User Interface Architecture: MVC, MVP, MVVM

An interactive application provides a user interface that lets people view and manipulate information, as well as invoke the functionality of the application. This article describes the MVC, MVP, and MVVM software design patterns that are useful for implementing interactive applications.

Underlying Principles

We will first explain the design principles that motivate the structure of the MVC, MVP, and MVVM design patterns.

Separation of Responsibilities

The design of an interactive application contains at least two types of code:

1. Business/Domain Logic (a.k.a. the Model) - The layer of code that stores the core data for the application and implements the algorithms that operate on that data, independent of the way it is presented by the user interface. Because this code "models" the application domain, it is often called the "Model".
2. User Interface - The layer of code responsible for interacting directly with the user. The user interface displays data to users, lets them modify the data, and lets them invoke the functionality of the application.

A well-designed application keeps Model code and User Interface code strictly separate. These two types of code should not be mingled in the same class or method. This separation is desirable for the following reasons:

1. More understandable. Keeping Model code and User Interface code separate results in a more understandable code base. (See Single Responsibility Principle)
2. More maintainable. Model code and User interface code changes at different times for different reasons. Therefore, keeping them separate makes it possible to change one without affecting the other. (See Single Responsibility Principle)
3. More reusable. Separating Model code from User Interface code allows the Model to be reused with different user interface implementations. For example, an application might have a web interface, a desktop interface, and a mobile interface, all of which are built using the same Business/Domain Logic layer, but with different User Interface layers. Or, an application might have different user interfaces, each targeted at a different kind of user. For example, a bank might have different interfaces for tellers, customers, loan officers, etc., but all of these interfaces could use the same Model code.
4. More convenient. On some projects there are different teams of people who work on the Model and User Interface parts of the application. In this case, separating Model and

User Interface code has the very practical benefit of letting people do their work without interfering with each other.

The User Interface layer also contains two different types of code:

1. Display Code: Code that displays (or outputs) content on the screen.
2. Input Processing Code: Code that processes user input, such as mouse clicks, hand gestures, and keyboard input.

For the same reasons it is good to separate Model and User Interface code (understandability, maintainability, etc.), it is also good to keep Display code separate from Input Processing code by implementing them in separate classes. Classes that contain Display code are often referred to as "view" classes, and classes that contain Input Processing code are often referred to as "controller" classes. Another reason to keep Display code and Input Processing code in separate classes is that Display code can be very difficult to write automated tests for, while Input Processing code is fairly easy to write automated tests for. Therefore, separating Display and Input Processing code has the effect of maximizing our ability to write automated tests for our code.

In summary, there are three kinds of code that need to be kept separate from each other.

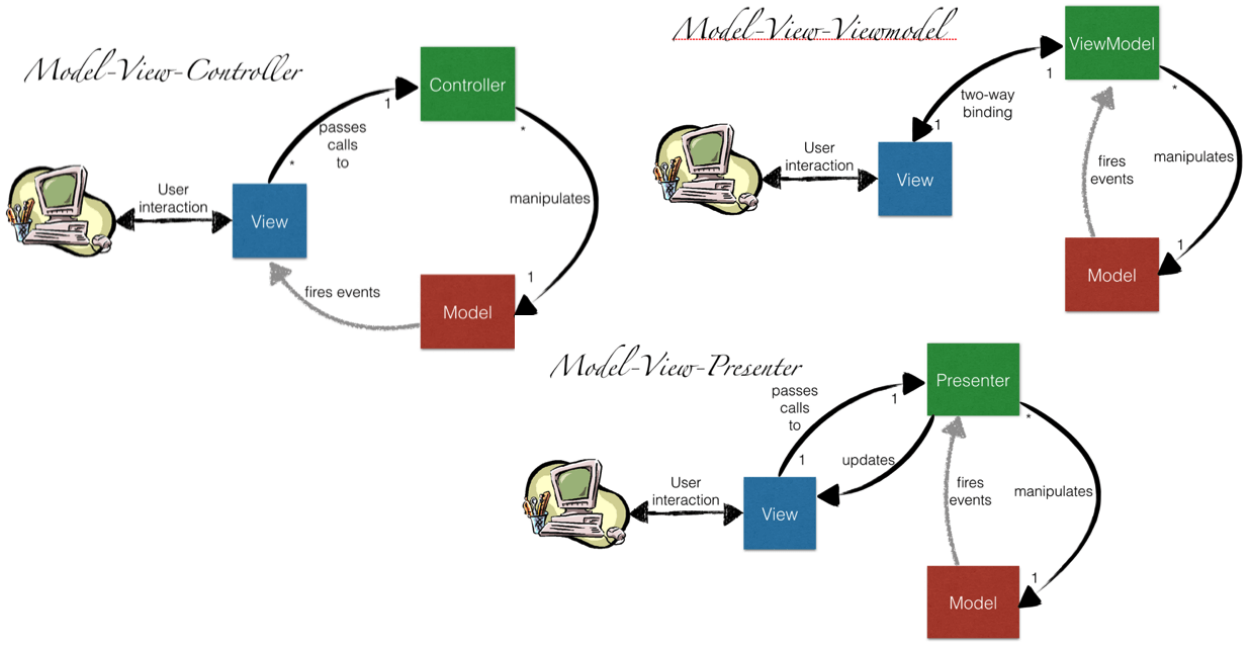
1. Model code
2. Display code
3. Input Processing code

