# Handling Multiple Domain Objects with Model-View-Controller

Michael J. Mahemoff
*Department of C.S.& S.E.*
*The University of Melbourne*
*Parkville, 3052, Australia*
*m.mahemoff@csse.unimelb.edu.au*
*Tel: +61 3 9344-9100*
*Fax: +61 3 9348-1184*

Lorraine J. Johnston
*School of I.T.*
*Swinburne University of Technology*
*Hawthorn, 3122, Australia*
*ljohnston@swin.edu.au*
*Tel: +61 3 9214-8742*
*Fax: +61 3 9214-5501*

***Abstract***

*The Model-View-Controller (MVC) architecture style separates software into models representing core functionality, views which display the models to the user, and controllers which let the user change the models. Although more sophisticated architectures have since been developed, MVC is interesting to explore because its simplicity makes it more acceptable to practitioners and it is beginning to become well-known in industry. However, MVC is rarely studied with regard to systems containing more than one domain model. Several issues are either ambiguous or missing in the literature: the distinction between views and controllers, the way model states are updated in a multiple-model architecture, and the creation of reusable domain-specific components. A program was developed to investigate these issues, and this paper documents the corresponding design decisions. MVC proved helpful in creating a multiple-model system with reusable components, although some weaknesses remain.*

## 1. Introduction

Managing the user-interface has proven to be a difficult task for developers. There is substantial complexity in ensuring that information is presented accurately to users, that users can control the application appropriately, and that views and domain objects are synchronised. To manage the complexity, reference architectures have been developed, such as Model-View-Controller (MVC), Presentation-Abstraction-Control (PAC), and their more sophisticated successors [4]. Visual user-interface builders have sped up GUI generation in recent years, but they do not exempt programmers from considering how to build maintainable and reliable user-interface architectures.

Reference architectures can be viewed as high-level design patterns — reusable design ideas which have been implemented in real applications. A common theme among user-interface styles is the separation of functionality from presentation. User-interface classes typically present domain classes to the user, enabling the user to view and control the domain. One benefit is the possibility of representing the same domain information in different ways. Designers can therefore tailor the user-interface to fit certain tasks and user characteristics.

Our interest in user-interface architecture stems from our efforts in relating software architecture to usability. We are studying patterns for usability and considering their implications for software

design. To this end, we recently chose MVC as a framework for the underlying software, on the basis that it would be a simple but effective way to develop prototypes. However, we were surprised to find that the literature typically discusses overly simple systems, where only one domain object (model) is present. Thus, we developed a prototype containing several domain objects in order to study how the various components can interact with each other. The lessons learned should be relevant to anyone considering the use of MVC.

Although more sophisticated reference architectures exist (e.g. Arch, PAC-AMODEUS [4]), MVC is still relevant to study and refine, for two reasons:

- Despite the proliferation of UI architectures within academia, it appears that they have not enjoyed much application in industry (see [12]). Before fancier architectures are taken up, MVC is a useful stepping stone for developers who have not previously worked with UI architectural patterns. The Java Foundation Class (JFC) library is inspired by MVC, and makes it likely that programmers will increasingly experiment with MVC in the future.
- There have been few attempts to document the creation of multiple-model systems. Furthermore, those applications which do contain more than one model usually deviate significantly from MVC conventions (e.g. [7, 9]).

This paper outlines problems in the current definition of MVC. It then describes a timezone application—"Marco"—which lets users display the time in a current location, and choose and edit locations (Figure 1). The system contains several domain models and has been developed using the basic conventions of MVC. Although not industry-scale, it does contain enough classes to examine the key issues in building multiple-model MVC applications; including abstract classes, there are 6 models, 21 views, and 8 controllers.

