a connector. A connector can have any number of components or other connectors connected to it. The key principle of C2 is that a component is only aware of components above it. Components indirectly communicate via implicit method invocations; that is, each component sends a request upward in the structure to be notified when a specific event occurs, and then the notifications are sent downward through the structure to all interested components.

Because this style has independent components that can only communicate via implicit invocation, this style promotes extension and reuse. Any component can

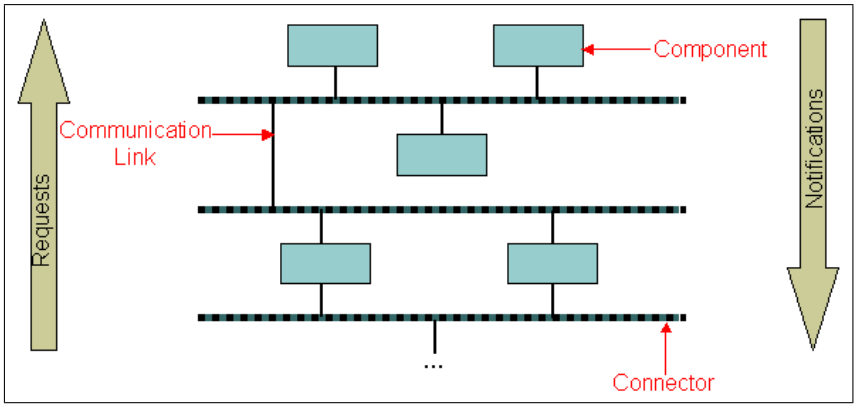


Fig. 3: An Illustration of the C2 Style [10]

easily be added to the current system or to a new system by simply sending appropriate requests. The connectors handle the rest of the work.

### 3.6 Heterogeneous Architectures

The architectural styles discussed in this section are generally considered “pure” styles. In actual systems, it is rare that a pure style is used. Generally, most architectures are considered heterogeneous, as noted by Garlan and Shaw, meaning that they are a composition of multiple “pure” architectural styles. For example, an architecture may be hierarchically layered but each layer may use a different style. The proposed architecture for the prototyping infrastructure will be a heterogeneous one that draws upon the various “pure” styles presented in this section.

## 4 A NEW INFRASTRUCTURE FOR SUPPORTING PROTOTYPING

Previous work by the author presented an infrastructure for supporting the transitioning of low- through high-fidelity prototyping, while bridging the gap with the software design process. No other prototyping tool supports all fidelities of prototypes, yet all fidelities are vital to achieving the best end product. Being able to transition quickly and easily between the different stages is key to encouraging the use of all stages in practice. Furthermore, software design is often performed independently from user interface design, which is not necessarily in the best in-

terest of the end product. This prototyping infrastructure tries to address this gap by bringing software designers and user interface designers together.

The key features of the prototyping infrastructure are supporting composition of low- through high-fidelity prototypes, tying the prototypes to application data and functionality, and linking interface design to software design. These features are depicted in Figures 4 and 5.

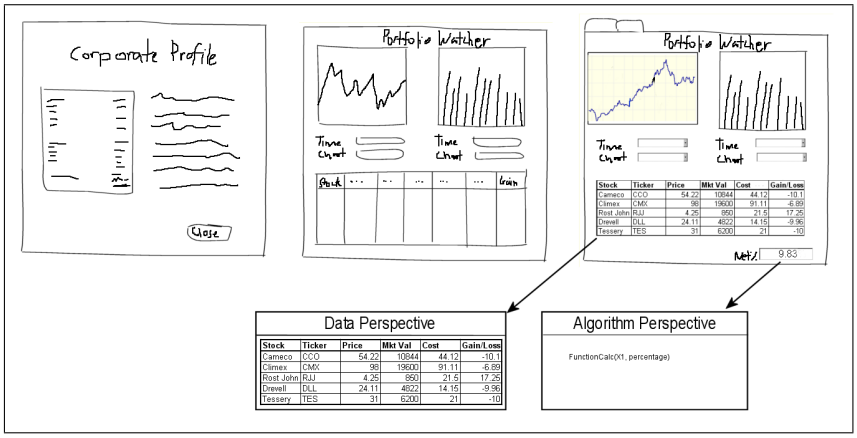


Fig. 4: Infrastructure promotes starting with low-fidelity sketches and transitioning to high by using high-fidelity components and tying into application data and functionality

Because this infrastructure has many key features in common with DENIM, it should be most efficient to extend DENIM to implement the new required features. By extending DENIM, many features including sketching, gestures, pie menus, zooming, and a run mode could be gained for free through reuse. This would save time and effort as compared to implementing these features from scratch. However, DENIM is not well-designed, so extending DENIM is not a feasible option. Instead, development will start from scratch so a new architecture is needed.

