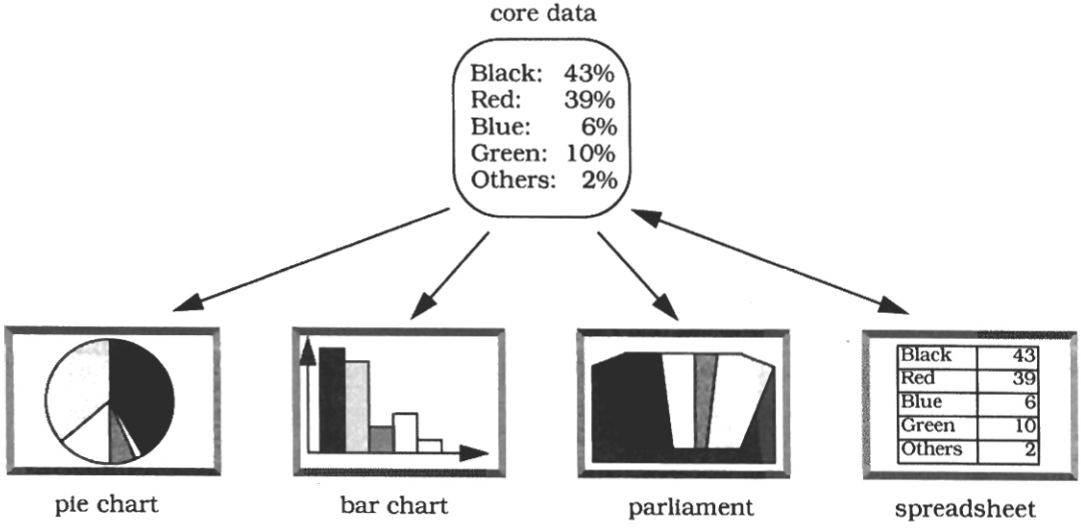
Model-View-Controller

The *Model-View-Controller* architectural pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

Example Consider a simple information system for political elections with proportional representation. This offers a spreadsheet for entering data and several kinds of tables and charts for presenting the current results. Users can interact with the system via a graphical interface. All information displays must reflect changes to the voting data immediately.



It should be possible to integrate new ways of data presentation, such as the assignment of parliamentary seats to political parties, without major impact to the system. The system should also be portable to platforms with different 'look and feel' standards, such as workstations running Motif or PCs running Microsoft Windows 95.

Context Interactive applications with a flexible human-computer interface.

Problem User interfaces are especially prone to change requests. When you extend the functionality of an application, you must modify menus to access these new functions. A customer may call for a specific user interface adaptation, or a system may need to be ported to another platform with a different 'look and feel' standard. Even upgrading to a new release of your windowing system can imply code changes. The user interface platform of long-lived systems thus represents a moving target.

Different users place conflicting requirements on the user interface. A typist enters information into forms via the keyboard. A manager wants to use the same system mainly by clicking icons and buttons. Consequently, support for several user interface paradigms should be easily incorporated.

Building a system with the required flexibility is expensive and error-prone if the user interface is tightly interwoven with the functional core. This can result in the need to develop and maintain several substantially different software systems, one for each user interface implementation. Ensuing changes spread over many modules. The following *forces* influence the solution:

- The same information is presented differently in different windows, for example, in a bar or pie chart.
- The display and behavior of the application must reflect data manipulations immediately.
- Changes to the user interface should be easy, and even possible at run-time.
- Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

Solution Model-View-Controller (MVC) was first introduced in the Smalltalk-80 programming environment [KP88]. MVC divides an interactive application into the three areas: *processing*, *output*, and *input*.

The *model* component encapsulates core data and functionality. The model is independent of specific output representations or input behavior.

View components display information to the user. A view obtains the data from the model. There can be multiple views of the model.

Each view has an associated *controller* component. Controllers receive input, usually as events that encode mouse movement, activation of mouse buttons, or keyboard input. Events are translated to service requests for the model or the view. The user interacts with the system solely through controllers.

