TOWARDS DECOMPOSABLE, REUSABLE SMALLTALK WINDOWS

Kevin McGuire, Paul White,
and Wilf R. LaLonde

SCS-TR-191, APRIL 1991

School of Computer Science, Carleton University
Ottawa, Canada, KIS 5B6

Towards Decomposable, Reusable Smalltalk Windows

Kevin McGuire†, Paul White†, and Wilf R. LaLonde*

School of Computer Science
Carleton University
Ottawa, Ontario Canada, K1S 5B6

Abstract

In Smalltalk, complex windows consisting of a large number of panes are often difficult to write and to modify. The present approach to window design tends to create very large models responsible for the application dependent behaviour of the entire window. Subparts of the windows cannot be extracted and allowed to retain any degree of their former behaviour.

We have expanded the model-view-controller (MVC) paradigm to allow for a more composite approach to window creation. We introduce "managers" to coordinate the communication between models and their views. Using these managers, individual panes or groups of panes can be given encapsulated behaviour that is highly independent of the context of their use, allowing them to be reused in a variety of applications. Decomposition makes window design both easier and faster.

1 Introduction

The MVC paradigm, typically sufficient for windows with one or two panes, tends to break down quickly even with relatively small browsers such as the Class Hierarchy Browser (CHB) in Smalltalk/V. Although object-oriented programming prides itself on code sharing and reuse, window applications tend not to reap these benefits. Our experience with writing numerous large browsers has motivated us to find a better approach to interface creation. We wanted to be able to create window systems using existing ones without having to relearn and rewrite all of the components. We also wanted to have the capability of reusing just parts of existing windows in perhaps different contexts within new windows. Ultimately, we would like to be able to construct windows using a module connection approach as in Fabrik [2] and InterCONS [5].

Often, groups of panes have a strong interconnection, and should be communicated with as one encapsulated object. What is required is to be able to view components of windows as objects unto themselves with their own consistent internal behaviour. These composite objects could be plugged into larger window applications and still retain their internal behaviour. The interaction of these composite objects with the application can then be supplied at window assembly time according to a strict and explicit communication protocol.

---

† Supported by an ORAE research contract.
* This research has been supported by NSERC.

2 Windows Are Composite Objects

As an example, consider the Class Hierarchy Browser (CHB) in Smalltalk/V (for this analysis, we will omit the ability to switch between instance and class methods). When a CHB is opened, this instance forms the model for all the panes in the window. Upon further analysis, one sees however that there are in fact many models involved: the class list, the method list, and either the class definition template, the method template, or the method text. In other words, the CHB is in fact a composite object — its parts are interdependent but nevertheless separate. The reason the CHB code is so large and unwieldy lies mainly in the fact that the browser model is too complex: it is actually several models coordinated by the unique browser model.

Now consider the introduction of a new type of browser for viewing instances found, say, in the Smalltalk dictionary. This "Instance Browser" consists of the CHB plus an extra pane which displays a list of the names of instances of the selected class. See Figure 1.

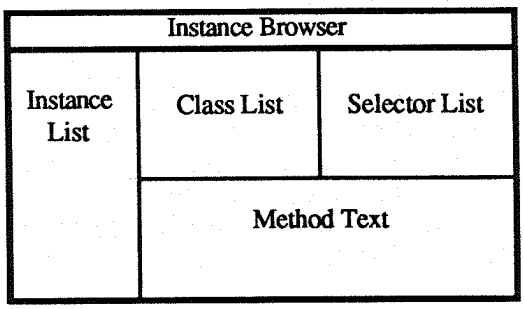

Figure 1. A new Instance Browser

The most common approach to creating our new Instance Browser class is to define it as a subclass of the Class Hierarchy Browser. Subclassing expresses an "is-a" relationship between the superclass and its subclass. There are two drawbacks to using subclassing in this case. First, it results in a class of object with more instance variables and more methods, thus yielding a larger, more complicated object. Second, it requires a deep understanding of the internal behaviour and thus the implementation of the superclass. In an object like the CHB, it is both unreasonable and undesirable to need that kind of deep knowledge. In fact, our experiences have shown that the usual approach is a best-guess solution. What often happens is that a developer is forced to re-implement a superclass' methods without understanding their details.