## 4.1 Towards An Architecture

The goals for the new architecture are ones that should be important to the design of any architecture. The first goal is to make the system extendable, so that new features can be added with minimal time, effort, and familiarity with past implementation details. An example feature that may be desirable to add in future versions is transitioning from high-fidelity prototypes to end-product implementa-

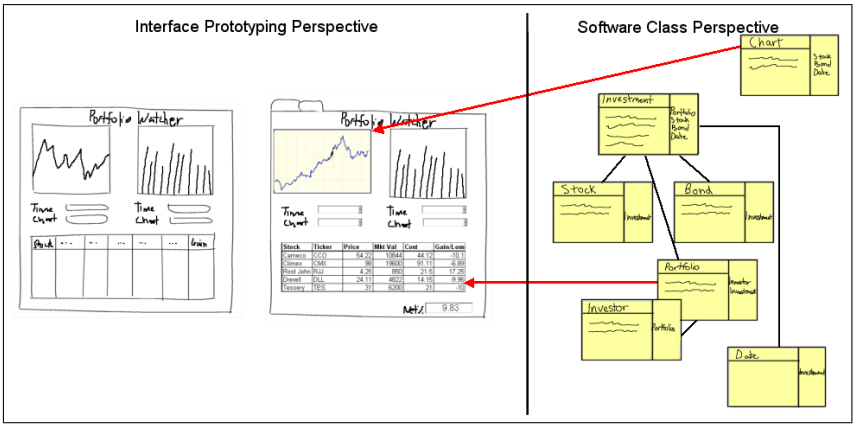


Fig. 5: Infrastructure supports mapping software artifacts (such as CRC Cards) and prototype components

tion. The second goal is for the system to be reusable. This means that not only should the system allow for the addition of new features to this domain, but possibly pieces of the system could be reused in different domains. For example, the sketching and gestures may be desired in other domain applications.

To accomplish the goals of extendability and reusability, the new architecture needs to clearly separate concerns of each module from others at an appropriate level, practice high cohesion and low coupling, as well as be easy to comprehend.

Prior to designing a new architecture, all features of the new infrastructure had to be made explicit. These features were then taken into account when drafting a new architecture. The features are as follows:

- Support for low- through high-fidelity prototyping, where low-fidelity prototypes may be sketched in the system or imported as images
- Mechanism for creating and linking prototypes to underlying application functionality and data
- Support for a run mode plus a design mode
- Support for mapping software and user interface models
- Multi-user interactions
- Pen-based input and use of pen-based widgets such as pie menus
- Mechanism for handling space management (layout and repositioning of ob-

jects, viewports for displaying scripts and data elements, different perspectives for software designers versus UI designers and for different modes)

- Support for creating and maintaining relationship lines (assumed human-directed rather than automatic)
- Timeline mechanism to support viewing or reverting back to any point in workspace history

## 4.2 A Proposed Architecture

The proposed high-level architecture is shown in Figure 6. It follows the hierarchical layers style of architecture, with the Device Input & Output at the bottom, then the Event-Handling, then the BlueWall Infrastructure Utilities, and then the Application Layer on top. A description of each layer will follow, with emphasis on the Application Layer.

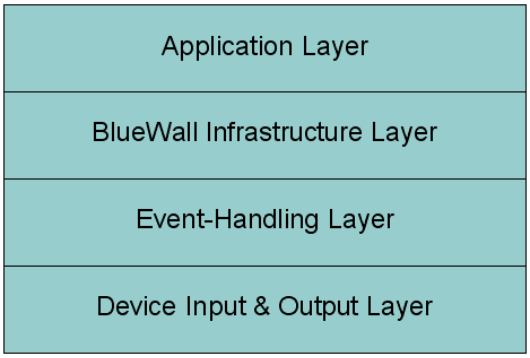


Fig. 6: High-level architecture is hierarchically layered to separate concerns

The Device Input & Output Layer handles core device input and display output issues. This layer should be designed to allow for stylus-based input, although other devices should be easily added as desired. This layer should also allow for multiple users to interact with the system simultaneously and thus handle input from multiple devices. Furthermore, this layer should support multiple displays, such as tiled large displays.