The separation of the model from view and controller components allows multiple views of the same model. If the user changes the model via the controller of one view, all other views dependent on this data should reflect the changes. The model therefore notifies all views whenever its data changes. The views in turn retrieve new data from the model and update the displayed information. This change-propagation mechanism is described in the Publisher-Subscriber pattern (339).

Structure The *model* component contains the functional core of the application. It encapsulates the appropriate data, and exports procedures that perform application-specific processing. Controllers call these procedures on behalf of the user. The model also provides functions to access its data that are used by view components to acquire the data to be displayed.

The change-propagation mechanism maintains a registry of the dependent components within the model. All views and also selected controllers register their need to be informed about changes. Changes to the state of the model trigger the change-propagation mechanism. The change-propagation mechanism is the only link between the model and the views and controllers.

Class Model	***Collaborators*** • View • Controller
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	

View components present information to the user. Different views present the information of the model in different ways. Each view defines an update procedure that is activated by the change-propagation mechanism. When the update procedure is called, a view retrieves the current data values to be displayed from the model, and puts them on the screen.

During initialization all views are associated with the model, and register with the change-propagation mechanism. Each view creates a suitable controller. There is a one-to-one relationship between views and controllers. Views often offer functionality that allows controllers to manipulate the display. This is useful for user-triggered operations that do not affect the model, such as scrolling.

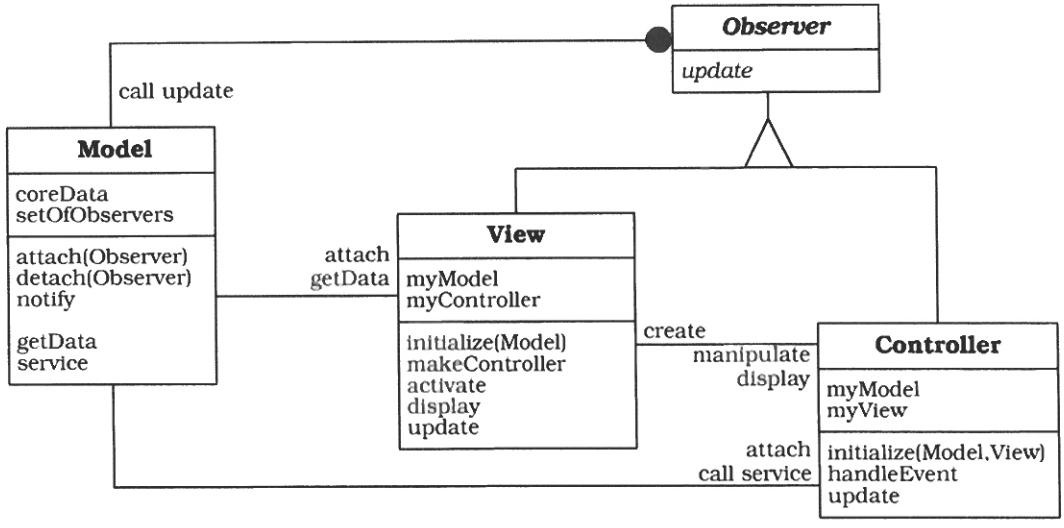
The *controller* components accept user input as events. How these events are delivered to a controller depends on the user interface platform. For simplicity, let us assume that each controller implements an event-handling procedure that is called for each relevant event. Events are translated into requests for the model or the associated view.

If the behavior of a controller depends on the state of the model, the controller registers itself with the change-propagation mechanism and implements an update procedure. For example, this is necessary when a change to the model enables or disables a menu entry.

Class View	***Collaborators***
Responsibility	• Controller • Model
• Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	

Class Controller	***Collaborators***
Responsibility	• View • Model
• Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	

An object-oriented implementation of MVC would define a separate class for each component. In a C++ implementation, view and controller classes share a common parent that defines the update interface. This is shown in the following diagram. In Smalltalk, the class Object defines methods for both sides of the change-propagation mechanism. A separate class Observer is not needed.



➥ In our example system the model holds the cumulative votes for each political party and allows views to retrieve vote numbers. It further exports data manipulation procedures to the controllers.

