# Architectural Styles

## Objectives

- To review the definitions of software architecture and architectural style
- To present several important architectural styles, including
  - Layered
  - Pipe-and-Filter
  - Shared-Data
  - Event-Driven
  - Model-View-Controller
  - Hybrid

## Topics

- Software architecture and architectural styles
- Layered style
- Pipe-and-Filter style
- Shared-Data style
- Event-Driven style
- Model-View-Controller style
- Hybrid architectures

## Software Architecture

A **software architecture** is the structure of a program comprised by its major constituents, their responsibilities and properties, and the relationships and interactions between them.

## Software Architecture Issues

- Gross decomposition of system function
- Allocation of functions to components
- Component interfaces
- Communication and interaction among components
- Component and system properties: capacity, throughput, constraints
- Reuse of common design styles

## Architectural Styles

An **architectural style** is a paradigm of program or system constituent types and their interactions.

## Layered Style Architectures

- The program is partitioned into an array of layers or groups.
- Layers use the services of the layer or layers below and provide services to the layer or layers above.
- The Layered style is among the most widely used of all architectural styles.

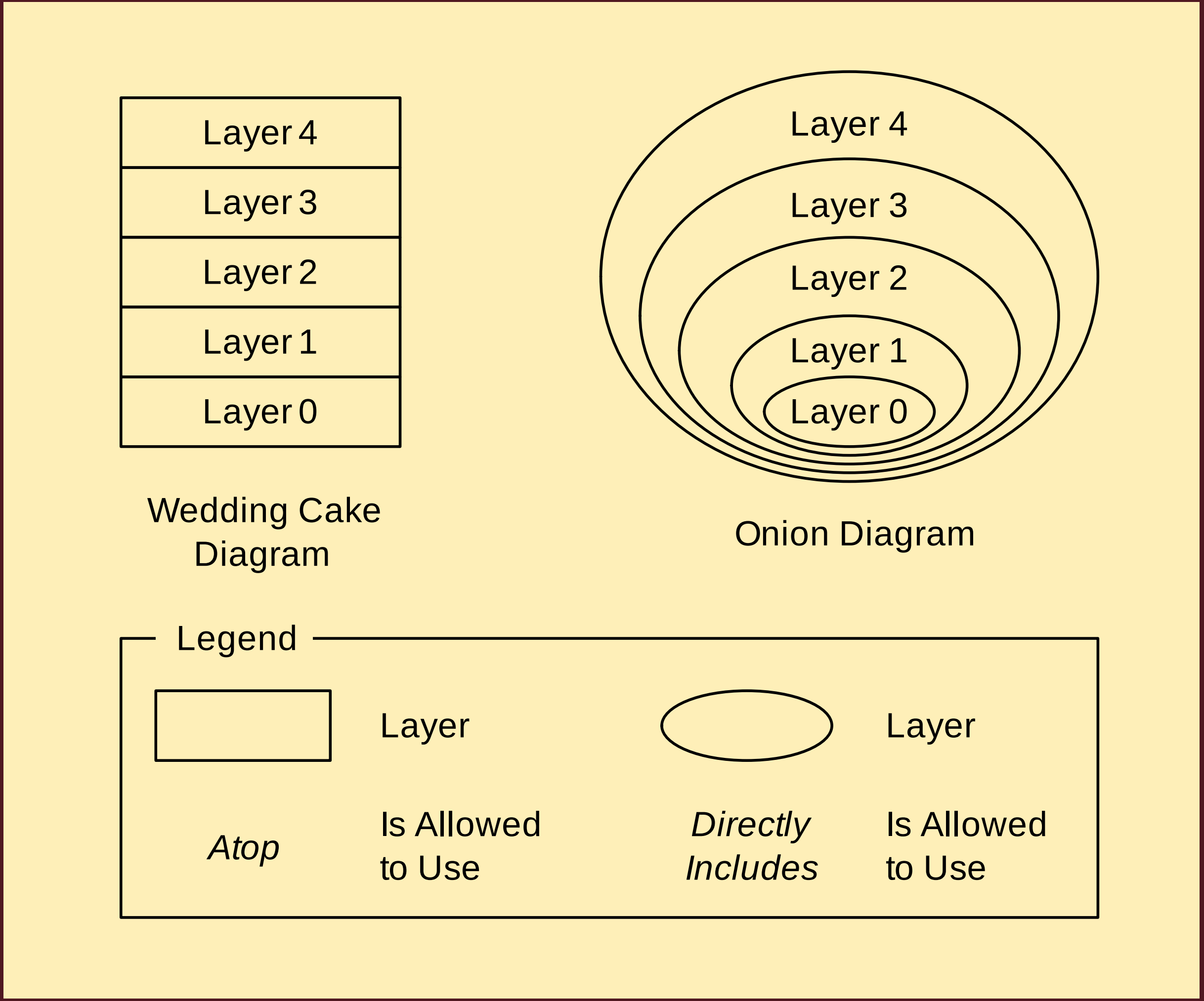
## Uses and Invokes

- Module A **uses** module B if a correct version of B must be present for A to execute correctly.
- Module A **calls** or **invokes** module B if A triggers execution of B.
- Note that
  - A module may use but not invoke another
  - A module may invoke but not use another
  - A module may both use and invoke another
  - A module may neither use nor invoke another

## Layer Constraints

- *Static structure*—The software is partitioned into layers that each provide a cohesive set of services with a well-defined interface.
- *Dynamic structure*—Each layer is allowed to use only the layer directly below it (**Strict Layered** style) or the all the layers below it (**Relaxed Layered** style).