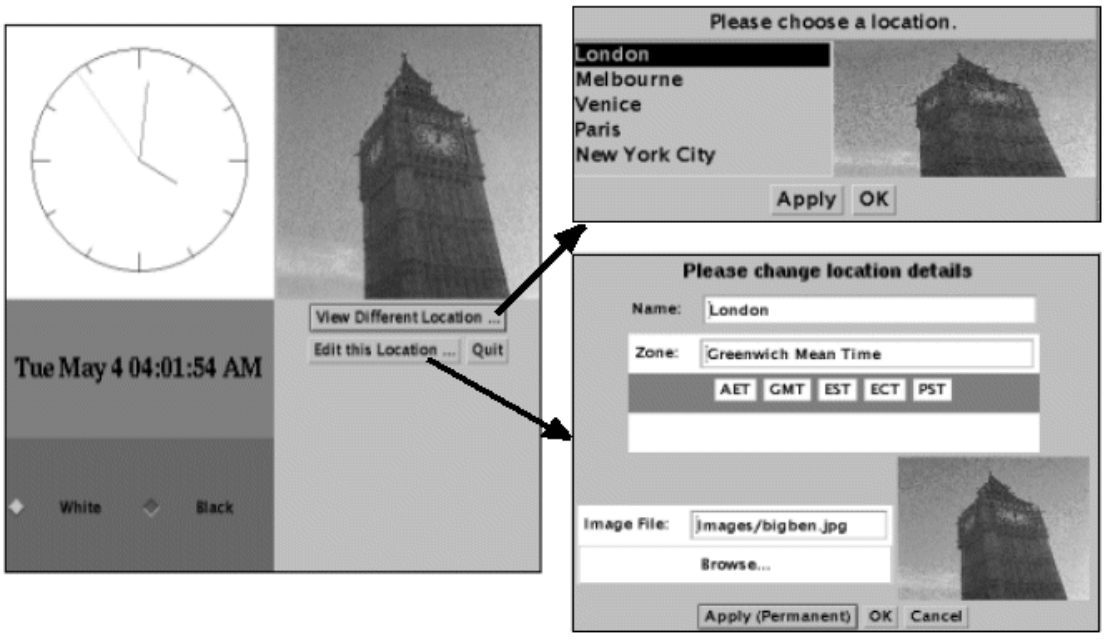


**Figure 1. Major views in Marco: "Focus View" of current location and time, Location Browser, and Location Change form. (See Section 4).**

## 2. MVC overview

The Model-View-Controller paradigm suggests three class categories [11]:

**Models** provide the core functionality of the system (e.g. a car object).

**Views** present models to the user (e.g. a table showing properties of the car). There can be more than one view of the same model.

**Controllers** control how the user interacts with the view objects and manipulate models or views (e.g. an object enabling the user to change a car model using the table view). There is usually a one-to-one relationship between views and controllers.

Models do not directly call views. Instead, each view registers itself with its model, and the model notifies all registered objects whenever it is updated. This lets a developer add or change views without altering the model. It also ensures that views are synchronous because each view reflects the same model state.

In this section, we focus on one particular model in our timezone application: the clock model. This model runs in a separate thread, continually updating itself. Whenever it updates, the digital, analogue, and status views are notified, and they reflect the change by moving the hands or changing the displayed text within their panels (Figure 1 shows digital and analogue clock views).

The digital view has a panel which is created and monitored by the digital view's controller. When the controller detects the user pushing a timezone button, it sends a message to the clock model to set itself to the new timezone. Again, the model will notify its views that it has changed. For illustration, the digital controller also allows the user to alter the colour of the view. This is a different kind of event because it will change only the digital view; the model will not be notified and the analogue view will therefore be unaffected.

The static class diagram is shown in Figure 2(a). View and ClockView encapsulate generic viewing capabilities. They ensure that each concrete view holds a reference to the clock model. They also define several methods which let concrete views initialise themselves, register with the model, create a controller if necessary, and redraw themselves whenever the model updates.

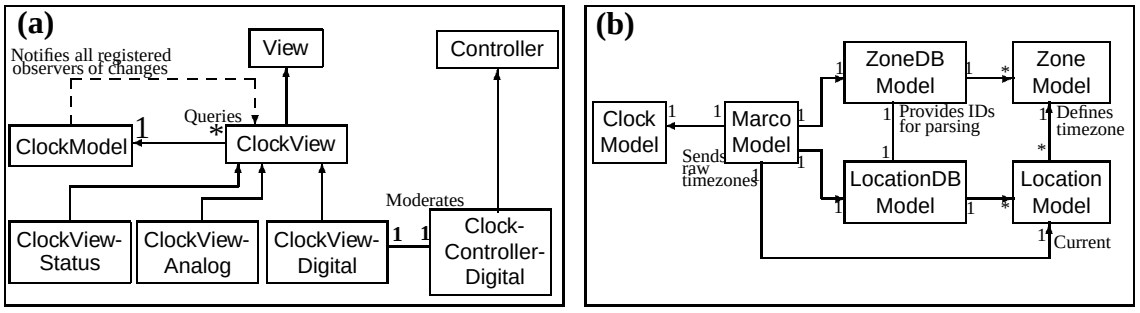


**Figure 2. (a) Static class model of ClockModel and its views and controller. (b) Static class model of all domain models.**