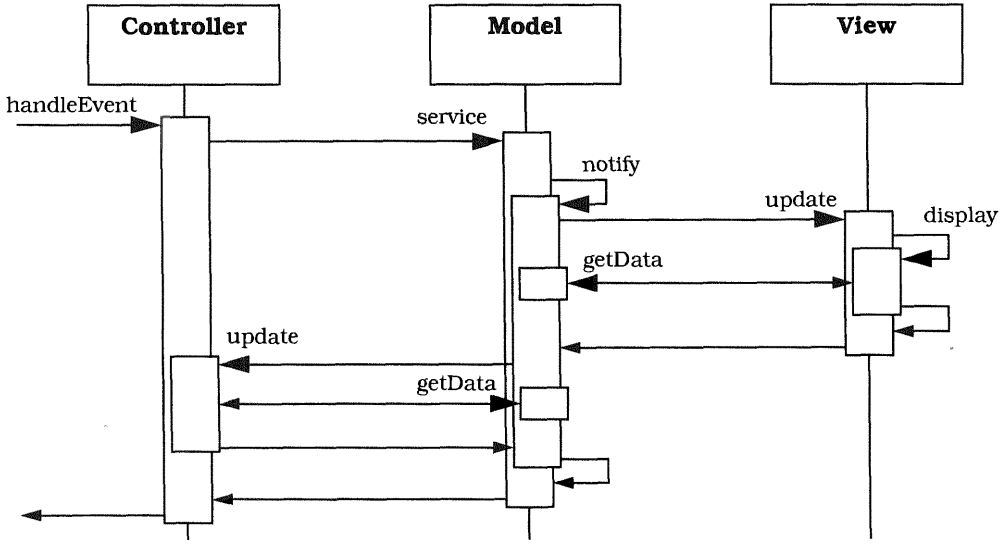
We define several views: a bar chart, a pie chart and a table. The chart views use controllers that do not affect the model, whereas the table view connects to a controller used for data entry. ❑

You can also use the MVC pattern to build a framework for interactive applications, as within the Smalltalk-80 environment [KP88]. Such a framework offers prefabricated view and controller subclasses for frequently-used user interface elements such as menus, buttons, or lists. To instantiate the framework for an application, you can combine existing user interface elements hierarchically using the Composite pattern [GHJV95].

Dynamics The following scenarios depict the dynamic behavior of MVC. For simplicity only one view-controller pair is shown in the diagrams.

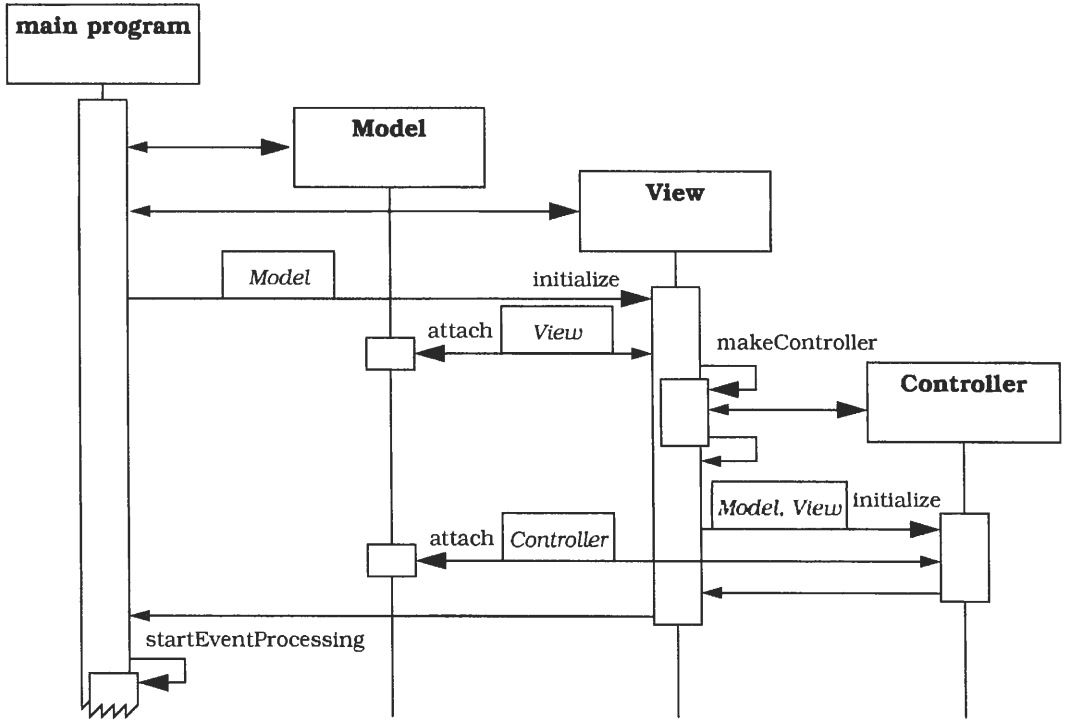
Scenario I shows how user input that results in changes to the model triggers the change-propagation mechanism:

