# COORDINATORS: A MECHANISM FOR MONITORING AND CONTROLLING INTERACTIONS BETWEEN GROUPS OF OBJECTS

Wilf R. LaLonde, Paul White,
and Kevin McGuire

SCS-TR-190, APRIL 1991

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

# Coordinators: A Mechanism for Monitoring and Controlling Interactions Between Groups of Objects

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

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

## Abstract

*Coordinators are mechanisms for explicitly specifying the important message traffic between groups of interacting objects and subsequently monitoring, intercepting, and massaging this message traffic to ensure proper communication. In general, they are useful in applications that require monitoring, statistics gathering, and coordination. Nevertheless, they were designed primarily to solve some outstanding problems in user-interface classes that hinder reusability.*

*Coordinators provide an explicit description of the important message traffic enabling users to quickly understand the important interaction messages. It replaces the change/update mechanism in the model-view-controller paradigm with one that makes the change messages more explicit. It permits arbitrary models to be plugged-in without having to instrument them with "self* **changed:***" messages. It also permits new interaction structures to be constructed out of simpler building blocks and makes the building blocks more reusable. For example, it would permit components of complex windows like browsers to be reused in application specific browsers without having to duplicate the inherent functionality.*

## 1 Introduction

As larger and more complex application libraries are developed, the need for highly integrated communities of interacting objects will grow. For example, a garage application might need to provide models for the garage itself, the parts department, the service department, and the repair centre (to name a few). A restaurant application might need to integrate the reservations desk with the kitchen and food supplies. A better known integrated collection of objects are the model-view-controller classes in the Smalltalk user interface library. Each of the components in an integrated community is generally designed to interact with the others (where necessary) through a specially designed protocol.

A major problem is how to reuse these integrated classes. In some cases, the classes are designed explicitly for reuse; viz. Smalltalk's pluggable views that are intended to permit users to plug in their own models and to interchange models and views, perhaps even replacing them by user designed components. In general, however, designers cannot forsee all potential uses. More often than not, the result in hindsight is far less reusable than originally expected.

So as to make our arguments more concrete, we will use the model-view-controller (MVC) classes as a source of inspiration. We will investigate some of the current barriers to reusability in these classes and suggest what would be needed to eliminate the inadequacies. These user interfaces classes are particularly appropriate for this investigation since they are one of the most complex triplet of classes ever designed and one of the richest sources of new ideas.

First, we introduce the notion of a coordinator at a conceptual level in order to provide a unified framework for subsequent discussion. Next we consider a simple MVC example and consider problems that might arise when a designer/implementer attempts to reuse it in an application specific context. Third, we discuss some problems that we encountered from our own experiences that led to the notion of coordinators. Lastly, we consider implementation techniques for providing the capabilities attributed to coordinators.

---

* This research has been supported by NSERC.

† Supported by an ORAE research contract.

## 2 Coordinators: Conceptual Definition

A **coordinator** is an object designed specifically to coordinate the message traffic between collections of interrelated objects. It must have the capability to

- explicitly store messages of interest and what to do when they are observed,
- monitor and intercept the inter-object messages of interest,
- perform the activities associated with the messages when they occur,
- dynamically add and remove members to the collection of communicating objects, and
- dynamically add and remove messages of interest.

To properly monitor the message traffic, the sender, message selector, and receiver are all needed in the general case. In a special situation, however, any of the three components may be omitted; i.e., treated as a don't care. This information can be provided in one of two ways: (1) by having the communicating object send it directly to the coordinator (the most likely scenario) or (2) by using encapsulators [7] or indirection objects [4] to transparently monitor all the messages to a particular object. When a message of interest is intercepted, an arbitrary computation associated with the message may be performed. In the simplest case, the message is simply relayed to its intended destination — this also happens if the message is removed from the list of messages of interest. The next simplest case is to relay the message using an alternate message selector. More complicated cases include pre- and post-processing before relaying it (as provided by systems supporting active values [5, 6, 12]) or relaying it to multiple destinations.