Even once a user fully understands the workings of a class such as the CHB, a large number of methods have to be re-implemented. This is a consequence of the use of the change/update mechanism. In the above example, changing the selected class will require updating the instance list. Consequently, all methods that change the selected class must now inform the instance list view of the change as well. Because of the large number of methods that directly or indirectly modify the browser's instance variables, it is generally not clear when and where the internal state of the browser is being changed.

A more appropriate approach is to consider the CHB to be "part-of" the Instance Browser. As a result, we no longer need to worry about the internal workings of the CHB. In fact, we now have but one real need — to know the communication protocol between these two browsers. This protocol consists not only of the selectors to which the CHB will publicly respond, but also the indirect protocol by which changes in the CHB are externally propagated (in this case to the

Instance Browser). The latter, although usually accomplished via the change/update mechanism, can in our case be performed more explicitly.

3 Propagating Side Effects — Explicit Dependency Management

When a new window is constructed, a mechanism is required to allow the components of the window to communicate. There are three different levels of communication. First, when a change is noted by the view/controller, it must inform its model of the change. Second, if the model changes, the view being used to display this model must be notified. Third, changes made in a model within a window must be directed to other models which are dependent upon it.

In the current Smalltalk/V implementation, these changes are rolled into the methods written for the one browser class. As mentioned above, this makes understanding these methods difficult, and makes it virtually impossible to inherit or reuse them. It is clear that one needs some mechanism by which window components can be connected together so that changes in one model can properly propagate changes in dependent models and views.

The MVC paradigm makes use of a somewhat complicated method of notifying the window of changes in the model. This method requires the window to be registered as a dependent of the model. Whenever the model changes, it informs the world of this fact by broadcasting an "update" message to its dependents.

This change/update mechanism has several drawbacks. For one, it hides the relationship between the model and the window and thus obscures the working mechanism from the user. Secondly, since all dependents of an object get informed of the change, in a case such as the CHB where all panes are dependents of the CHB model, a significant amount of message processing gets needlessly performed (each pane must process the change message to determine if it is affected or not).

Although systems such as Glazier [1] have addressed this problem by allowing dependents of instance variables, the problem can better be solved by facilitating explicit dependent handling according to selected messages received by the model. When the model receives one of these messages, a decision table can be used to determine what messages need to be sent to the concerned dependents. EVA [4] also presents a different approach to this by having all subviews within a window "register" for the events received by the window to which it cares to respond. Although this does provide a degree of visibility to dependence management, it does not address many of the issues involved in building composite windows. Furthermore, EVA requires drastic modifications to the Pane/Dispatcher classes.

Another shortcoming of the existing change/update mechanism is that it only allows for one change selector. However, many cases exist where the action to be taken is dependent upon the way in which something has changed. For instance, a window application might want to be told of both mouse selects and mouse releases in order to be able to implement dragging. At present, to implement this requires the model reading the keyboard, an action that strictly speaking is the responsibility of the controller and is therefore a violation of the MVC paradigm.

4 Building Browsers Using Managers

We are proposing a new way to build browsers within Smalltalk. Browsers will consist of a series of **model-view managers**, or managers for short. Managers are a specialization of coordinators as described in [3]. They can either be simple or composite. Simple managers coordinate the

interaction between one pane and its model; composite managers coordinate the interaction between a series of either simple or composite managers.

Simple managers form a bridge between a view within a window and the data being modelled. There will now be a one-to-one correspondence between each view and its model, and the interaction between the view and its model is the responsibility of the manager. This is different than the usual approach of having just one model for all the panes within a window. For example, in the Instance Browser, a simple manager can be used to coordinate the interaction between the list of instances for the selected class and the pane displaying the list. Figure 2 shows the relationship between these three objects.

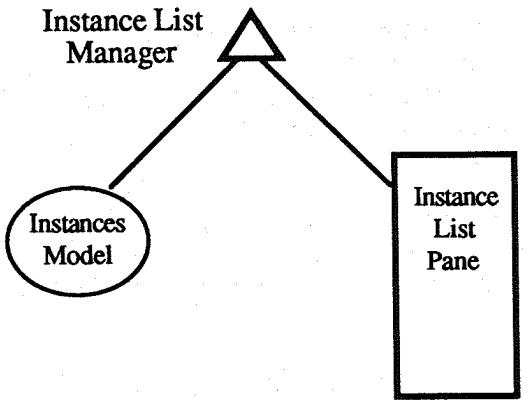

Figure 2. An example of a simple manager

One of our design criteria was to be able to use the existing Pane and Dispatcher family of classes without modification. Additionally, we wanted to eliminate the use of the 'name' and 'change' selectors by application writers. The manager can communicate with the pane using the traditional 'name' and 'change' mechanisms, while using more application specific protocols for communicating with the model. Since the manager forms the model for the view, it is appropriate for it to hold the state information with respect to both the modelled data and the view. For example, in the class list view in the CHB, the manager would store the 'classList' and the 'selectedClass' for the pane.