MVC's efficacy is demonstrated by its widespread influence in the design patterns arena. Gamma et al. [6] explain how `Observer`, `Composite`, and `Strategy` relate to Smalltalk MVC. Views are `Observer`s of models. Views are `Composite`s containing their own information as well as subviews. Controllers define a `Strategy` which moderates the behaviour of Views. A pattern specifically about MVC has also been documented[1], but it mainly covers the relationships between model, view, and controller, and only describes a single-model example. In the next section, we

raise questions about MVC in the context of larger or more complex software.

## 3. Aspects of MVC in need of clarification

### 3.1. Role of controllers

The view and controller are supposed to create and maintain the user interface, track user events, and update the model when appropriate. If the designer understands the model, and knows how the user interface should behave, it is quite straightforward to construct a hybrid view-controller class. Indeed, the Document-View architectural pattern—exemplified by the Microsoft Foundation Classes—does exactly this, collapsing MVC's view and controller into a single class. PAC's presentation class does likewise [3].

However, MVC calls for separate view and controller classes in order to provide more flexibility to the user interaction. With a modern user interface toolkit, the distinction loses some of its relevance [2] and a common question about MVC is whether the distinction is really necessary [13]. In particular, there is usually no need for the programmer to send events to the appropriate widget; the external environment ensures that the appropriate widget will receive events. Separating visual properties from event-handling capabilities can also be difficult, because modern widgets encapsulate appearance and behaviour. Nevertheless, there are still some good reasons to separate view and controller. It can be helpful to add new buttons, keyboard shortcuts, and so on, without directly changing the view. The key issue is creating a workable policy which determines how the class types are separated.

### 3.2. Updating models

Consider an application where the user can select several file icons at once. In this case, each icon is a view of type "Icon" on a File model. Another model, perhaps one representing a new directory, will store the list of File models. In MVC, this could be difficult to achieve. The icons present views of File models, but the controller of these views enables the user to change Directory models, not File models. This deviates from a basic rule of MVC, that a view-controller pair is associated with one model. It is not hard to see why this rule is in place. Allowing every controller to interact with any model in the application (or any controller) would create excessive coupling, since there can be many different controllers for each model. The coupling would lead to unmaintainable code, and make it impossible to reuse the components.

MVC's tendency to split control across all three component types has been noted previously [3, 14]. Indeed, a variation called MVC++ has been devised, in which controller classes mediate between views and models, and only controllers can interact with other M-V-C groups [9]; PAC takes a similar approach. The Visualworks Smalltalk framework provides a broadcasting protocol enabling messages to be sent down the visual component structure, which can help reduce direct coupling of views. This framework also provides specialised model classes (e.g. AspectAdaptors) which can help views to access models contained within other models [8].

The present work aimed to investigate the applicability of MVC itself to a multiple-model, rather than rely on a language-dependent framework (although it may be very reasonable to do this in practice) or devise a new reference architecture. Thus, the program described in Section 4 did not vary MVC in this way. Instead, as will be discussed, an event-generation mechanism was used to overcome the scaleability problem.

### 3.3. Reusability of components

Reusable component technology has become a widespread issue in software engineering in recent years [10]. Developers are hungry for autonomous, cohesive, dynamic packages which can be incorporated into target systems without having to devote large amounts of time re-coding or reconfiguring. Reusability was one of the benefits touted for MVC a few years ago, but this referred to buttons, popup-menus, and so on. These are widgets provided by most modern toolkits. Whether or not the MVC framework provides a better way to use these widgets may still be the subject of debate, but clearly modern components should do more than just act as widgets.

To facilitate greater reuse in MVC, domain-specific packages can be created, each consisting of one model class and their associated view and controller classes. The problems with scaleability and control discussed above suggest that classes may not be independent enough to achieve this goal. Models interact with other models. Views add and remove subviews—which are often views of different models. Controllers receive input through one view, and it may often be tempting to permit them to alter other models. Thus, the challenge for MVC component developers is to create autonomous model-view-controller packages.