The notion of coordinators mentioned above permits many implementations and many protocols that satisfy the general requirements. Some of them might only provide a subset of the general capabilities. It is also possible to design hierarchies of coordinator classes each with different protocols and capabilities. Which one is used would depend on the coordination requirements. In some cases, coordinators might be transparent to the community being monitored. In other cases, the members of the community can be specifically designed to communicate through the coordinator. This latter approach is likely to be the easiest and most efficient way to provide coordinators. Coordinators are an extension and elaboration of the manipulator concept introduced in Glazier [1]. The Glazier notion of a manipulator, by contrast, requires a new manipulator class to be constructed for each new window configuration and model. It is not possible to use a generic manipulator since extensive methods must be added.

## 3 Some Problems That Coordinators Can Solve

Let's imagine that we have a new implementation of a menu view and a menu controller, both designed to interact with a coordinator (see Figure 1). In Smalltalk, they are called list views and controllers respectively. The pair is designed to provide a list of items (obtained from the model) in a window and permit users to select one of the items in the list by pressing the mouse over the desired item. The window can be scrolled when more items exist than can be displayed in the available space. The coordinator is designed to intercept messages from the controller (or view) to the model to get the menu list and to indicate which item has been selected (0 for no selection, 1-n for a selection). In order to gain a better understanding of the benefits of coordinators, we consider three imaginary scenarios that coordinators were designed to handle easily. For comparison purposes, we also provide a comparable solution to the problem (if it exists) using the standard MVC approach.

**Problem 1:** A user wishes to use menu windows for his own application but he's not sure of the protocol he will need to understand in order to reuse the basic structure. In the absence of documentation, the simplest solution is to observe an example coordinator. Since the important message traffic is explicitly encoded in the coordinator, inspecting it will normally be sufficient to determine the needed information.

By constrast, the more traditional MVC design without coordinators is extremely difficult to unravel. The obvious reason is that the the important message traffic is imbedded in the implementation code. Moreover, it is difficult to distinguish between the important message traffic (the messages that must be understood to reuse the design in new contexts) and messages that are internal to the implementation; e.g., private communication between the view and controller. Another reason is that some of the communication is implicit; e.g., the communication from the model to the view is implemented using a powerful change/update mechanism. The model simply says "I've changed" and all views on that model are broadcast the message "update yourselves". Each view then explicitly asks its model

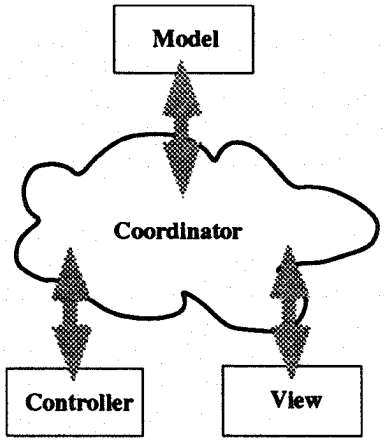


**Figure 1** A menu window designed to interact through a coordinator.

for new display information. For this to happen, it is sufficient to register each model when it is connected to a view — the process simply adds the view to a set of views associated with the model — sometimes kept in a global identity dictionary and sometimes kept in one that is local to the model. Closing the window causes automatic deregistration — the view releases or decouples itself from the model. To summarize, unravelling the important message traffic is extremely difficult with the traditional design.

**Problem 2**: A user wishes to use an existing application object as the model; e.g., he might wish to provide a menu window on pizza orders for a small pizza shop application. It is a simple matter to plug in the application model. However, some messages to the application can result in modifications to the pizza order. The modifications must be reflected in the window displaying the pizza order. To achieve this, messages to the pizza order (the model) must be monitored to cause "update" messages to be sent to the view whenever the modification messages are observed. Moreover, this monitoring must be transparent since the messages can come from any object in the system.

By contrast, even though pluggable views in the more traditional design permits application objects to be "plugged in", this in itself is not enough. The implementor must physically add "self **changed**" messages to all methods in the application object that can result in modifications that should be reflected in the view. Over time, many application objects would have to be so instrumented. Clearly, the proliferation of such modifications (even when they behave as no-ops when there is no window using them as models) is undesirable. A less intrusive approach can be used in systems that provide active objects; i.e., systems that permit pre- and post-processing to be performed when selected methods are executed. Nevertheless, the user is responsible for adding and removing the additional processing steps because there is no mechanism for explicitly encoding what needs to be undone when associated windows are closed.