The Event-Handling Layer receives signals from the Device Input & Output Layer. The Event-Handling Layer defines what events are necessary in the system, and then layers above subscribe to the specific events they are interested in. When a

specific event is fired, the Event-Handling Layer then notifies all interested modules. This layer will also provide support for gesture recognition, so the higher layers can simply be informed of what gesture occurred.

The BlueWall Infrastructure Layer contains functionality that will be commonly required across all large display applications. Specifically, this layer handles all space management issues such as windowing of objects, flexible positioning of objects, zooming, and radar views. It also handles screen partitioning and maintaining of different perspectives within the partitions. This layer provides support for drawing relationship lines between objects, possibly across different perspectives. It also provides a historical record mechanism that allows for reverting back in time and making changes at any point. Furthermore, this layer also provides the Application Layer with specialized widgets such as pie menus.

The Application Layer contains any domain- or application-specific details. In this case, the application is an infrastructure for supporting low- through high-fidelity prototyping with tie-ins to software design. As such, this layer handles all aspects of creating prototypes, including mechanisms for tying in functionality and data with the prototypes, running the prototypes, and creation and mapping of software design artifacts. This layer will be described in more detail next.

***Application Layer in More Detail***

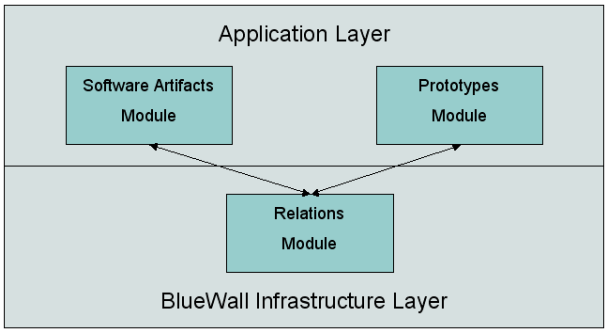


Fig. 7: High-level architecture diagram of the Application Layer

Figure 7 contains a very high-level architecture of the Application Layer for the prototyping infrastructure. The two main modules are the Prototypes Module and the Software Artifacts Module. The Prototypes Module contains all fidelities of prototypes (individual screens) as well as details on how these prototypes navigationally link, and how functionality and data tie into prototypes. The Software Artifacts Module contains all of the software design artifacts that software design-

ers' use. For example, CRC Cards would be created and/or stored here. The Relations Module of the BlueWall Infrastructure Layer bridges these two modules together since software artifacts may be associated with prototypes to help the UI and software designers to get a better understanding of the project as a whole. The Relations Module maintains these relations between prototypes and software design artifacts and provides support for drawing relationship lines.

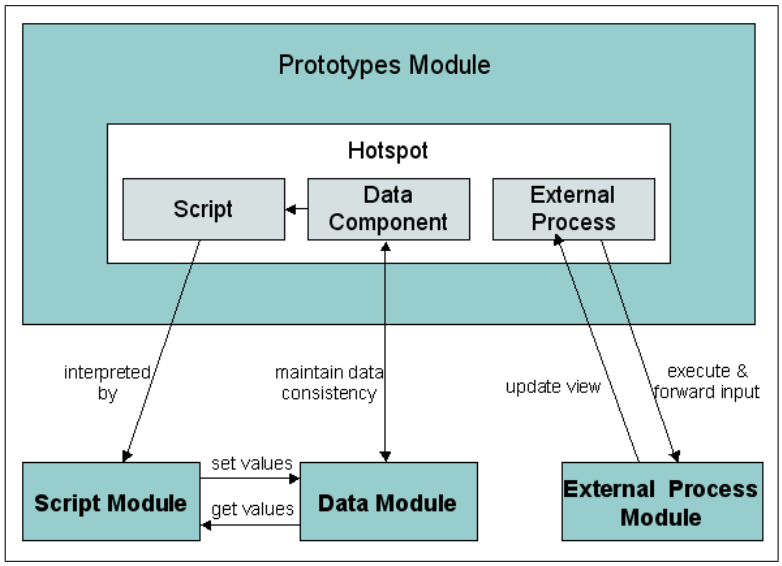


Fig. 8: More detailed architecture diagram of the Application Layer, focusing on how functionality and data is tied into prototypes

A more detailed architecture of the Application Layer is shown in Figure 8. Specifically, this figure depicts how each prototype contains hotspots, which are used to tie functionality or data into the prototype. Hotspots are active regions of a prototype defined by the designer that display data and/or perform a function, when interacted with, in run mode. Each region of a prototype may have multiple hotspots associated with it, through layering of hotspots, to generate different behaviors. Each hotspot is uniquely identified within the system so scripts can refer to desired hotspots, for example.