The JFC library is a step in this direction, with a few semantic models, such as Tree, Table, and List models with corresponding views and controllers. The program we developed does likewise for domain-specific components, such as Clocks and Locations. The program provides an insight into the construction of components in a language-independent manner, by: (a) being based on the more primitive AWT library, rather than JFC, and (b) avoiding the Javabean library, which facilitates component reuse. Although these are useful tools, their absence improved the proof-of-concept argument. Furthermore, components such as Tree work well in isolation, but they still leave open the question of combining components in a reliable and flexible manner. The next section considers how higher-level components—such as databases of models—can build lower-level components.

## 4. Multiple-model MVC application in action

The sample program is called MARCO (Multiple-Agent Real-time Clock Objects), implemented in Java 1.1. The application is a clock which can show the time at various locations in the world. Whereas the single-model application discussed above "hard-coded" the locations, MARCO reads initial locations from a file and the user can change the location database through the interface provided. A fixed database stores the domain of timezones for the locations along with common names (e.g. "Greenwich Mean Time").

There is a model called MarcoModel which represents the overall system state, similar to the ApplicationModel concept in VisualWorks [8]. From this, there are three top-level "MarcoViews", i.e. views of MarcoModel. Each can be run as a separate program. Each MarcoView is composed of views of lower-level models. In fact, it is also possible to run all three overall views at the same time, and the sub-views of each will stay synchronised with each other.

**1. MarcoViewFocus** This view (Figure 1) contains two clock views, an image of the location, and buttons which let the user change the location details or switch to another location.
When the user clicks on "View Different Location", a Location Browser is loaded. The user can browse through locations, see their images, and choose a new location. Whenever a new location is chosen, the main view updates its clocks and image. There is no limit on the number of location browsers which can be open at once.

By clicking on "Edit this Location", a Location Change Form is produced. This lets the user modify the location's name, timezone, and image. When a user changes a location's details, the modifications propagate to all other views. When an image is updated on the change form, for example, the image will also change in the main view and all location browsers in which that location is selected. When a timezone is updated, the two clocks update. If the user chose to edit the current location, then selected a new location with the browser, the change form would still relate to the initial location. Thus, changing details on the form would have no effect on the main view. The user can have any number of browsers and change forms open at the same time. All are kept synchronous.

**2. MarcoViewBackground** This view shows a digital clock and lets the user change timezone (instead of location, to demonstrate flexibility). It is a toplevel window with just the digital clock view and the zone-choosing component which is contained in the Location Change Form.

**3. MarcoViewStatus** This is a more technical view, showing internal status strings for each object and could be used for development purposes.

## 5. Design overview

### 5.1. Models

The MARCO system consists of six domain model classes, whose relationships are shown in Figure 2(b). The models are:

**MarcoModel** An overall application class which exemplifies the `Mediator` pattern [6]. It stores a ClockModel, a ZoneDBModel, and a LocationDBModel and also a reference to the current LocationModel. It fulfils our vision of component reusability by demonstrating how the other components can be brought together to interact with each other. The ClockModel does not know about the other classes; nor do they know about the ClockModel.

**ClockModel** A clock with an associated timezone which continually updates itself. It uses the TimeZone library class to calculate appropriate time.

**ZoneModel** A timezone, common name and Java TimeZone library ID.

**LocationModel** A location with a name, image, and associated ZoneModel.

**ZoneDBModel and LocationDBModel** Classes capable of storing and manipulating groups of ZoneModels and LocationModels (the list is stored as a Vector class).

Component autonomy is ensured by state information being stored only in MarcoModel; the databases, for instance, do not store the current location or zone. All models extend Java's Observable class, which provides the behaviour required to maintain a list of observers and notify them upon changes to the model (the class could easily be created for another programming language).

The initialisation of models is shown in Figure 3. MarcoModel requests both databases to parse the files containing zone and location information. It becomes an observer of these databases, explained in Section 6 (Problem 5). It then sets the current location to the first item in the location database, and starts the clock, with this location's timezone. The clock maintains independence from other packages by accepting only standard Java TimeZone objects, rather than ZoneModels.

### 5.2. Views

There is an abstract View class which: registers itself with the model and stores a reference to the

model; defines an update method which (a) checks that the correct object has notified it and (b) calls the draw method, for objects to inspect the model and redraw themselves; provides an empty method to create the controller, which subclasses can override if they require a controller; contains convenience methods setVerbosity() and getString() which enable subclasses to easily obtain strings which portray the model's states at varying levels of detail.