Observable Model

The user interface displays data from the Model on the screen. This means there are actually two copies of the application's data, one copy stored in the Model and one copy displayed on the screen. A big part of user interface programming is keeping the data in the Model in sync with the data on the screen. Specifically, when data in the Model changes, any copies of that data currently on the screen must be updated. In the other direction, when the user modifies a piece of data on the screen, corresponding changes must be made to that data in the Model.

To simplify the task of keeping the data in the Model in sync with the data displayed on the screen, Model classes can implement the Observer pattern. User interface objects can register as observers of the Model so they will be automatically notified when Model data changes, thus allowing them to update the corresponding data on the screen.

The MVC, MVP, and MVVM Design Patterns



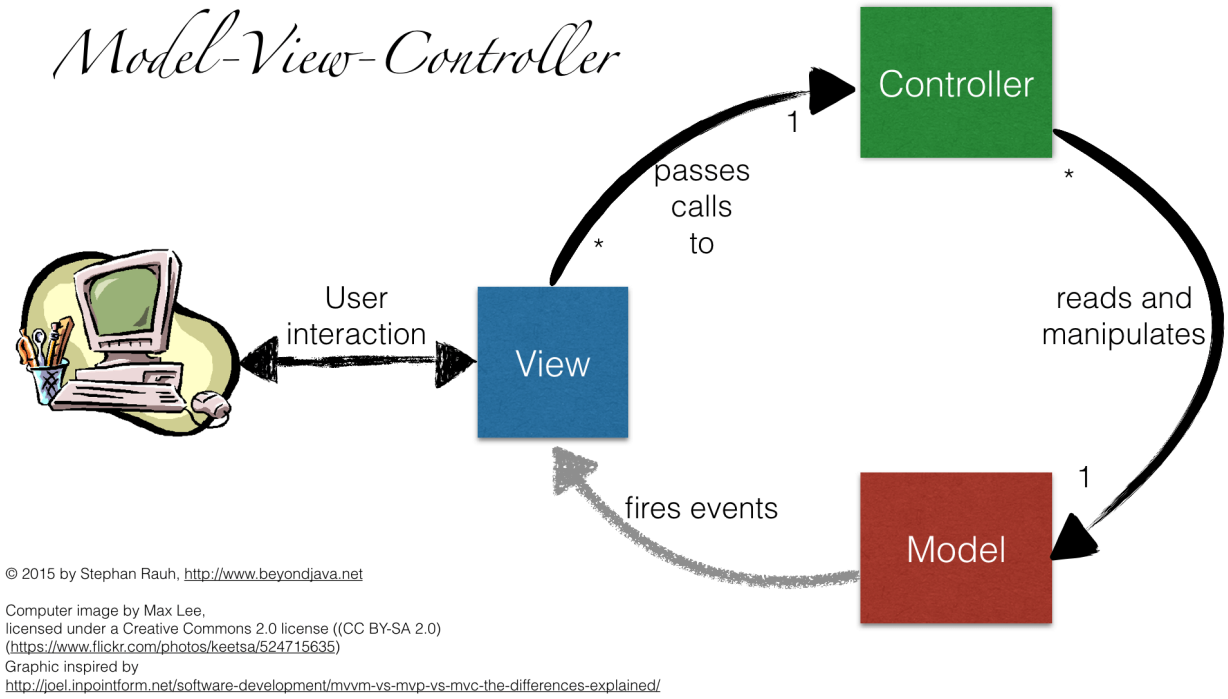
The popular Model-View-Controller, Model-View-Presenter, and Model-View-ViewModel design patterns are all based on the separation of responsibilities described previously.

