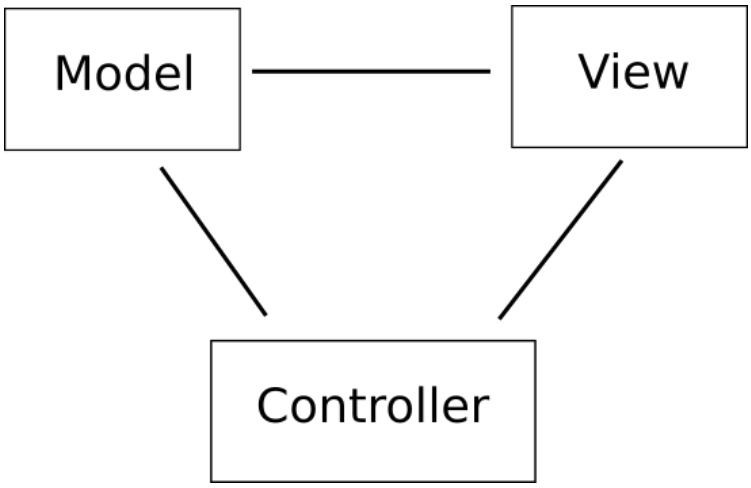
Stage 6: video 1

Scene 1: On-Set



The Model-View-Controller or MVC pattern is more abstract than the others we've discussed. Rather than structuring *objects* or *interactions* between objects, MVC defines the relationship between subsystems composed of classes or, more generally, source modules. The pattern separates an application into three functional parts:

1. The **Model** is the core of the application; this is what provides the unique capabilities that justifies the development of the software. Often this functionality includes data management and persistence, but might include network communication, or complex calculations. The model should be a nearly autonomous subsystem.
2. The **View** presents data from the **model** and provides an interface for user interaction. Ideally, views should include very little logic or "smarts" beyond what is directly necessary to receive input or display data on a screen.
3. The **Controller** facilitates interaction between the **Model** and the **View** and manages functionality that does not clearly fall within the purview of either. The controller encapsulates the functionality that allows the model and view subsystems to specialize and remain decoupled. Often the controller is described as "glue" or "insulation" between the model and view.

The software *virtue* extolled by MVC is modularity, which helps the programmer manage the complexity of developing an application. It allows the Model, View, and Controller modules to focus on a limited range of functionality. In short, it allows them to specialize. As a general rule, it is nearly always preferable to have parts with limited functionality that can be combined, rather than complex monolithic systems that are hard to separate.

MVC is unique among the patterns we've looked at in its general applicability. It was originally introduced in Smalltalk in the 70s and it is language and platform agnostic. The Model-View-Controller design pattern may be applied to the development of nearly any application that supports a graphical user interface(or GUI).

I'll illustrate the principle of MVC with an example from the physical world.

<MOTION>
Our analogy is an old-school landline telephone. The key functionality of this device is the subsystem that physically connects to the line leading to the home or business and transmits and receives signals to and from that wire. Since this is the core of what a phone does, we'll call this subsystem, the **model**.

The *interface* for very early phones was a horn-shaped microphone mounted on the front of the device and a speaker held to your ear. Later the interface evolved so that the speaker and microphone were integrated into a handset. And "dialing a number" evolved from speaking to an operator at a switchboard, to a dial-rotor and later to a keypad. Though there is no screen, the speaker and microphone are an interface, so it's analogous to the **view**.

The **controller** in this example would be the other parts needed to complete the device: the casing, connections between the interface elements and the core phone components.

In the 70s, the core phone functionality (the **model**) was integrated into a new device. This one coupled the line communication hardware to an interface that modulated binary signals into ones suited for the telephone system (that is, ones in the human auditory range). This new device was the dialup-modem.
</MOTION>

The analogy is telling in that it is the core-phone (or model) functionality that endured largely unchanged the longest. Likewise in software, the view(or interface) sub-system

tends to be subject to the most change: driven by user feedback and evolving interface styles and standards.

In the next video we'll look at some examples of where MVC appears in Cocoa and XCode.