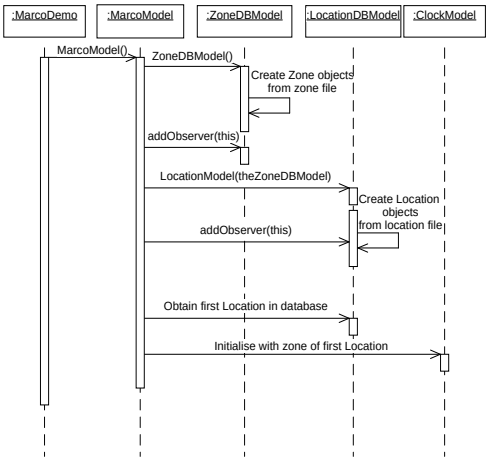


**Figure 3. Initialisation of major models.**

The structure of Views follows the Composite pattern [6]. When a View is created, it constructs its own widgets and also adds sub-views. The sub-views look after themselves. Thus, the clock views inside the main Focus View both observe the Clock model and update themselves accordingly. The Focus View itself does not have to observe the Clock.

View does not contain a reference to the model, because the model type is different for each subclass of View. Instead, there is an abstract view class for each model, e.g. ClockView. This class stores a reference to the model, provides a constructor which sets the model, and can define the convenience string-handling methods. Model-specific views can also perform miscellaneous functions specific to the model type. ClockView, for instance, overrides the update method to impose a minimum number of seconds between each redraw.

The concrete views have one or more widgets which can be obtained via public methods. For instance, the timezone selection view provides one panel showing buttons for each zone, and another status bar label which shows the zone's common name whenever the mouse hovers above a button. A developer using this class has the freedom to show only the buttons, or the buttons with the status bar, and the two widgets can be placed in separate screen locations.

### 5.3. Controllers

It was decided to explicitly distinguish between controllers which control through user interaction with specific views and those which are generic and control a model directly. The one class in the second category is the Location Change Form. It is initialised with the model as a parameter, and disposes itself when the user hits the "OK" or "Cancel" buttons. Its independence makes it easy to add to existing views. The controller for the location status view enables the user to produce the form by double-clicking on the status text. This will generate the location form, and when the user changes location details, the model will be altered, and the original status view will update itself.

In this implementation, controllers which relate to views have several responsibilities. First, they implement Java's event-handler interfaces to listen for appropriate events, e.g. the zone selection controller listens for occasions when the user rolls the mouse over a zone button and tells the view's status bar to update. Second, views delegate construction and maintenance of button panels to controllers. Third, controllers can broadcast semantic events (e.g. zone changed), to objects who have informed the controller that they are interested in these events.

Following the initialisation of models (Figure 3, MarcoViewFocus is constructed with its model.

It stores the model and begins to observe it. This is standard procedure for all views, embodied in the View superclass constructor. Another procedure inherited from View, `initialise`, begins by calling `makeController`. In the case of MarcoViewFocus, this is left blank because no constructor is required. `initialise` then calls `constructInitialView` to create a digital clock view, an analog clock view, and a location image. The digital clock view creates its controller. It then constructs its own initial view, consisting of a label to store the time on as well as the control panel obtained from its controller. Finally, it calls `draw`, which shows the current time. `MarcoViewFocus.constructInitialView` creates the analog view and the location image in this same way. It then updates itself by calling its own `draw` method. Since the subviews update themselves, this method is very small; all it does is update the window title.

Incidentally, it might seem odd to have an `initialise` method — why not use the constructor? The answer is that `initialise` creates a controller. Since the controller will immediately store a reference to the view, it cannot be called from the view's constructor because the view reference will not yet be valid [1].

## 6. Lessons learned

• **Problem 1:** The view and controller can be separated in different ways—see Section 3.1. Which is appropriate here?

**Solution: Controllers implement the methods which handle events on the view's widgets.** Some coupling between view and controller is still necessary, because the views must configure the widgets with the appropriate listener. For instance, the Location Status View must inform its display panel that the location status controller will listen to button-clicking events. However, the level of coupling is non-critical, since this is the only situation in which the controller is referred to. Furthermore, most event-listening occurs on button-panels, and these are created by the controller.

This approach splits presentation across two classes, which could lead to confusion. To minimise complexity, views provide some methods for the controller to perform output, e.g. the zone database view provides methods to set the status bar according to a string or a ZoneModel parameter.