1. Models - In all three patterns, the Model classes contain core business/domain data and algorithms, and implements the Observer design pattern.
2. Views - In all three patterns, the user interface is divided into "views". A "view" can be any part of the user interface. In a desktop application, each window in the interface could be a view. In a mobile application, each screen in the interface could be a view. In a web application, each page in the interface could be a view. A complex view can also be subdivided into multiple views. Really, a view can be any part of the interface you want it to be. For each view we create a "View" class in the code which is responsible for displaying output to its part of the interface.
3. Controllers - In all three patterns, for each "View" class we create a corresponding "Controller" class that handles user input processing for the view. This means that each part of the interface is implemented by a pair of classes, a "View" class to display output, and a "Controller" class to process user input. Depending on which pattern you're talking about, the input processing classes are called "controllers", "presenters", or "view models", but they all play the same essential role - input processing.

Having described the similarities between these patterns, the following sections describe each pattern in detail, including their differences.

Model-View-Controller

The MVC pattern is the original of the three patterns, and today is used less frequently than the MVP and MVVM patterns. However, it is still worth understanding MVC to put MVP and MVVM in context.



Graphic inspired by
http://joel.inpointform.net/software-development/mvvm-vs-mvp-vs-mvc-the-differences-explained/

MVC Setup

When an interactive program starts up, one of the first things it does is initialize its user interface. This involves creating all of the View and Controller objects for each view and connecting them together, as well as connecting each View and Controller to the Model. Once all of the Views and Controllers have been initialized, the user interface can be displayed on the screen, and user input can be processed.

View and Controller have references to each other so they can call methods on each other.
View and Controller both have references to Model so they can call methods on Model.
View registers as an observer of Model so it is notified of data changes in Model.

MVC Interaction

1. Views query data from Model
2. Views display data
3. Views wait for user input
4. View receives user input and passes to Controller

5. Controller
    a. Modifies data in Model
    b. Invokes algorithms in Model (which update data)
    c. Sets state of View
        i. Display message to user
        ii. Enable/disable UI components
        iii. Set sort order of data
        iv. Display "busy" feedback
        v. Etc.
6. Model notifies Observers (i.e., Views) that data has changed
7. Views re-query data from Model
8. Views display new data
9. Views wait for user input

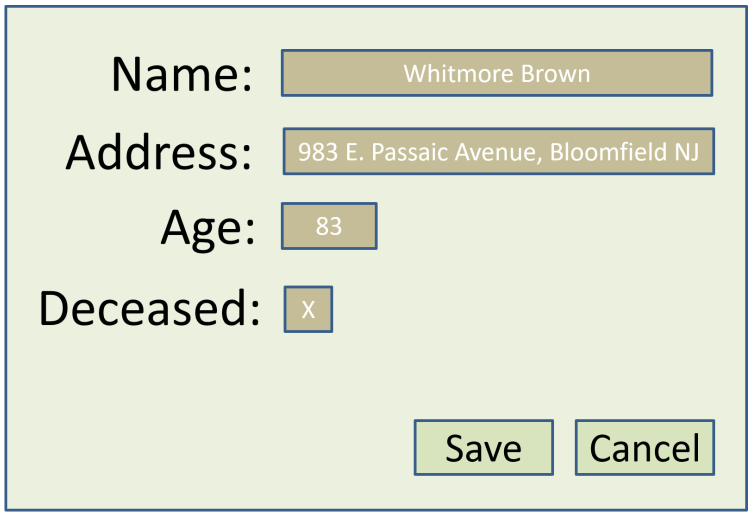
NOTE: The MVC diagram above does not show the method calls between View and Model nor the method calls between Controller and View (these graphics were borrowed from another article, and they were incomplete in this aspect).

In the MVC pattern, the View decides what data to display and directly retrieves the data it wants from the Model. This means that View contains a fair amount of intelligence, and has a dependency on the Model. The MVP and MVVM patterns remove this intelligence from the View, thus eliminating the dependency on the Model, which has advantages that are discussed later.