- The controller accepts user input in its event-handling procedure, interprets the event, and activates a service procedure of the model.
- The model performs the requested service. This results in a change to its internal data.
- The model notifies all views and controllers registered with the change-propagation mechanism of the change by calling their update procedures.
- Each view requests the changed data from the model and re-displays itself on the screen.
- Each registered controller retrieves data from the model to enable or disable certain user functions. For example, enabling the menu entry for saving data can be a consequence of modifications to the data of the model.
- The original controller regains control and returns from its event-handling procedure.



Scenario II shows how the MVC triad is initialized. This code is usually located outside of the model, views and controllers, for example in a main program. The view and controller initialization occurs similarly for each view opened for the model. The following steps occur:

- The model instance is created, which then initializes its internal data structures.
- A view object is created. This takes a reference to the model as a parameter for its initialization.
- The view subscribes to the change-propagation mechanism of the model by calling the attach procedure.
- The view continues initialization by creating its controller. It passes references both to the model and to itself to the controller's initialization procedure.
- The controller also subscribes to the change-propagation mechanism by calling the attach procedure.
- After initialization, the application begins to process events.



Implementation Steps 1 through 6 below are fundamental to writing an MVC-based application. Steps 7 through 10 describe additional topics that result in higher degrees of freedom, and lend themselves to highly flexible applications or application frameworks.

1 *Separate human-computer interaction from core functionality.* Analyze the application domain and separate the core functionality from the desired input and output behavior. Design the model component of your application to encapsulate the data and functionality needed for the core. Provide functions for accessing the data to be displayed. Decide which parts of the model's functionality are to be exposed to the user via the controller, and add a corresponding interface to the model.

➥ The model in our example stores the names of the political parties and the corresponding votes in two lists of equal length[18]. Access to the lists is provided by two methods, each of which creates an iterator. The model also provides methods to change the voting data.

```
class Model{
     List<long>   votes;
     List<String> parties;
public:
     Model(List<String> partyNames);

// access interface for modification by controller
     void clearVotes(); // set voting values to 0
     void changeVote(String party, long vote);

// factory functions for view access to data
     Iterator<long>   makeVoteIterator(){
          return Iterator<long>(votes);
     }
     Iterator<String> makePartyIterator(){
          return Iterator<String>(parties);
     }
// ... to be continued
}
```

❑

2 *Implement the change-propagation mechanism.* Follow the Publisher-Subscriber design pattern (339) for this, and assign the role of the publisher to the model. Extend the model with a registry that holds references to observing objects. Provide procedures to allow views and

18. An associative array with party names as keys and votes as the information would be a more realistic implementation but would bloat the example code.

controllers to subscribe and unsubscribe to the change-propagation mechanism. The model's notify procedure calls the update procedure of all observing objects. All procedures of the model that change the model's state call the notify procedure after a change is performed.