Java's event-handling mechanism provided a convenient way to use controllers—to ensure that controllers implement the appropriate event-handlers. For efficiency purposes the Java 1.1 toolkit requires specific event listeners, e.g. a mouse listener, a text component listener. This forces some coupling between view and controller, because it means the view must add the appropriate listener to each widget.

The Location Browser is a controller with no views, but could also have been implemented as a view with text fields, connected to a controller which knows how to handle them. This would have been rather contrived, however, and led to a very tight coupling between view and control. Clearly, the form's purpose is to control the model. Therefore it was reasonable to abandon the typical MVC pattern of one-to-one view-controller relationships.

• **Problem 2:** Each model-view-controller package should rely on as few other packages as possible.

**Solution: The relationship among views and controllers is aligned with their models.** That is, if a model $M_1$ depends uni-directionally on $M_2$, then views and controllers of $M_1$ can use the $M_2$ package, but not vice-versa. This means that a package of a model and its views and controllers has its degree of independence determined by the independence of the model alone.

To promote independence of models, class associations should be uni-directional where possible. In Figure 2(b), ZoneModel and its associated views and controllers ("the Zone package") is com-

pletely independent. The database of Zones, ZoneDBModel, depends only on ZoneModel, so the pair of classes could be reused, for example, in travel-related software. In fact, there are several combinations of packages which could easily be combined and reused: {Zone}, {ZoneDB, Zone}, {LocationDB, Location, ZoneDB, Zone}, {Clock}.

**Alternative Solutions:** One alternative design would eliminate the database classes and use MarcoModel to directly store lists of LocationModels and ZoneModel. This would force MarcoView subclasses to allow users to manipulate the lists. However, this is functionality which would be desirable in any application using Location and Zone, and should consequently be separated from the application, hence the need for database classes.

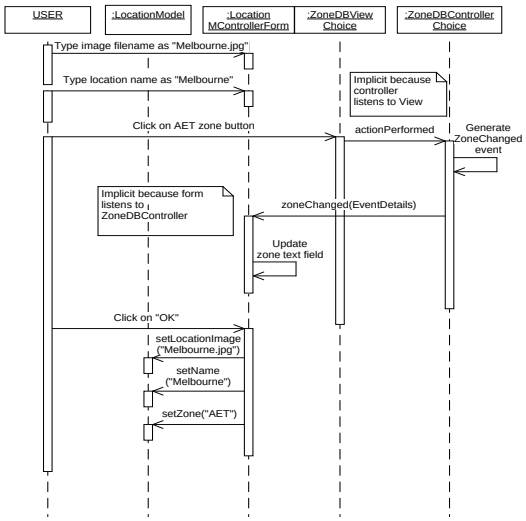


**Figure 4. User edits Location.**

- **Problem 3:** How can views and controllers present information relating to one model and simultaneously enable a user to change another model? For example, the Location Change Form controls a LocationModel, and the user selects the zone from an inset Zone Database View. The dilemma is that the database view presents one model, but user events on the database view affect another model. If the zone database controller directly changes the location model, or instructs a location controller to do so, a bi-directional dependency is introduced.

  **Solution: Objects can generate semantic events to signal important changes, and this avoids them having to notify specific classes**.

The handling of the above example is shown in Figure 4. When it was initialised, the Location Change Form registered itself to receive semantic events from the inset zone database view. When the user clicks on a button on the zone database, the zone database controller catches the event. It generates a `ZoneDBEvent`, a newly-defined event type which stores the view that was chosen. A callback method in the location form receives this event, and updates the zone text field beside the database view (if desired, the form could immediately change the underlying model).

Controller classes maintain a list of classes which have registered for the event, and when the user implicitly requests the event via the user-interface (e.g. clicking on a zone button), a method is called to loop through each registered class and then call its callback method (e.g. `zoneChanged(ZoneDBEvent theZoneDBEvent)`).

Semantic event generation is one important way to scale up MVC applications. Events can be refined in more abstract terms as they move upwards towards higher-level classes. This helps to reduce complexity. Reusability is also facilitated; in this example, the zone database view does need to know the specific classes which use it.

Semantic events are used in various frameworks, such as ActiveX, Javabeans, and the Business Object Component Architecture (BOCA) [5]. Often, these components effectively represent a view,

a model, or both, and event types could be a user-interaction or an internal change. Also, widgets typically use this notion to some extent; as well as allowing callbacks for primitive user events such as button-clicks, many widgets can notify the program of such higher-level events as an item being added to a list. In the context of multiple-model MVC, semantic events are beneficial because they reduce direct references between objects.