If multiple Views are visible on the screen simultaneously, the data in all of the views will be kept in sync because all Views observe the same Model. When data changes in the Model, all Views are notified of the change, and they all refresh their displayed data simultaneously. This means that views can interact with each other through the Model without knowing about each other. Therefore, the views are decoupled from each other, and new views can be added without affecting existing views. This is a very powerful concept.

MVC Example

As a concrete example, suppose we have a view that lets you edit the properties of a person (name, address, age, deceased), as shown below:



In this case, the PersonController class would implement methods that the PersonView class would call to pass user inputs to it. In this case, the user inputs include changes to the name, address, age, and deceased fields in the view, and user clicks on the “Save” and “Cancel” buttons. PersonController’s method interface would look something like this:

```
interface IPersonController {
        void nameChanged();
        void addressChanged();
        void ageChanged();
        void deceasedChanged();

        void save();
        void cancel();
}
```

These methods would be called by PersonView any time one of those inputs occurs in the view, and PersonController would respond in whatever way it deemed appropriate.

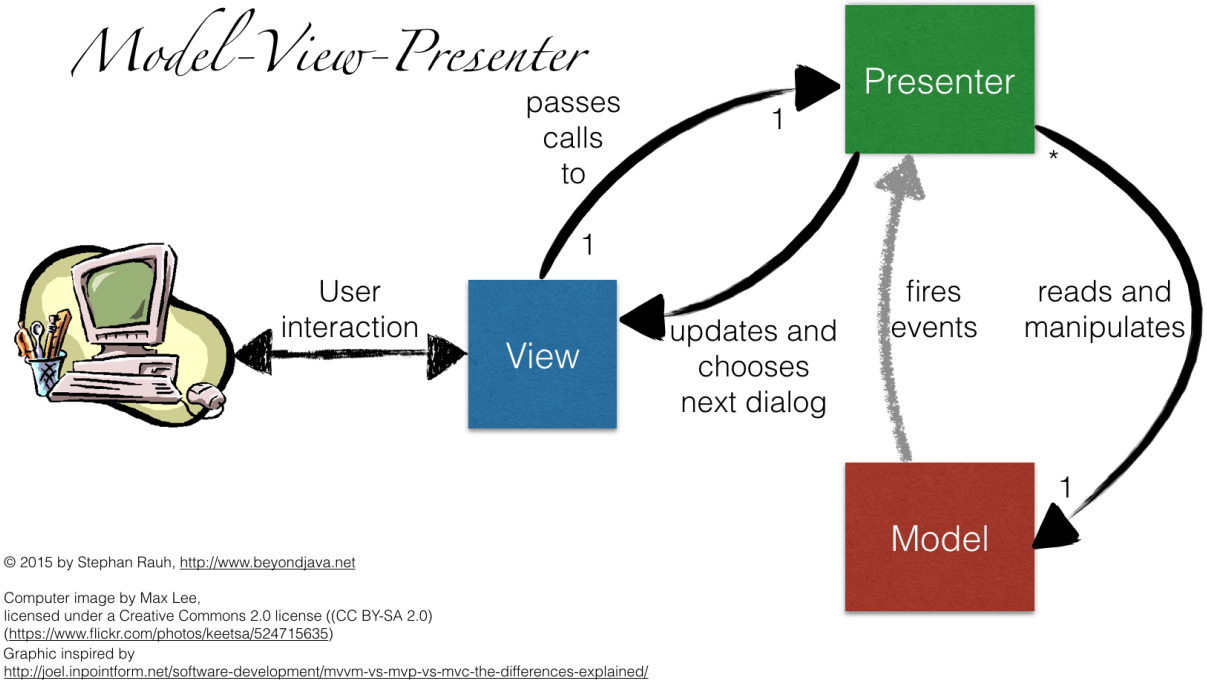
In the other direction, the PersonView class would implement methods that the PersonController class would call to retrieve data values from the view, enable or disable the buttons, and display messages to the user. PersonView’s method interface would look something like this:

```
interface IPersonView {
        String getName();
        String getAddress();
        String getAge();
        boolean getDeceased();

        void enableSave(boolean value);
```

```
        void enableCancel(boolean value);

        void displayMessage(String value);
}
```

Model-View-Presenter

The MVP pattern is a variation of the MVC pattern that results in more testable code.



In the MVP pattern, the object that processes user input is called “Presenter” instead of “Controller”. The MVP pattern also moves as much intelligence as possible out of the View into the Presenter. Specifically, the responsibility of deciding what data to display on the screen and retrieving that data from the Model is moved from the View into the Presenter. After retrieving data from the Model, the Presenter passes the data to the View. The View displays whatever data it is given by Presenter. MVP also moves the responsibility of observing the Model for data changes from the View to the Presenter.

