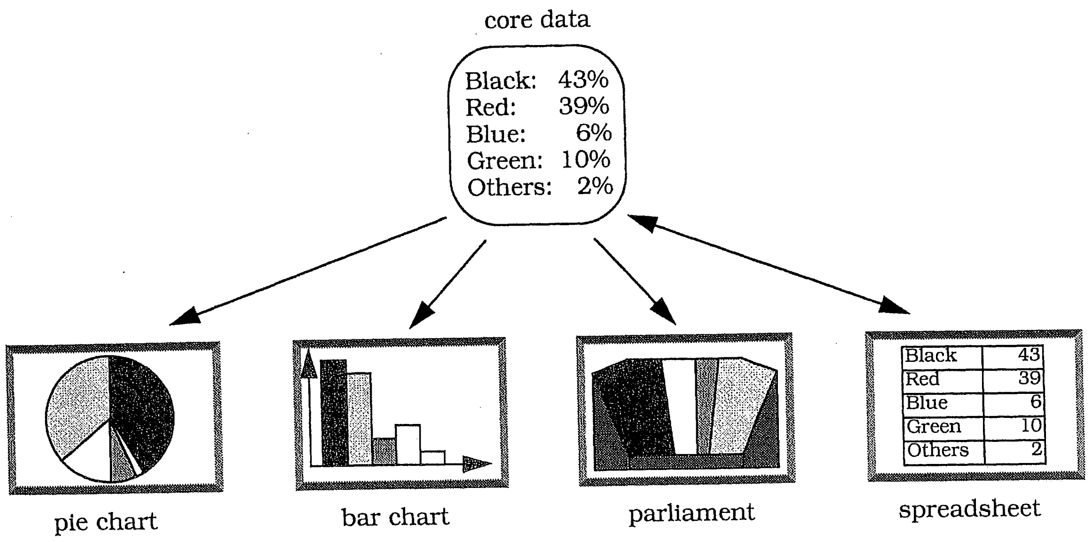
Model-View-Controller

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***
Responsibility	• View • Controller
• 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.

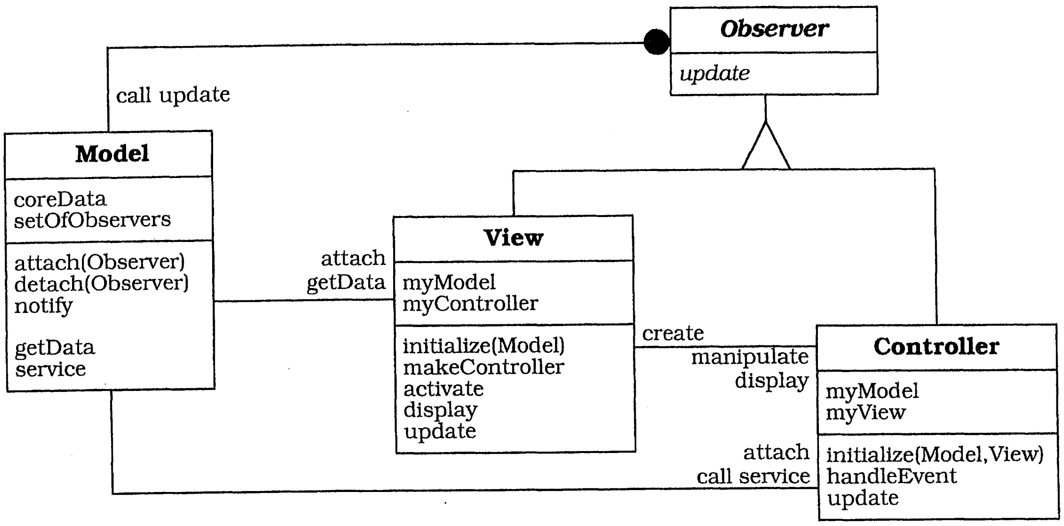
The *controller* components accept user input as events. How these events are delivered to a controller depends on the user interface platform. For simplicity, let us assume that each controller implements an event-handling procedure that is called for each relevant event. Events are translated into 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*** • Controller • Model
Responsibility • Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	

Class Controller	***Collaborators*** • View • Model
Responsibility • 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.