➥ Proper C++ usage suggests that one should define an abstract class `Observer` to hold the update interface. Both views and controllers inherit from `Observer`. The `Model` class from step 1 is extended to hold a set of references to current observers, and two methods, `attach()` and `detach()`, to allow observing objects to subscribe and unsubscribe. The method `notify()` will be called by methods that modify the state of the model.

```
class Observer{ // common ancestor for view and controller
public:
     virtual void update() { }
// default is no-op
};

class Model{
// ... continued
public:
     void attach(Observer *s) { registry.add(s); }
     void detach(Observer *s) { registry.remove(s); }
protected:
     virtual void notify();
private:
     Set<Observer*> registry;
};
```

Our implementation of the method `notify()` iterates over all Observer objects in the registry and calls their update method. We do not provide a separate function to create an iterator for the registry, because it is only used internally.

```
void Model::notify(){
     // call update for all observers
     Iterator<Observer*> iter(registry);
     while (iter.next()){
          iter.curr()->update();
     }
}
```

The methods `changeVote()` and `clearVotes()` call `notify()` after the voting data is changed. ❑

3 *Design and implement the views.* Design the appearance of each view. Specify and implement a draw procedure to display the view on the screen. This procedure acquires the data to be displayed from the model. The rest of the draw procedure depends mainly on the user interface platform. It would call, for example, procedures for drawing lines or rendering text.

Implement the update procedure to reflect changes to the model. The easiest approach is to simply call the draw procedure. The draw procedure goes ahead and fetches data needed for the view. For a complex view requiring frequent updates, such a straightforward implementation of update can be inefficient. Several optimization strategies exist in this situation. One is to supply additional parameters to the update procedure. The view can then decide if a re-draw is needed. Another solution is to schedule, but not perform, the re-draw of the view when it is likely that further events also require it. The view can then be redrawn when no more events are pending.

In addition to the update and draw procedures, each view needs an initialization procedure. The initialization procedure subscribes to the change-propagation mechanism of the model and sets up the relationship to the controller, as shown in step 5. After the controller is initialized, the view displays itself on the screen. The platform or the controller may require additional view capabilities, such as a procedure to resize a view window.

➥ For all the views used by the election system we define a common base class `View`. The relationships to model and controller are represented by two member variables with corresponding access methods. The constructor of `View` establishes the relationship to the model by subscribing to the change-propagation mechanism. The destructor removes it again by unsubscribing. `View` also provides a simple non-optimized `update()` implementation.

```
class View : public Observer {
public:
    View(Model *m) : myModel(m), myController(0)
        { myModel->attach(this); }
    virtual ~View() { myModel->detach(this); }
    virtual void update() { this->draw(); }
    // abstract interface to be redefined:
    virtual void initialize() ;// see below
    virtual void draw() ;      // (re-)display view
// ... to be continued below
```

```
    Model *getModel() { return myModel; }
    Controller *getController() { return myController; }
protected:
    Model        *myModel;
    Controller   *myController; // set by initialize
};

class BarChartView : public View {
public:
    BarChartView(Model *m) : View(m) { }
    virtual void draw();
};

void BarChartView::draw(){
    Iterator<String> ip = myModel->makePartyIterator();
    Iterator<long> iv = myModel->makeVoteIterator();
    List<long> dl; //for scaling values to fill screen
    long     max = 1;// maximum for adjustment

    // calculate maximum vote count
    while (iv.next()) {
        if (iv.curr() > max ) max = iv.curr();
    }
    iv.reset();
    // now calculate screen coordinates for bars
    while (iv.next()) {
        dl.append((MAXBARSIZE * iv.curr())/max);
    }

    // reuse iterator object for new collection:
    iv = dl; // assignment rebinds iterator to new list
    iv.reset();

    while (ip.next() && iv.next()) {
        // draw text: cout << ip.curr() << " : " ;
        // draw bar: ... drawbox(BARWIDTH, iv.curr());...
    }
}
```

The class definition of `BarChartView` demonstrates a specific view of our system. It redefines `draw()` to show the voting data as a bar chart. ❑

4 *Design and implement the controllers.* For each view of the application, specify the behavior of the system in response to user actions. We assume that the underlying platform delivers every action of a user as an event. A controller receives and interprets these events using a dedicated procedure. For a non-trivial controller, this interpretation depends on the state of the model.