**Alternative Solutions:** It is not feasible to use the existing Java event and listener classes, because the events do not carry the semantic information required, e.g. the ZoneModel which the user requested. This is why new classes were required, as well as controllers to keep lists of listeners. However, the code to do this is minimal.

It would also be possible to make controllers Observable, and let interested parties implement Observer to watch for changes. This is really another version of registering for events, but would be an abuse of the Observable class, since user events do not really constitute changes to the controller. Semantic events are more flexible, as there can be more than one type of event, and they are conceptually a more logical way to handle semantic-level user requests.

• **Problem 4:** Some models and views have their state defined by one or more models. In the case of the location browser view which enables users to switch between LocationModels, the present state can be defined by the LocationModel which the user has most recently selected from the list (a property of the browser view). As the user clicks on list items, the corresponding image view must change to reflect the current model. A similar situation arises for MarcoViewFocus, whose image view must change when MarcoModel's current LocationModel changes.

**Solution: When a container view includes a view of a model class, and the particular model object might change, a view of each possible model object is created upon initialisation and whenever the possible set of models changes. Whenever the model object changes, the view is cached in memory and simply needs to be displayed.**

For example, when the LocationModel database changes, a list of views is created, one for each model in the database. Whenever the user switches models, the pre-existing view is replaced by the view of the selected LocationModel.

This approach has the disadvantage that view objects are updated whenever their models change, even if they are not visible. A more sophisticated technique—perhaps based on Gamma et al.'s Proxy pattern [6]—could prevent this by ensuring views only update themselves when they are visible.

**Alternative Solutions:** Perhaps a more obvious solution is a "lazy" approach: simply create a view of each model whenever it is required. This would certainly be appropriate if the user rarely changed the current model, or there were many models. This approach uses less memory than the previous method, but will be slower. To improve speed, already-created views could be stored. However, this would be similar to the solution described above.

Alternatively, it would be possible to create a new state model on initialisation, and create a single view to observe it. Nothing more would need be done to the view, because it would update as the model changes. However, the model would be a copy of the real model in the database, rather than a reference, and this means that when a user tried to change the current model, they would be changing a copy of it; the real model in the database would not be updated. This complication could probably be worked around, but would pose a large risk of synchronisation errors.

• **Problem 5:** Some models reference other models, and need to know when they have changed. For example, the overall application model, MarcoModel, needs to know when the current Location changes zone, so it can inform the clock. However, to avoid a cyclic dependency, the changed models should not directly communicate with the model which references them.

**Solution: When a model stores a reference to another model, it registers itself as an observer of that model.**

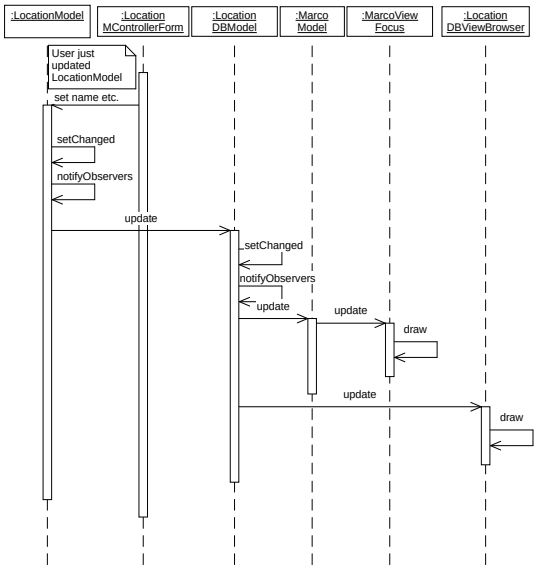


**Figure 5. Updating views to reflect Location details change (follows Figure 4).**

Figure 5 shows what happens when the user changes details on a Location Change Form. It follows immediately from Figure 4. The Location database has registered as an observer of each of its Locations, and therefore knows that the LocationModel in question has been updated. MarcoModel, in turn, is observing the LocationModel, and also updates. As with the standard MVC definition, MarcoViewFocus notices the change to its model and also updates (it redraws the window title). All other views of the location database will also update, such as any Location Browsers which are open.

This approach is helpful to developers of large systems where high-level models depend on lower-level models, and the changes must be propagated throughout. It creates some additional overhead because higher-level views need to update, but it would be possible to create views which only update if the features they are displaying have changed.

