Introduction to Model-View-Controller

14

The objective of this chapter is to introduce the Model–View–Controller (MVC) architectural pattern.

14.1 Preconditions

The following should be true prior to starting this chapter.

- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.

14.2 Concepts and Context

The case study software designs discussed in Chaps. 12 and 13 exhibit similar design characteristics. The OOD and SD designs have four characteristics in common suggesting the designs are good. These four design characteristics are simple, have low or weak coupling, have good (logarithmic) time performance, and they do input and output data validation. However, both the OOD and SD designs have low or weak cohesion, which is a bad design trait. This chapter introduces an architectural pattern called Model–View–Controller as a way to improve the cohesion of these designs while still maintaining the good characteristics these designs already exhibit.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_14

14.2.1 Introduction to Model–View–Controller (MVC)

Model–View–Controller (MVC) is a software architecture or framework that splits a software application into three design components.

- The *model component* is responsible for managing data. This may include interfacing with any persistent data storage and using memory-based data structures to store data during execution. The model component implements logic to allow the application to create, read, update, and delete (CRUD) the application data.
- The *view component* is responsible for providing an interface for user interactions, assuming the software requires a user interface.
- The *controller component* is responsible for the domain logic, also called business logic, associated with the software. This includes having the controller communicate with the view and model components. In essence, the controller component includes code that *glues* these three components together.

There are two clear benefits to using MVC. First, keeping the user interface separate from data management allows the technology for one of these components to be changed without impacting the technology used by the other component. For example, we can change the persistent data store from XML to a relational database without needing to change anything in the view (i.e., user interface) component. Likewise, the user interface can be changed from a Java-based graphical user interface to a mobile-based application without having to change anything in the model (i.e., data management) component. Second, using MVC results in having three components that are each more cohesive. As will be shown in Chaps. 15 and 16, each MVC component is more cohesive given the more focused set of responsibilities assigned to the component.

The MVC software architecture may be explained using the IDEF0 function model, as shown in Fig. 14.1. The *View* component sends data provided by the user to the *Controller*. The Controller may validate this data and/or apply domain rules to transform this input data before sending *domain data* to the Model component. Note the *UI data* output arrow from the View goes into the Controller as a control, to show this data is being validated, and as an input, to show this data is being transformed by domain rules. As a result of processing done by the Controller, domain data is sent to the *Model*. The Model may validate this data (as shown by the *domain data* control arrow) and/or treat this data as input (as shown by the *domain data* input arrow). The Model will send *model data* to the Controller, or optionally to the View for display to the user. When the model data is sent to the Controller, this data may be validated and/or transformed by applying domain rules, and then given back to the Model component or sent to the View for display to the user.

Figure 14.1 is a good example of the ICOM arrows in an IDEF0 function model providing too much information for the type of design abstraction we want to express. Essentially, the generic description of MVC using IDEF0 is difficult to read given the number of ICOM arrows used to express MVC. For this reason, a data-flow diagram (DFD) is shown in Fig. 14.2. A DFD includes notations that represent a process

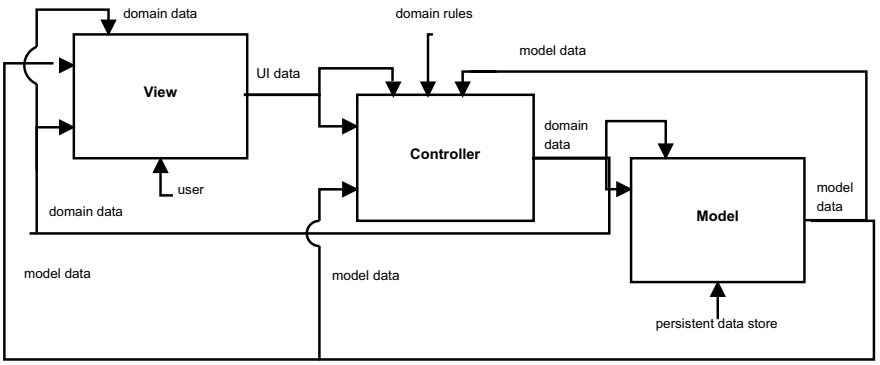
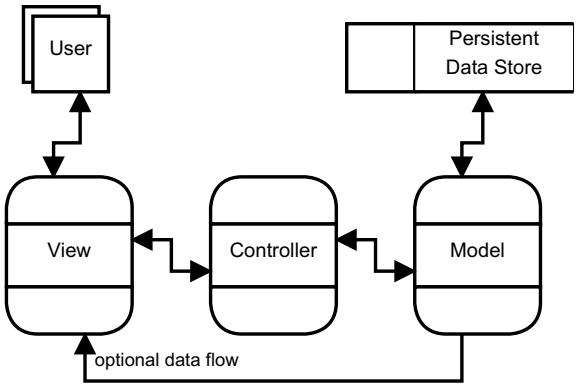