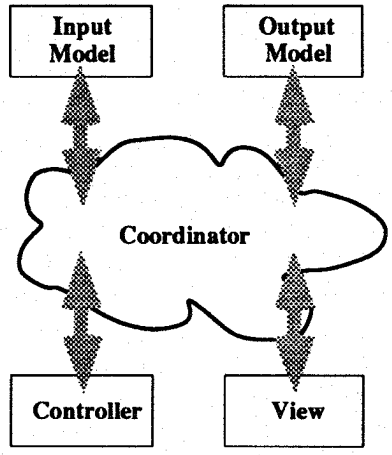


**Figure 2** A coordinator permitting interaction with two models.

**Problem 3**: A user might wish to reuse the design in a situation that was totally unexpected. For example, to build an application specific browser, it might be necessary to use two models (see Figure 2) rather than one — an input model from which the menu list can be obtained and an output model to which the user's selection ought to be reported. Without making any changes to the design, it is sufficient to use the input model in the manner specified by the original design. When a user selection is made, the message that would normally be sent to the input model is re-routed by the coordinator to the output model.

By contrast, the standard MVC design cannot be extended to two models unless a new class of objects is designed specifically for the task; e.g., a virtual model for the two real models. This virtual model can be designed to perform the necessary re-routing. In this case, the user is clearly designing an application specific coordinator. Building a new class of models is fundamentally less desirable than plugging in rerouting information in an existing coordinator. More important, this special class is not likely to be reusable. Each new unexpected extension is likely to be application specific. Note: in an advanced system, we might want to design a special class of coordinators; e.g., if it serves to provide a capability that is a useful addition to the coordinator sub-library or if there are efficiency concerns that make the specialization desirable. However, it ought to be an explicit choice on the part of the user — not a requirement.

In general, coordinators can be used to monitor traffic to, from, and between arbitrary objects. Hence they can be used for performance monitoring, statistics gathering, and debugging. Indeed, it ought to be possible to take existing window classes such as an inspector window, build a coordinator that monitors a specific object (without modifying the object), and have information dynamically updated in the window whenever messages of interest are received by the monitored object. Such reusability is simply not possible with the existing window paradigms (in any system).

## 4 Experiences That Led To The Notion of Coordinators

The idea for coordinators originated from our experiences in building different kinds of browsers in Smalltalk. We observed that very little code could be reused, even though many of the browsers shared common features. Even in the cases where new windows were created using existing windows as "subparts", it was virtually impossible to simply "port" the old into the new. The reason, we discovered, was that browsers were traditionally designed using a super-model specific to each browser (a new class that can be used as the model for all subviews). All of the behaviour of the window is provided in this one class. This is a problem because most of these windows don't browse just one object, but instead each subwindow typically provides a view on some component of the information being displayed.

Consider a class hierarchy browser found in Smalltalk V (similar remarks also apply to Smalltalk-80). The model for each of the panes[1] is an instance of the ClassHierarchyBrowser class itself (that's why we refer to it as a super-model). But of course this isn't correct. If you consider the three primary panes of the browser, namely the class list, the method list, and the edit pane, each of these models is a different "object". The class list is really a view on the list of subclasses of class Object. The method list is really a view on the list of methods of the class selected within the class list. The edit pane models either the class selected within the class list, or the method selected within the method list, depending upon which of the two lists was last activated. In the current implementation of this class, the interdependencies between the panes are "hardwired". Each time an action occurs that causes a modification to the contents of the browser, the instance variables for the browser (the instance of ClassHierarchyBrowser) must be set explicitly, and then the appropriate change messages must be sent to each of the panes.

Reuse becomes an issue when we attempt to use part of the existing behaviour of the class hierarchy browser. For example, assume we want to build an instance browser — a window containing just two lists: the first list containing the classes in the system, and the second containing the names of global variables bound to instances of the class selected in the first pane. This can be easily determined by scanning dictionary Smalltalk which contains all global variables. The class list pane should have the same behaviour as in the class hierarchy browser. Once a class is selected from the class list, a list should be displayed of all the instances of this class.

So, why require coordinators? First, consider implementing this without using coordinators. The first thought is to subclass ClassHierarchyBrowser, and try to inherit the existing behaviour. This is a problem, though, because a large number of the existing methods would have to be modified since the relationships between the existing panes is hardwired throughout the methods. Another choice is to try to copy the needed methods from the ClassHierarchyBrowser into a new class. But of course, this is the wrong approach since what we want has already

---

[1] Smalltalk V terminology for view.

been written. Even so, if we do copy, it is often difficult to determine what methods to leave out; i.e., that are superfluous to the new browser. For example, consider removing the code that provides the hide/show behaviour in the class list from the ClassHierarchyBrowser class in Smalltalk V/286!!