## 7. Discussion

The multiple-model MVC program was developed quite smoothly using some of the concepts described in this paper. Initially, it was anticipated the program would act an illustration of the overly-complex nature of MVC. However, the result was quite reasonable, with a good degree of independence among packages and a manageable policy of combining packages together. The design's generality is evidenced by the fact that it has been implemented in a fairly standard language (i.e. Java), and relied on the kinds of libraries which are available for most modern languages.

Some inherent weaknesses with MVC do remain. It is difficult to separate views and controllers in a totally clean manner, because modern toolkits usually mix output and input. Even though there will always be some coupling, the separation is likely to be worthwhile in many cases because there are always going to be different ways of manipulating the same view.

Another weakness is performance. The registration mechanism has been extended here to allow models to observe other models. Furthermore, the semantic event-generation concept has also been introduced to cope with multiple packages. Both of these patterns will cause some performance degradation. The maintenance of view lists does likewise, although optimisations—like an image view cache—are possible.

The program is larger than many previously-documented MVC applications, and therefore provides

some insight about how to integrate different model-view-controller packages and design for reuse. However, with five domain classes, it is hardly a heavyweight system. This leads to the question about whether the concepts introduced would scale further. As long as the software could be broken into cohesive subsystems, it seems feasible to suggest that large-scale systems are able to be developed with the ideas described here. In particular, this would be possible by defining clear policies for splitting view and controller, applying semantic events, ensuring that models store other models as references rather than as copies, and having models register to observe changes on other models.

Future work should aim at establishing the validity of these ideas by applying them to new applications. It would also be valuable to enhance the framework by applying more flexibility. For instance, a subview might adapt to the colour and layout scheme of the view it is inserted into. Ultimately, the framework could be extended further to produce a code-generation tool. This would be very helpful, because a large amount of code in the program is of the trivial-but-necessary variety, and could easily be automated. Finally, we deliberately chose to avoid relying on a particular language or framework to improve generalisability. Nevertheless, it would be useful to investigate whether our findings could assist with development under an existing framework such as JFC.

## 8. Acknowledgements

The authors thank Andrew Hussey (SVRC, University of Queensland) for reviewing an early version, and the anonymous reviewers for their helpful comments.

## References

[1] F. Bushcmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns*. John Wiley & Sons, Chichester, 1995.

[2] D. Collins. *Designing Object-Oriented User Interfaces*. Benjamin/Cummings, Redwood City, CA, 1995.

[3] J. Coutaz. The construction of user interfaces and the object paradigm. In J. Bézivin, J. M. Hullot, P. Cointe, and H. Lieberman, editors, *European Conference on Object-Oriented Programming (Lecture Notes in Computer Science 276)*, pages 121–130. Springer-Verlag, Berlin, 1987.

[4] J. Coutaz. Architectural design for user interfaces. In J. J. Marciniak, editor, *Encyclopaedia of Software Engineering*, pages 38–49. John Wiley & Sons, New York, 1994.

[5] T. Digre. Business object component architecture. *IEEE Software*, 15(5):60–69, September 1998.

[6] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[7] E. Gülch and H. Müller. Object-oriented software design in semiautomatic building extraction. In D. M. McKeown, J. McGlone, and O. Jamet, editors, *Integrating Photogrammetric Techniques with Scene Analysis and Machine Vision III*, pages 37–48. SPIE, Bellingham, WA, 1997.

[8] T. Howard. *The Smalltalk developer's guide to VisualWorks*. SIGS, New York, 1995.

[9] A. Jaaksi. Implementing interactive applications in C++. *Software–Practice and Experience*, 25(3):271–289, March 1995.

[10] W. Kozaczynski and G. Booch. Component-based software engineering. *IEEE Software*, 15(5):34–36, September 1998.

[11] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *JOOP*, 1(3):26–49, August/September 1988.

[12] S. Pemberton. Teenagers, sex education and Microsoft. *SIGCHI Bulletin*, 30(2):160, April 1998.

[13] T. Reenskaug. *Working With Objects: The OOram Software Engineering Method*. Manning, Greenwich, CT, 1996.

[14] Y. Shan. Mode: A uims for Smalltalk. In N. Meyrowitz, editor, *ECOOP/OOPSLA '90 Proceedings*, pages 258–268. ACM Press, New York, 1990.