The initialization of a controller binds it to its model and view and enables event processing. How this is achieved depends on the user-interface platform. For example, the controller may register its event-handling procedure with the window system as a callback.

➥ Most views in our example do not require any specific event processing—they are only used for display. We therefore define a base class `Controller` with an empty `handleEvent()` method. The constructor attaches the controller to its model and the destructor detaches it again.

```
class Controller : public Observer {
public:
    virtual void handleEvent(Event *) { }
        // default = no op

    Controller( View *v) : myView(v) {
        myModel = myView->getModel();
        myModel->attach(this);
    }

    virtual ~Controller() { myModel->detach(this); }
    virtual void update() { } // default = no op
protected:
    Model    *myModel;
    View     *myView;
};
```

We omit a separate controller initialization method, because the relationship to the view and the model is already set up by its constructor. ❑

Calling the functional core closely links a controller with the model, since the controller becomes dependent on the application-specific model interface. If you plan to modify functionality, or if you want to provide reusable controllers and therefore would like the controller to be independent of a specific interface, apply the Command Processor (277) design pattern. The model takes the role of the supplier of the Command Processor pattern. The command classes and the command processor component are additional components between controller and model. The MVC controller has the role of controller in Command Processor.

5 *Design and implement the view-controller relationship.* A view typically creates its associated controller during its initialization. When you build a class hierarchy of views and controllers, apply the Factory Method design pattern [GHJV95] and define a method `makeController()` in the view classes. Each view that requires a controller that differs from its superclass redefines the factory method.

➥ In our C++ example the `View` base class implements a method `initialize()` that in turn calls the factory method `makeController()`. We cannot put the call to `makeController()` into the constructor of the View class, because then a subclass' redefined `makeController()` would not be called as desired. The only `View` subclass that requires a specific controller is `TableView`. We redefine `makeController()` to return a `TableController` to accept data from the user.

```
class View : public Observer {
// ... continued
public:
//C++ deficit: use initialize to call right factory method
     virtual void initialize()
          { myController = makeController();}
     virtual Controller *makeController()
          { return new Controller(this); }
};

class TableController : public Controller {
public:
     TableController(TableView *tv) : Controller(tv) {}
     virtual void handleEvent(Event *e) {
     // ... interpret event e,
     //     for instance, update votes of a party
          if(vote && party){ // entry complete:
               myModel->changeVote(party,vote);
          }
     }
};
class TableView : public View {
public:
     TableView(Model *m) : View(m) { }
     virtual void draw();
     virtual Controller *makeController()
          { return new TableController(this); }
};
```

❑

6 *Implement the set-up of MVC.* The set-up code first initializes the model, then creates and initializes the views. After initialization, event processing is started, typically in a loop, or with a procedure that includes a loop, such as `XtMainLoop()` from the X Toolkit. Because the model should remain independent of specific views and controllers, this set-up code should be placed externally, for example, in a main program.

➥ In our simple example the main function initializes the model and several views. The event processing delivers events to the controller of the table view, allowing the entry and change of voting data.

```
main() {
    // initialize model
    List<String> parties;    parties.append("black");
    parties.append("blue "); parties.append("red  ");
    parties.append("green"); parties.append("oth. ");
    Model m(parties);

    // initialize views
    TableView *v1 = new TableView(&m);
    v1->initialize();
    BarChartView *v2 = new BarChartView(&m);
    v2->initialize();
    // now start event processing ...
```

❑

7 *Dynamic view creation.* If the application allows dynamic opening and closing of views, it is a good idea to provide a component for managing open views. This component, for example, can also be responsible for terminating the application after the last view is closed. Apply the View Handler (291) design pattern to implement this view management component.

8 *'Pluggable' controllers.* The separation of control aspects from views supports the combination of different controllers with a view. This flexibility can be used to implement different modes of operation, such as casual user versus expert, or to construct read-only views using a controller that ignores any input. Another use of this separation is the integration of new input and output devices with an application. For example, a controller for an eye-tracking device for disabled people can exploit the functionality of the existing model and views, and is easily incorporated into the system.