We found that the proper way to implement browsers so that we could reuse the code was to develop each pane independently, and have their interdependent behaviour controlled by coordinators. Thus, the class hierarchy browser would be viewed as having three separate models. A coordinator would intercept messages sent to any of the three coordinated objects, and notify the other "panes" when a change was made. For example, when the class list receives the message 'addSubclass', the coordinator can recognize that this message will cause a change in the list of classes, and that both of the other panes will have to change. Moreover, the coordinator can determine what type of change is required of the others, not just the fact that something has changed. In this manner, using coordinators effectively allows us to replace the current change/update mechanism with one that is both more explicit and more general. Instead of a model being forced to simply say "I've changed", the coordinator can determine exactly what the change was, and the appropriate response engaged.

Note that coordinators can be part of the abstraction for a particular class of windows. When they are integrated as components of other windows, designers have a choice for the top level coordinator: (1) coordinate with the existing lower-level coordinators (i.e., maintain the lower-level abstractions, avoid decomposing them) or (2) integrate the lower-level coordinators into the top level and discard the lower level coordinators. This new design dimension may prove to be an advantage — but more experience is needed to determine that.

Another problem we found with implementing browsers this way was that it was impossible to reuse existing browsers as parts of other browsers. For example, suppose we want to create a new class browser that consists of three list panes instead of two, where the third list contains the names of global variables bound to instances of the selected class. That is, once a class is selected from the class list, the methods defined for the class are shown in the methods list as before, but additionally, the globally named instances are shown in the third pane. When a user selects one of these globally named instances, an inspector on that instance is opened. Clearly, a powerful mechanism and organizational structure was needed to promote rather than hinder reusability.

## 5 Implementation Strategies

In general, the target system used to implement coordinators will influence the implementation strategies and in some cases limit their capabilities. For example, a system like C++ [11] can easily be provided with coordinators that require explicit communication between the coordinated objects and the coordinator. However, it is substantially more difficult to provide it with efficient mechanisms for transparently monitoring objects. More powerful systems like Smalltalk [3] or Trellis/Owl [9] with built-in garbage collection and mutation operations can easily provide the notion of encapsulators [7] and proxies [4] for trapping messages entering and leaving selected objects. Loops [2, 10] with its notion of active values can similarly perform the same task. Class based systems, however, do have a slight disadvantage compared with newer generation systems like Self [13] that is based on instance inheritance and prototypes. The important message traffic and the re-routed destinations cannot be easily determined from static comments (they can be out of date) or methods that explicitly construct a coordinator (the code can be difficult to read). It is most convenient to browse an existing example. Exemplar or prototype-based systems like Self require the designer to construct examples just to provide it with the behavior methods. Hence the examples are available for inspection. Additionally, it is easier to build new coordinators by copying and subsequently modifying existing coordinated groups. Transparent monitoring can also be provided easily in law-based systems [8] because they are explicitly designed to permit transparent message monitoring.

To briefly review how encapsulators are implemented in Smalltalk, consider an object O to be monitored and an encapsulator E. To begin with, O and E are mutated into each other so that all messages sent to O actually go to E. Encapsulator E explicitly keeps track of O. Moreover, the encapsulator is explicitly designed with almost no protocol — it does not even inherit from Object. However, it does provide a method for handling **doesNotUnderstand:**. When an arbitrary message originally intended for O is received by E instead, the message is not understood and it is therefore processed by the **doesNotUnderstand:** method. The method is provided with one parameters, a **Message** object that contains both the message selector and the message arguments. With this information, the encapsulator can execute pre-processing code, send the original message to O, execute post-processing code, and finally return the result to the original sender. Note that the sender of the message originally intended for O can be easily determined by the receiver (there are explicit messages in Smalltalk for getting this information) so that all information needed by a coordinator is available.