Each hotspot may have an assigned script, data component, or external process. Scripts are assigned to hotspots to create application functionality, including basic linking to navigate from one screen to the next. The scripts are interpreted by the Script Module. Data components are assigned to display data. These components can be fully interacted with and interactions with them can affect other hotspots.

Data components get their underlying data schema and data values from the Data Module. Data components may also have a script associated with it for situations when the data needs to be processed prior to displaying. External processes are assigned to display some external application, for example Microsoft Excel, and the External Process Module handles communication with the external process. External processes displayed in hotspots can be interacted with as allowed by the external process; however, interaction is isolated to that hotspot so the results of interacting with the external process cannot affect any other hotspots.

As an example, consider Figure 9, which illustrates a prototype containing several high-fidelity data components: a chart, a table, and a textfield. Each component is linked into the prototype through use of uniquely identified hotspots, which have been made visible in Figure 10. Each of these components is displaying the same underlying data obtained from the Data Module. Since they share the same data, changing a value in the table, for example, will automatically update both the graph and the textfield's display. Based on the architecture diagram in Figure 8 this will happen by the Data Module automatically sending update messages to all hotspots with data components that use the modified set of data (HS1, HS2, and HS3).

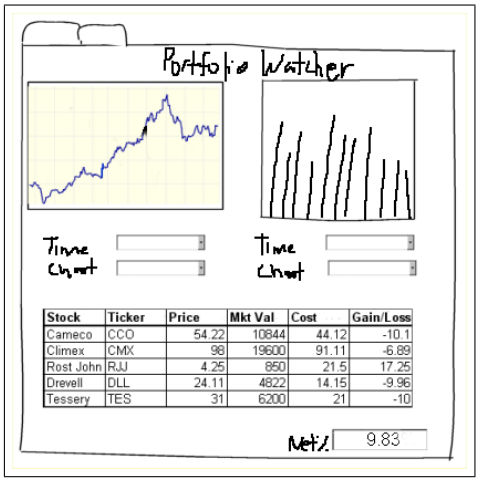


Fig. 9: A higher-fidelity prototype that displays data in a table, visualizes that same data in a line graph, and calculates a value based on that data, which is displayed in a textfield

Now consider that the textfield's displayed value is derived from the underlying data so some processing must occur before a value can be displayed. A script has been associated with the textfield to perform this processing. When the underlying data is changed, the textfield's hotspot (HS3), and specifically HS3's data compo-

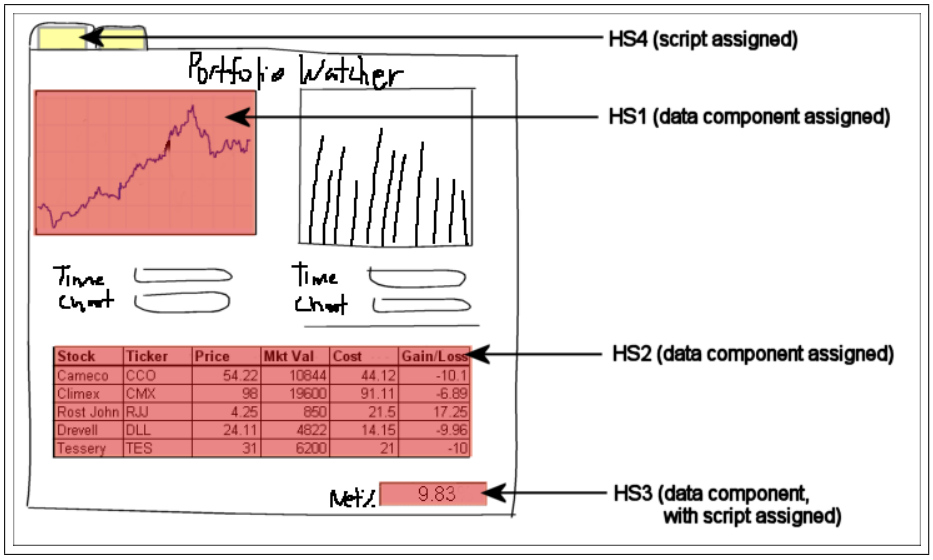


Fig. 10: Prototype from Figure 9 with hotspots made visible

nent, receives notification from the Data Module. HS3's script is executed upon notification and is interpreted by the Script Module, which gets necessary values from the Data Module and sets the resulting values in the Data Module. The Data Module then tells the textfield component to display the new value.