Fig. 14.1 Generic Model–View–Controller IDEF0 function model

Fig. 14.2 Generic Model–View–Controller data-flow diagram



(e.g., see View, Controller, and Model), data flows (e.g., see the directed arrows that connect the other notations), external entities (e.g., see User), and data stores (e.g., see Persistent Data Store). Since a DFD data flow can represent any type of flow between components, it represents three of the ICOM arrows—input, control, and output. Thus, the DFD in Fig. 14.2 is significantly easier to read and more clearly shows the MVC architectural pattern.

Based on the two architecture diagrams shown in Figs. 14.1 and 14.2, the following identifies the key characteristics of the MVC architecture/framework.

- The user interacts only with the view component.
- A persistent data store is used only by the model component. The model is also responsible for using memory-based data structures for temporary data storage.
- The view and controller communicate with each other. Data may flow in either direction between these two components.
- The model and controller communicate with each other. Data may flow in either direction between these two components.
- Optionally, the model may send data directly to the view, but not vice versa.

The last bullet is worth a little discussion. Since the model is responsible for managing the data, it may be a good idea (e.g., for performance reasons) to have the model send data directly to the view for display to the user without involving the controller. On the other hand, any data coming from the view (which may have been modified by a user, perhaps one with malicious intent) must go through the controller before the model is allowed to update a data store. This places responsibility for validating user data in the controller. The validation done by the controller would be specific to the application domain being implemented. (It is also likely the model does some data validation based on the type of persistent data store being used.)

14.2.1.1 MVC in Distributed Applications

The ABA case study is a standalone software application, which means that it runs on a single computing device. Given this, the three components communicate with each other via function/method calls. How might the MVC description change if we are applying this architectural pattern to a distributed application?

Figure 14.3 shows a client–server (or 2-tier) architecture for MVC, where the client contains the view and a client-side controller while the server contains the model and server-side controller. The controller component is essentially split between the two tiers, resulting in data validation occurring on both the client and server devices. Why should validation be done on both tiers? First, it may be fairly easy for a user with malicious intent to create a client that communicates with the server. If the server were to assume that validation is done by the client, and thus the server does no validation, then the spoofed client could inject malicious data, do an SQL injection attack, or perform some similarly bad action that results in a less secure application. Second, if the client were to assume that validations are being done by the server, and thus the client does no validation, it may be possible for an attacker to spoof the server side of this application, resulting in the validation not being done. In this scenario, the user of the client may be providing sensitive data to the spoofed server.

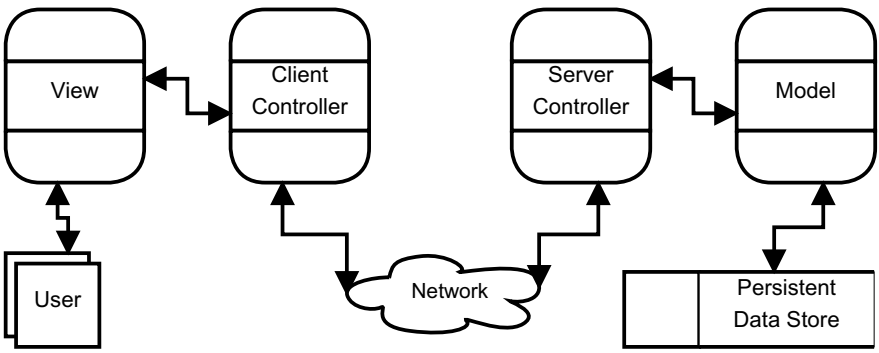


Fig. 14.3 Generic 2-tier Model–View–Controller data-flow diagram

We can extend this discussion to a 3-tier or n-tier application that uses the MVC architecture. Like the 2-tier discussion just presented, any component that communicates with another over a network should not assume the other component is doing validation. Validation needs to be done by each distributed component to avoid malicious users from taking advantage of your less-secure design.

14.3 Post-conditions

The following should have been learned when completing this chapter.

- You understand Model–View–Controller is a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.

The following should have been learned when completing previous chapters.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand the notion of abstraction as the process of generalizing a concept by removing details that are not needed to properly convey the design perspective being emphasized.

Exercises

Discussion Questions

1. What are the benefits of separating the user interface from the application data?
2. In a standalone application using MVC, which component should validate data and why?
3. Give an example of a type of application that would benefit from having the model component send data directly to the view?
4. Pick an application domain, identify requirements for this domain, and then assign each requirement to one or more of the MVC components.