It is clear that a coordinator can simply maintain a list of elements each consisting of a message selector, a sender (potentially don't care), a list of receivers with corresponding associated actions, and an indicator that specifies whether or not the monitoring is to be implicit (invisible) or explicit. The coordinator itself must determine which objects (if any) require encapsulators and must create them when needed. Closing the window must undo the changes that were made. Indeed, a robust implementation should still work even if the application window fails to close itself properly. It is a relatively simple task for the coordinator to determine if the window is still scheduled for execution. If it is not, it can undo all the changes invisibly. Such coordinators, however, would only make sense for top level views (scheduled windows). Hence a small set of distinct coordinator classes organized in a straigthforward hierarchy would be useful.

In general, the hierarchy could provide coordinators with different levels of capability. For example, one class of coordinators might permit arbitrary blocks as the actions associated with messages. Another might require only a message selector. A general coordinator might also permit both. Of course, there is no point having specialized subclasses unless there is an implementation or conceptual advantage.

**Efficiency Considerations:** In order to be as efficient as possible, we would likely design our windows so that views and controllers interact directly with the coordinator. This is not possible, however, with the models. In that case, less efficient encapsulators must be used. Overall, this general strategy should be acceptable for the design of user interface components since the bottleneck in the man/machine loop is usually the "man".

## 6 Conclusions

As designers and programmers tackle larger and more complex application areas, it becomes more and more important to reuse parts of existing libraries. Often, components are reused in ways that were not anticipated by the original designers. More often, however, components can almost be reused but not quite. Coordinators were designed to help alleviate this problem — to remove some of the barriers to effective reusability. Their major innovation is making this message traffic explicit. Notions like encapsulators, active values, and manipulators provide a suitable substrate for implementing encapsulators which itself is a higher level abstraction.

In general, coordinators are not super-controllers — they are distinct from controllers since they play a different role. However, controllers could be made obsolete by merging them with their associated views as was done in Smalltalk/V running on OS/2's Presentation Manager. With the view/controller pair replaced by a single window object, the coordinator is still needed to permit effective pluggability and reusability. Neither are coordinators super-models, sometimes referred to as a model for a model since they are primarily intended for application independent reconfiguration. Why, for example, would a super-model monitor some other object?

As part of our ongoing research, we are developing a prototype of coordinators. The prototype will serve as a "proof of concept" to ensure that coordinators can actually be implemented and integrated into an existing environment, to develop further extensions to the notions, and to verify our intuitions about their usefulness for promoting reusability.

## References


1. Alexander, J.H. Painless panes for Smalltalk windows. *OOPSLA '87*, Orlando Florida, Oct. 1987, pp. 287-294.
2. Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. CommonLoops: Merging Common Lisp and object-oriented programming. OOPSLA '86, Portland, Oregon (September 1986), 17-29.
3. Goldberg, A. and Robson, D. *Smalltalk-80: The language and its implementation.* Addison-Wesley, Reading, Mass., 1983.
4. McCullough, P.L., Transparent forwarding: first steps, *OOPSLA '87*, Orlando Florida, Oct. 1987, pp. 331-341.
5. Murata, M. and Kusumoto, K., Daemon: another way of invoking methods, *Journal of Object-Oriented Programming*, Vol. 2, No. 2, July 1989, pp. 8-12.
6. Osterbye, K., Active objects: an access-oriented framework for object-oriented languages, *Journal of Object-Oriented Programming*, Vol. 1, No. 2, June 1988, pp. 6-10.
7. Pascoe, G.A., Encapsulators: a new software paradigm in Smalltalk-80, *OOPSLA '86*, Portland Oregon, Sept. 1986, pp. 341-346.
8. Rozenshtein, D. and Minsky, N.H., A law-governed object-oriented system, *Journal of Object-Oriented Programming*, Vol. 1, No. 6, March 1989, pp. 14-29.

9. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. *An Introduction to Trellis/Owl.* OOPSLA '86, Portland, Oregon (September 1986), 9-16.
10. Stefik, M.J. and Bobrow, D.G. *Object-oriented programming: Themes and variations.* AI Magazine, Vol. 6, No. 4 (1986), 40-62.
11. Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, 1986.
12. Szekely, P.A. and Myers, B.A., A user interface toolkit based on graphical objects and constraints, *OOPSLA '88*, San Diego California, Nov. 1988, pp. 36-45.
13. Ungar, D. and Smith, R.B., Self: the power of simplicity, *OOPSLA '87*, Orlando Florida, Oct. 1987, pp. 227-242.

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