Composite managers provide us with the ability to encapsulate a group of views. They coordinate the interaction between a collection of managers, both simple and composite. Since they don't interact directly with the pane classes, they do not have to follow the 'name' and 'change' protocol for reporting updates to other interested managers. Instead, they can explicitly describe the interaction between the submanagers using a table of messages being received and what action to take when it is received. The protocol for communication between a composite manager and each of its submanagers is also explicit. In essence, composite managers form the "glue" for building windows.

To illustrate the use of managers, let us show the managers involved in building the Instance Browser described earlier (see Figure 3). First, the class hierarchy browser can be represented using one composite manager (with minimal effort). The instance list being added can be coordinated using a simple manager. The interaction between these two managers is the responsibility of the new composite manager, referred to as the Instance Browser Manager. It stores no state, but instead can obtain the information it needs from its two submanagers. Changes in the instance list or the class list will be forwarded to the Instance Browser manager, who must

decide what action must be taken. In the case of a new class selection, it will inform the instance list to update itself, whereas in the case of a new instance selection, no update of the class list is required. Note that any other changes within the CHB (such as method selection, changes in the text editor, etc.) are not reported to the Instance Browser manager, since they are encapsulated within the CHB itself.

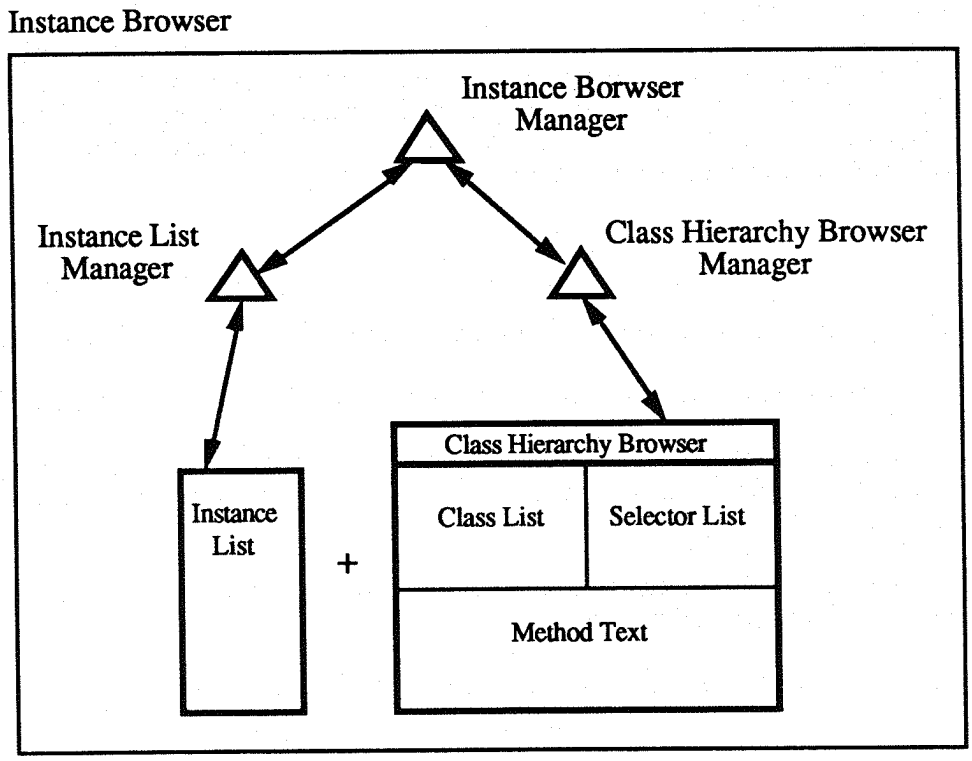

Figure 3. The components of the Instance Browser

5 Conclusions — Future Work

We discovered that the MVC paradigm as it is currently used within Smalltalk/V is inadequate for developing large browsers. It was not possible to reuse existing windows as parts of new, more complex windows. The existing browser classes are too complex to allow for subclassing or code sharing. The interaction between views within these browsers is buried within the code as opposed to explicitly represented as part of the browser itself.