Furthermore, consider the hotspots on the tabbed panes of the prototype (such as HS4). They each have an assigned script which specify that a new screen is to be displayed when the hotspot is interacted with (in this case, clicked). When a hotspot is clicked, the script is interpreted by the Script Module and a new screen appears.

### 4.2.1 A Comparison to DENIM's Architecture

This proposed architecture shows promise for extension and reuse, in contrast to that of DENIM. This architecture uses a clean, layered structure to separate low-level concerns from application-specific concerns, which is something not successfully done in DENIM's architecture. For example, DENIM has event and graphics related details at the same layer in the architecture as domain-specific objects such as web pages. In this architecture, each layer only interacts with its neighboring layers, so it is more loosely coupled than DENIM. Using a layered architectural style allows for some of the layers to be reused in another system. For example,

non-domain specific functionality (BlueWall Infrastructure Utilities) may be easily reused in other pen-based, large display applications by simply replacing the application layer. Conversely, DENIM merges all non-domain specific functionality like zooming and sketching with domain-specific functionality. Furthermore, each layer in the proposed architecture can also be easily modified without affecting the entire system. In DENIM, a change to one class potentially breaks the entire system due to high coupling.

The proposed architecture also appropriately modularizes concepts within each layer as compared to DENIM's very fine-grained modularization. Having a higher level of modularization allows for the system to be more easily understood. For example, in DENIM's system it is not evident where one would find objects related to an individual prototype, whereas in this new architecture the prototypes are clearly found in the Prototypes Module. Thus this architecture is much more cohesive than DENIM's with closely related items packaged together rather than separately.

## 5 CONCLUSION AND FUTURE WORK

This paper proposed a high-level architecture for the prototyping infrastructure. The architecture is designed to be extendable and reusable, which DENIM's architecture failed at. The proposed architecture is a layered structure at the highest level, with the layers from the bottom up being Device Input & Output, Event-Handling, BlueWall Infrastructure, and finally the Application Layer. The paper presented a more detailed view of the Application Layer, and the Prototypes Module in particular, where a scenario of tying in application functionality and data was described in terms of the proposed architecture.

The architecture proposed in this paper is only preliminary and needs to be further refined. In particular, the Application Layer needs to have communication between the Script Module and Data Module as well as between the Data Module and Data Component explored. Also, the External Process communication layer needs to be further investigated. Once the Application Layer is refined, the other three layers also need to be designed, with emphasis on the BlueWall Infrastructure Layer.

## References

[1] BIDDLE, R., MARTIN, A., AND NOBLE, J. No name: just notes on software

reuse. *SIGPLAN Not. 38*, 12 (2003), 76–96.

[2] HEADWAY. reView Tool. Available at http://www.headwaysoft.com/.

[3] HONG, J. I., AND LANDAY, J. A. Satin: a toolkit for informal ink-based applications. In *Proceedings of the 13th annual ACM symposium on User interface software and technology* (2000), ACM Press, pp. 63–72.

[4] LANDAY, J. A., AND MYERS, B. A. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (1995), ACM Press/Addison-Wesley Publishing Co., pp. 43–50.

[5] LIN, J., NEWMAN, M. W., HONG, J. I., AND LANDAY, J. A. Denim: finding a tighter fit between tools and practice for web site design. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (2000), ACM Press, pp. 510–517.

[6] PAQUETTE, D. Source3d. Graphics Project, Available at http://www.cs.usask.ca/classes/880/t1/final.html, 2003.

[7] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (1972), 1053–1058.

[8] PETRIE, J. Supporting user interface prototyping using large interactive displays. 880 Research Project, Available at http://www.cs.usask.ca/research/research_groups/selab/, 2004.

[9] SHAW, M., AND GARLAN, D. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[10] TAYLOR, R. N., MEDVIDOVIC, N., ANDERSON, K. M., WHITEHEAD, JR., E. J., AND ROBBINS, J. E. A component- and message-based architectural style for gui software. In *Proceedings of the 17th international conference on Software engineering* (1995), ACM Press, pp. 295–304.