➥ In our example only the class `TableView` supports several controllers. The default controller `TableController` allows the user

to enter voting data. For display-only purposes, `TableView` can be configured with a controller that ignores all user input. The code below shows how a controller is substituted for another controller. Note that `setController` returns the previously-used controller object. Here the controller object is no longer used and so it is deleted immediately.

```
class View : public Observer{
// ... continued
public:
     virtual Controller *setController(Controller *ctlr);
};

main()
// ...
     // exchange controller
     delete v1->setController(
          new Controller(v1)); // this one is read only
// ...
     // open another read-only table view;
     TableView *v3 = new TableView(&m);
     v3->initialize();
     delete v3->setController(
          new Controller(v3)); // make v3 read-only
     // continue event processing
// ...
}
```

❑

9 *Infrastructure for hierarchical views and controllers.* A framework based on MVC implements reusable view and controller classes. This is commonly done for user interface elements that are applied frequently, such as buttons, menus, or text editors. The user interface of an application is then constructed largely by combining predefined view objects. Apply the Composite pattern [GHJV95] to create hierarchically composed views. If multiple views are active simultaneously, several controllers may be interested in events at the same time. For example, a button inside a dialog box reacts to a mouse click, but not to the letter 'a' typed on the keyboard. If the parent dialog view also contains a text field, the 'a' is sent to the controller of the text view. Events are distributed to event-handling routines of all active controllers in some defined sequence. Use the Chain of Responsibility pattern [GHJV95] to manage this delegation of events. A controller will pass an unprocessed event to the controller of the parent view or to the controller of a sibling view if the chain of responsibility is set up properly.

10 *Further decoupling from system dependencies.* Building a framework with an elaborate collection of view and controller classes is expensive. You may want to make these classes platform independent. This is done in some Smalltalk systems. You can provide the system with another level of indirection between it and the underlying platform by applying the Bridge pattern [GHJV95]. Views use a class named *display* as an abstraction for windows and controllers use a *sensor* class.

The abstract class *display* defines methods for creating a window, drawing lines and text, changing the look of the mouse cursor and so on. The *sensor* abstraction defines platform-independent events, and each concrete *sensor* subclass maps system-specific events to platform-independent events. For each platform supported, implement concrete *display* and *sensor* subclasses that encapsulate system specifics.

The design of the abstract classes *display* and *sensor* is non-trivial, because it impacts both the efficiency of the resulting code, and the efficiency with which the concrete classes can be implemented on the different platforms. One approach is to use *sensor* and *display* abstractions with only the very basic functionality that is provided directly by all user-interface platforms. The other extreme is to have *display* and *sensor* offer higher-level abstractions. Such classes need greater effort to port, but use more native code from the user-interface platform. The first approach leads to applications that look similar across platforms, while the second results in applications that conform better to platform-specific guidelines.

Variants *Document-View.* This variant relaxes the separation of view and controller. In several GUI platforms, window display and event handling are closely interwoven. For example, the X Window System reports events relative to a window. You can combine the responsibilities of the view and the controller from MVC in a single component by sacrificing exchangeability of controllers. This kind of structure is often called a Document-View architecture [App89], [Gam91], [Kru96]. The document component corresponds to the model in MVC, and also implements a change-propagation mechanism. The view component of Document-View combines the responsibilities of controller and view in MVC, and implements the user interface of the system. As in MVC, loose coupling of the

document and view components enables multiple simultaneous synchronized but different views of the same document.

Known Uses **Smalltalk** [GR83]. The best-known example of the use of the Model-View-Controller pattern is the user-interface framework in the Smalltalk environment [LP91], [KP88]. MVC was established to build reusable components for the user interface. These components are shared by the tools that make up the Smalltalk development environment. However, the MVC paradigm turned out to be useful for other applications developed in Smalltalk as well. The VisualWorks Smalltalk environment supports different 'look and feel' standards by decoupling view and controllers via *display* and *sensor* classes, as described in implementation step 10.