Managers permit a more modular approach to window design. The change/update mechanism is replaced with a more explicit and versatile protocol. It is now possible to build large windows out of existing windows without having to modify the existing code for the window. Libraries of window modules can now be built. Although there is an overhead involved in the initial design of these base managers, our experience indicates that most browsers can be designed as composites of these basic modules.

As proof of concept, we have built a Hierarchical List Manager — a manager for controlling hierarchical lists such as class lists and directory structures. These managers have proved sufficiently versatile to be used in a variety of contexts. We also used managers to coordinate the interaction between two views of the same data in the same window. The manager was used to allow changes made in one of the views to be reported to the other view. In this case, the views were an instance list and a graph pane with icons. The significant point was that no changes to the implementation of the model were required. Furthermore, the manager itself was but a component of a larger, more complicated window.

Eventually we would like to create a window construction kit to graphically handle the connection of window managers. Since the communication protocol between managers is explicit, these selectors can be represented in lists from which the user can select. Side effect management and propagation would be more visible and easier to describe.

References

1. Alexander, J.H. Painless panes for Smalltalk windows. *OOPSLA '87*, pp. 287-294.

2. Ingalls, D. et. al. Fabrik: A visual programming environment. *OOPSLA '88*, pp. 176-190.

3. Lalonde, W., White, P. and McGuire, K. Coordinators: A Mechanism for Monitoring and Controlling Interactions Between Groups of Objects. Carleton University School of Computer Science Technical Report.

4. McAffer, J. and Thomas, D. EVA: An event driven framework for building user interfaces in Smalltalk. *Graphics Interface '88*, pp. 168-175.

5. Smith, David N. Building interfaces interactively. *SIGCHI '88*, pp. 144-151.

School of Computer Science, Carleton University
Recent Technical Reports

SCS-TR-151 **On Doubly Linked List ReOrganizing Heuristics**
D.T.H. Ng and B. John Oommen, February 1989

SCS-TR-152 **Implementing Data Structures on a Hypercube Multiprocessor, and Applications in Parallel Computational Geometry**
Frank Dehne and Andrew Rau-Chaplin, March 1989

SCS-TR-153 **The Use of Chi-Squared Statistics in Determining Dependence Trees**
R.S. Valiveti and B.J. Oommen, March 1989

SCS-TR-154 **Ideal List Organization for Stationary Environments**
B. John Oommen and David T.H. Ng, March 1989

SCS-TR-155 **Hot-Spot Contention in Binary Hypercube Networks**
Sivarama P. Dandamudi and Derek L. Eager, April 89

SCS-TR-156 **Some Issues in Hierarchical Interconnection Network Design**
Sivarama P. Dandamudi and Derek L. Eager, April 1989

SCS-TR-157 **Discretized Pursuit Linear Reward-Inaction Automata**
B.J. Oommen and Joseph K. Lanctot, April 1989

SCS-TR-158 (revised) **Parallel Fractional Cascading on a Hypercube Multiprocessor**
Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, May 1989 (Revised April 1990)

SCS-TR-159 **Epsilon-Optimal Stubborn Learning Mechanisms**
J.P.R. Christensen and B.J. Oommen, June 1989

SCS-TR-160 **Disassembling Two-Dimensional Composite Parts Via Translations**
Doron Nussbaum and Jörg-R. Sack, June 1989

SCS-TR-161 (revised) **Recognizing Sources of Random Strings**
R.S. Valiveti and B.J. Oommen, January 1990
Revised version of SCS-TR-161 "On the Data Analysis of Random Permutations and its Application to Source Recognition", published June 1989

SCS-TR-162 **An Adaptive Learning Solution to the Keyboard Optimization Problem**
B.J. Oommen, R.S. Valiveti and J. Zgierski, October 1989

SCS-TR-163 **Finding a Central Link Segment of a Simple Polygon in O(N Log N) Time**
L.G. Alexandrov, H.N. Djidjev, J.-R. Sack, October 1989

SCS-TR-164 **A Survey of Algorithms for Handling Permutation Groups**
M.D. Atkinson, January 1990