This structure results in a “smarter” Presenter and a “dumber” View. The Presenter is the “brains” of the operation, and the View just does what he’s told. This division of responsibilities is often more advantageous than MVC because View code can be difficult to test in an automated way. This is because Views often depend on user interface libraries and frameworks that do not lend themselves well to automated testing. Specifically, it can be difficult for an automated test to tell if the right content is on the screen, making it hard to determine

programmatically if a test case succeeded or failed (Web frameworks are a notable exception to this). For this reason, View testing is often done manually. By moving as much logic as possible into the Presenter, the amount of code on which automated testing can be done is maximized. As a general rule, code that depends on the user interface library or framework should go in the View, and code that does not have this dependency should go in the Presenter.

MVP Setup

View and Presenter have references to each other so they can call methods on each other.
Presenter has a reference to Model so it can call methods on Model.
Presenter registers as an observer of Model so it is notified of data changes in Model.

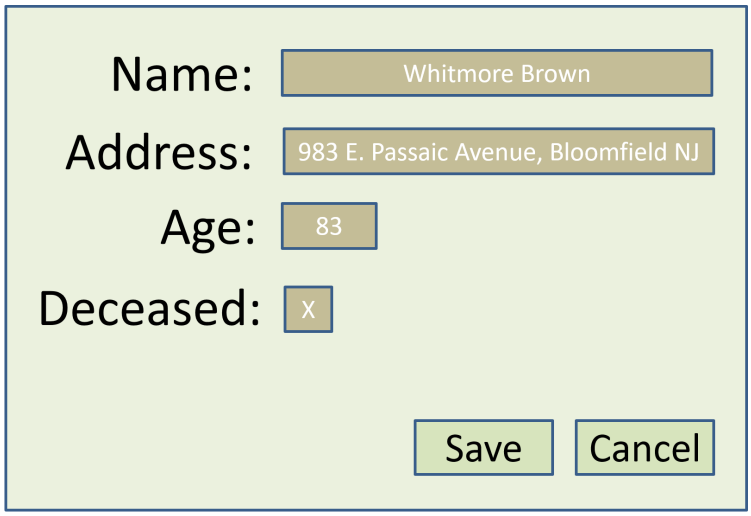
MVP Interaction

1. Presenters query data from Model
2. Presenters push data into Views
3. Views display data
4. Views wait for user input
5. View receives user input and passes to Presenter
6. Presenter
    a. Modifies data in Model
    b. Invokes algorithms in Model (which update data)
    c. Sets state of View
        i. Display message to user
        ii. Enable/disable UI components
        iii. Set sort order of data
        iv. Display "busy" feedback
        v. Etc.
7. Model notifies Observers (i.e., Presenters) that data has changed
8. Presenters re-query data from Model
9. Presenters push new data into Views
10. Views display new data
11. Views wait for user input

As with the MVC pattern, if multiple Views are visible on the screen simultaneously, the data in all of the views will be kept in sync because all Views observe the same Model.

MVP Example

Let’s return to the person editor example shown below:



In this case, the PersonPresenter class would implement methods that the PersonView class would call to pass user inputs to it, including changes to the name, address, age, or deceased fields in the view, and user clicks on the “Save” and “Cancel” buttons. Therefore, PersonPresenter’s method interface would look something like this:

```
interface IPersonPresenter {
        void nameChanged();
        void addressChanged();
        void ageChanged();
        void deceasedChanged();

        void save();
        void cancel();
}
```

This interface is identical to the controller interface in the MVC example.

In the other direction, the PersonView class would implement methods that the PersonPresenter class would call to pass data into the view, retrieve data values from the view, enable or disable the buttons, and display messages to the user. Therefore, PersonView’s method interface would look something like this:

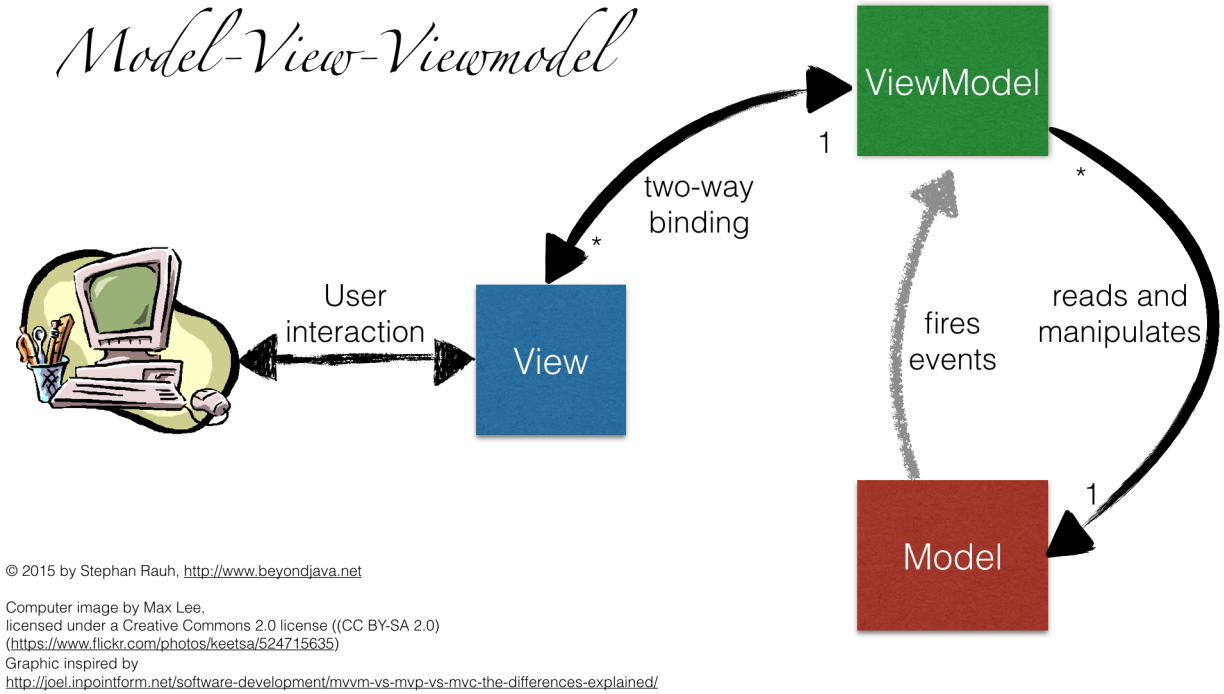
```
interface IPersonView {
        void setName(String value);
        void setAddress(String value);
        void setAge(String value);
        void setDeceased(boolean value);

        String getName();
        String getAddress();
```

```
        String getAge();
        boolean getDeceased();

        void enableSave(boolean value);
        void enableCancel(boolean value);

        void displayMessage(String value);

        String getName();
        String getAddress();
        String getAge();
        boolean getDeceased();

        void enableSave(boolean value);
        void enableCancel(boolean value);

        void displayMessage(String value);
}
```

This interface is similar to the view interface from the MVC example, except that there are now methods for “setting” as well as “getting” data values in the view. These “setting” methods are necessary, because in MVP the Presenter tells the view what data to display.

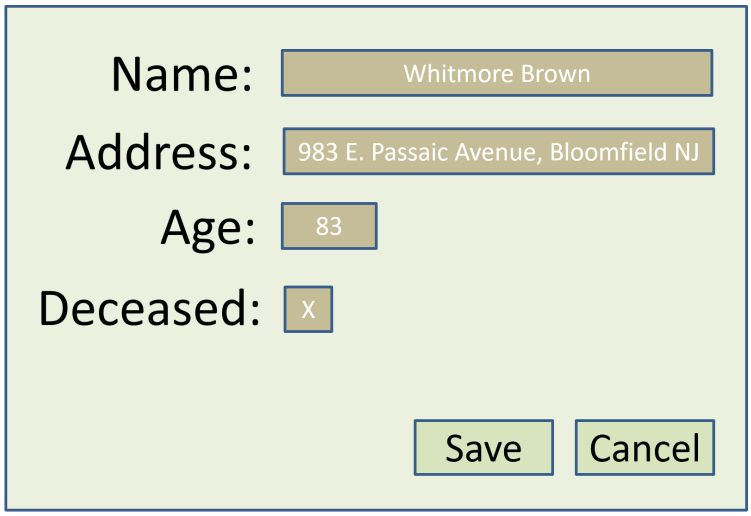
Model-View-ViewModel

MVC and MVP assume that view creators are skilled programmers, as evidenced by the fact that the View and Controller/Presenter objects call methods on each other to communicate. This approach is fine for programmers, but user interfaces are often designed by user interface designers who are not programmers. To minimize the amount of programming required in views, the MVVM pattern lets user interface designers define the graphical contents of views using declarative languages like HTML or XML, and use “data binding” instead of method calls to connect a view with the data it should display.