MFC [Kru96]. The Document-View variant of the Model-View-Controller pattern is integrated in the Visual C++ environment—the Microsoft Foundation Class Library—for developing Windows applications.

ET++ [Gam91]. The application framework ET++ also uses the Document-View variant. A typical ET++-based application implements its own document class and a corresponding view class. ET++ establishes 'look and feel' independence by defining a class `WindowPort` that encapsulates the user interface platform dependencies, in the same way as do our *display* and *sensor* classes.

Consequences The application of Model-View-Controller has several **benefits**:

Multiple views of the same model. MVC strictly separates the model from the user-interface components. Multiple views can therefore be implemented and used with a single model. At run-time, multiple views may be open at the same time, and views can be opened and closed dynamically.

Synchronized views. The change-propagation mechanism of the model ensures that all attached observers are notified of changes to the application's data at the correct time. This synchronizes all dependent views and controllers.

'Pluggable' views and controllers. The conceptual separation of MVC allows you to exchange the view and controller objects of a model. User interface objects can even be substituted at run-time.

Exchangeability of 'look and feel'. Because the model is independent of all user-interface code, a port of an MVC application to a new platform does not affect the functional core of the application. You only need suitable implementations of view and controller components for each platform.

Framework potential. It is possible to base an application framework on this pattern, as sketched in implementation steps 7 through 10. The various Smalltalk development environments have proven this approach.

The **liabilities** of MVC are as follows:

Increased complexity. Following the Model-View-Controller structure strictly is not always the best way to build an interactive application. Gamma [Gam91] argues that using separate model, view and controller components for menus and simple text elements increases complexity without gaining much flexibility.

Potential for excessive number of updates. If a single user action results in many updates, the model should skip unnecessary change notifications. It may be that not all views are interested in every change-propagated by the model. For example, a view with an iconized window may not need an update until the window is restored to its normal size.

Intimate connection between view and controller. Controller and view are separate but closely-related components, which hinders their individual reuse. It is unlikely that a view would be used without its controller, or vice-versa, with the exception of read-only views that share a controller that ignores all input.

Close coupling of views and controllers to a model. Both view and controller components make direct calls to the model. This implies that changes to the model's interface are likely to break the code of both view and controller. This problem is magnified if the system uses a multitude of views and controllers. You can address this problem by applying the Command Processor pattern (277), as described in the Implementation section, or some other means of indirection.

Inefficiency of data access in view. Depending on the interface of the model, a view may need to make multiple calls to obtain all its display data. Unnecessarily requesting unchanged data from the model weakens performance if updates are frequent. Caching of data within the view improves responsiveness.

Inevitability of change to view and controller when porting. All dependencies on the user-interface platform are encapsulated within view and controller. However, both components also contain code that is independent of a specific platform. A port of an MVC system thus requires the separation of platform-dependent code before rewriting. In the case of an MVC framework or a large composed application, an additional encapsulation of platform dependencies may be required.

Difficulty of using MVC with modern user-interface tools. If portability is not an issue, using high-level toolkits or user interface builders can rule out the use of MVC. It is usually expensive to retrofit toolkit components or the output of user interface layout tools to MVC. Additional wrapping would be the minimum requirement. In addition, many high-level tools or toolkits define their own flow of control and handle some events internally, such as displaying a pop-up menu or scrolling a window. Finally, a high-level user interface platform may already interpret events and offer callbacks for each kind of user activity. Most controller functionality is therefore already provided by the toolkit, and a separate component is not needed.

See Also The Presentation-Abstraction-Control pattern (145) takes a different approach to decoupling the user-interface aspects of a system from its functional core. Its abstraction component corresponds to the model in MVC, and the view and controller are combined into a presentation component. Communication between abstraction and presentation components is decoupled by the control component. The interaction between presentation and abstraction is not limited to calling an update procedure, as it is within MVC.

Credits Trygve Reenskaug created MVC and introduced it to the Smalltalk environment [RWL96].