SCS-TR-165 **Key Exchange Using Chebychev Polynomials**
M.D. Atkinson and Vincenzo Acciaro, January 1990

SCS-TR-166 **Efficient Concurrency Control Protocols for B-tree Indexes**
Ekow J. Otoo, January 1990

SCS-TR-167 **A Hierarchical Stochastic Automaton Solution to the Object Partitioning Problem**
B.J. Oommen, January 1990

SCS-TR-168 **Adaptive List Organizing for Non-stationary Query Distributions. Part I: The Move-to-Front Rule**
R.S. Valiveti and B.J. Oommen, January 1990

SCS-TR-169 **Trade-Offs in Non-Reversing Diameter**
Hans L. Bodlaender, Gerard Tel and Nicola Santoro, February 1990

SCS-TR-170 **A Massively Parallel Knowledge-Base Server using a Hypercube Multiprocessor**
Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, April 1990

SCS-TR-171 **Parallel Processing of Quad Trees on the Hypercube (and PRAM)**
Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, April 1990

SCS-TR-172 **A Note on the Load Balancing Problem for Coarse Grained Hypercube Dictionary Machines**
Frank Dehne and Michel Gastaldo, May 1990

SCS-TR-173 **Self-Organizing Doubly-Linked Lists**
R.S. Valiveti and B.J. Oommen, May 1990

SCS-TR-174 **A Presortedness Metric for Ensembles of Data Sequences**
R.S. Valiveti and B.J. Oommen, May 1990

SCS-TR-175 **Separation of Graphs of Bounded Genus**
Ljudmil G. Aleksandrov and Hristo N. Djidjev, May 1990

SCS-TR-176 **Edge Separators of Planar and Outerplanar Graphs with Applications**
Krzystof Diks, Hristo N. Djidjev, Ondrej Sykora and Imrich Vrto, May 1990

SCS-TR-177 **Representing Partial Orders by Polygons and Circles in the Plane**
Jeffrey B. Sidney and Stuart J. Sidney, July 1990

SCS-TR-178 **Determining Stochastic Dependence for Normally Distributed Vectors Using the Chi-squared Metric**
R.S. Valiveti and B.J. Oommen, July 1990

SCS-TR-179 **Parallel Algorithms for Determining K-width-Connectivity in Binary Images**
Frank Dehne and Susanne E. Hambrusch, September 1990

SCS-TR-180 **A Workbench for Computational Geometry (WOCG)**
P. Epstein, A. Knight, J. May, T. Nguyen, and J.-R. Sack, September 1990

SCS-TR-181 **Adaptive Linear List Reorganization under a Generalized Query System**
R.S. Valiveti, B.J. Oommen and J.R. Zgierski, October 1990

SCS-TR-182 **Breaking Substitution Cyphers using Stochastic Automata**
B.J. Oommen and J.R. Zgierski, October 1990

SCS-TR-183 **A New Algorithm for Testing the Regularity of a Permutation Group**
V. Acciaro and M.D. Atkinson, November 1990

SCS-TR-184 **Generating Binary Trees at Random**
M.D. Atkinson and J.-R. Sack, December 1990

SCS-TR-185 **Uniform Generation of Combinatorial Objects in Parallel**
M.D. Atkinson and J.-R. Sack, January 1991

SCS-TR-186 **Reduced Constants for Simple Cycle Graph Separation**
Hristo N. Djidjev and Shankar M. Venkatesan, February 1991

SCS-TR-187 **Multisearch Techniques for Implementing Data Structures on a Mesh-Connected Computer**
Mikhail J. Atallah, Frank Dehne, Russ Miller, Andrew Rau-Chaplin, and Jyh-Jong Tsay, February 1991

SCS-TR-188 **Generating and Sorting Jordan Sequences**
Alan Knight and Jörg-Rüdiger Sack, March 1991

SCS-TR-189 **Probabilistic Estimation of Damage from Fire Spread**
Charles C. Colbourn, Louis D. Nel, T.B. Boffey and D.F. Yates, April 1991

SCS-TR-190 **Coordinators: A Mechanism for Monitoring and Controlling Interactions Between Groups of Objects**
Wilf R. LaLonde, Paul White, and Kevin McGuire, April 1991

SCS-TR-191 **Towards Decomposable, Reusable Smalltalk Windows**
Kevin McGuire, Paul White, and Wilf R. LaLonde, April 1991