In the MVVM pattern, the Controller/Presenter is replaced by an object called a “ViewModel”. Each View has a corresponding ViewModel. As with Presenters in MVP, a ViewModel retrieves data from the Model to be displayed, and also processes user input. The difference between Presenters in MVP and ViewModels in MVVM is the way that data is passed between the ViewModel and the View. With MVP, data is passed back and forth between Presenter and View via method calls. With MVVM, data is passed back and forth between ViewModel and View using “data binding”.

To implement data binding, the ViewModel class declares properties whose values directly map to the data values displayed in the View. The ViewModel also declares properties that control the current state of the View, such as the enable/disable state of the various UI controls and messages displayed to the user on the screen. Additionally, the ViewModel class implements methods that directly map to the operations users can perform in the View. In this way, the ViewModel “models the view”, meaning that its properties and methods directly model the data and actions in the View’s user interface.



In the person editor example, the ViewModel class would look something like this:

```
class PersonViewModel {

        // Properties
        String name;
        String address;
        int age;
        boolean deceased;

        boolean isSaveEnabled;
        boolean isCancelEnabled;

        String message;

        // Operations
        void nameChanged() { ... }
        void addressChanged() { ... }
        void ageChanged() { ... }
        void deceasedChanged() { ... }

        void save() { ... }
        void cancel() { ... }
}
```

Given this ViewModel class, the View would be implemented in a declarative language such as XML. The elements in the XML would represent the UI controls in the view and their layout. The attributes of the UI controls would be declaratively bound to the properties and methods of the ViewModel class. For example, the View for the person editor might look like this:

```
<Table>
```

```
        <Row>
                <Label>Name:</Label>
                <TextEdit value="viewmodel.name"
                        changed="viewmodel.nameChanged()" />
        </Row>
        <Row>
                <Label>Address:</Label>
                <TextEdit value="viewmodel.address"
                        changed="viewmodel.addressChanged()" />
        </Row>
        <Row>
                <Label>Age:</Label>
                <TextEdit value="viewmodel.age"
                        changed="viewmodel.ageChanged()" />
        </Row>
        <Row>
                <Label>Deceased:</Label>
                <CheckBox value="viewmodel.deceased"
                        changed="viewmodel.deceasedChanged()" />
        </Row>
        <Row>
                <Button action="viewmodel.save()"
                        enabled="viewmodel.isSaveEnabled">Save</Button>
                <Button action="viewmodel.cancel()"
                        enabled="viewmodel.isCancelEnabled">Cancel</Button>
        </Row>
        <Row visible=”viewmodel.message != null”>
                <Label value=”viewmodel.message” />
        </Row>
</Table>
```

The values of the name, address, age, and deceased UI controls are bound to the properties of the PersonViewModel class. At runtime, the MVVM framework would keep the values of the ViewModel properties and UI controls constantly in sync. Initially, the UI controls' values would be initialized using the ViewModel property values. Then, whenever the user interactively modifies the value of a UI control, the new value would be automatically propagated to the ViewModel properties. And, when the properties of the ViewModel class change for any reason, their new values would be automatically propagated to the UI controls. This is called “data binding”.

Additionally, the “changed” and “action” events of the person editor controls are bound to methods on the PersonViewModel class. This means that when a “changed” or “action” event occurs in one of the controls, the ViewModel method to which it is bound will be called automatically.

The enable/disable states of the buttons are bound to the “isSaveEnabled” and “isCancelEnabled” ViewModel properties. As a result, the buttons can be enabled or disabled simply by modifying the values of these ViewModel properties.

Finally, if the program wants to display a message on the screen, all it must do is set the value of the ViewModel’s “message” property. The UI controls that display the message are only visible when the message property has a non-null value. When there is no message to be displayed, the controls will automatically disappear. When there is a message to be displayed, the controls will automatically appear and display the message.

This style of programming is very convenient, even for people who can program, because the MVVM framework does many things automatically without the programmer having to write much code. And, the View logic is simple enough that even non-programmers can write it, making it accessible to user interface designers.