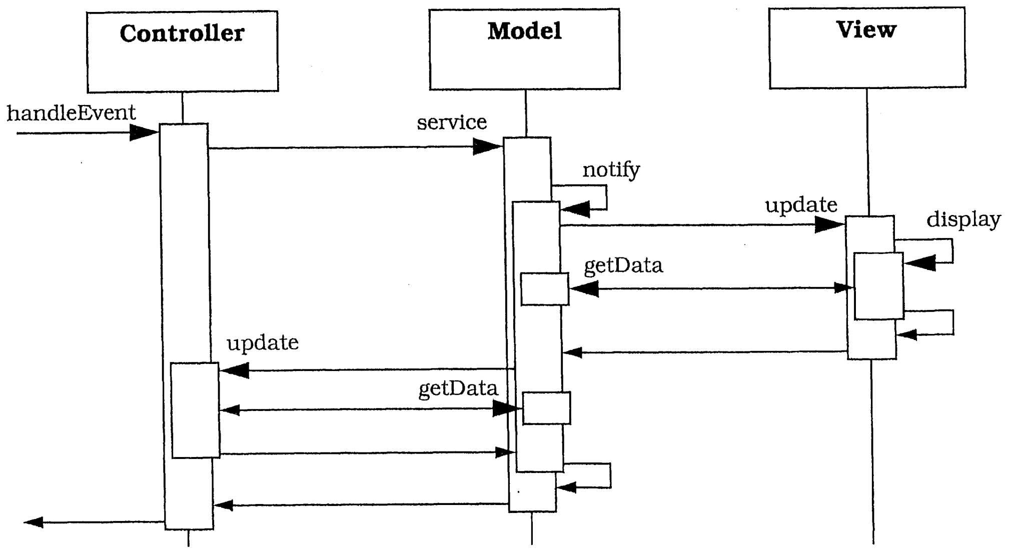
We define several views: a bar chart, a pie chart and a table. The chart views use controllers that do not affect the model, whereas the table view connects to a controller used for data entry. ❑

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.

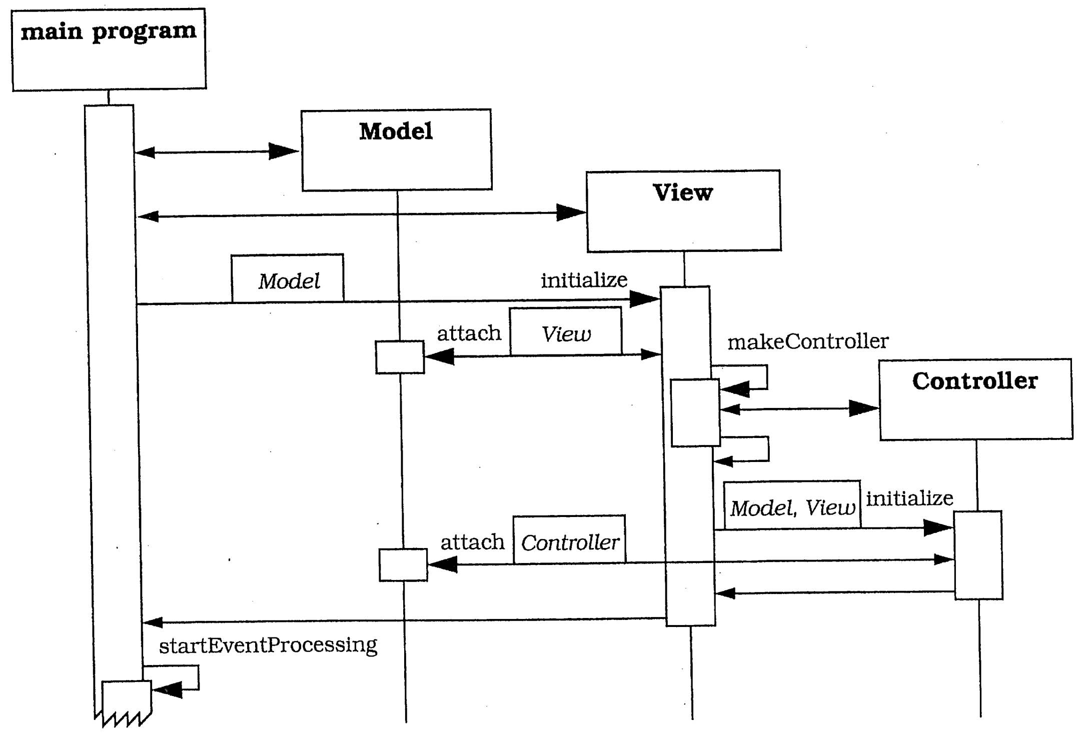
Scenario I shows how user input that results in changes to the model triggers the change-propagation mechanism:

- 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.


main program
Model
View
Model
initialize
attach
View
makeController
Controller
Model, View
initialize
attach
Controller
startEventProcessing