# Representing Layers



## Forming Layers

- Levels of abstraction
  - Example: Network communication layers
- Virtual machines
  - Examples: Operating systems, interpreters
- Information hiding, decoupling, etc
  - Examples: User interface layers, virtual device layers

## Layered Style Advantages

- Layers are highly cohesive and promote information hiding.
- Layers are not strongly coupled to layers above them, reducing overall coupling.
- Layers help decompose programs, reducing complexity.
- Layers are easy to alter or fix by replacing entire layers, and easy to enhance by adding functionality to a layer.
- Layers are usually easy to reuse.

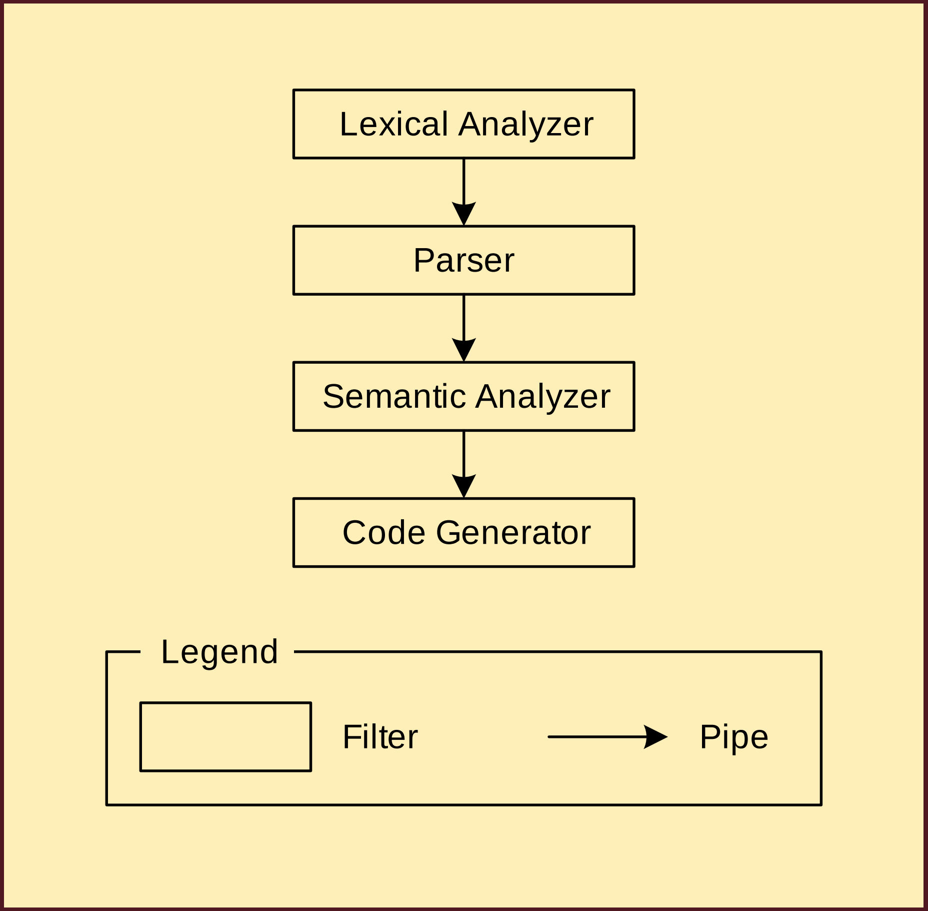
## Layered Style Disadvantages

- Passing everything through many layers can complicate systems and damage performance.
- Debugging through multiple layers can be difficult.
- Getting the layers right can be difficult.
- Layer constraints may have to be violated to achieve unforeseen functionality.

## Pipe-and-Filter Style

- A **filter** is a program component that transforms an input stream to an output stream.
- A **pipe** is conduit for a stream.
- The **Pipe-and-Filter** style is a dynamic model in which program components are filters connected by pipes.

## Pipe-and-Filter Example



## Pipe-and-Filter Characteristics

- Pipes are isolated and usually only communicate through data streams, so they are easy to write, test, reuse, and replace.
- Filters may execute concurrently.
  - Requires pipes to synchronize filters
- Pipe-and-filter topologies should be acyclic graphs.
  - Avoids timing and deadlock issues
- A simple linear arrangement is a *pipeline*.

## Pipe-and-Filter Advantages

- Filters can be modified and replaced easily.
- Filters can be rearranged with little effort, making it easy to develop similar programs.
- Filters are highly reusable.
- Concurrency is supported and is relatively easy to implement.

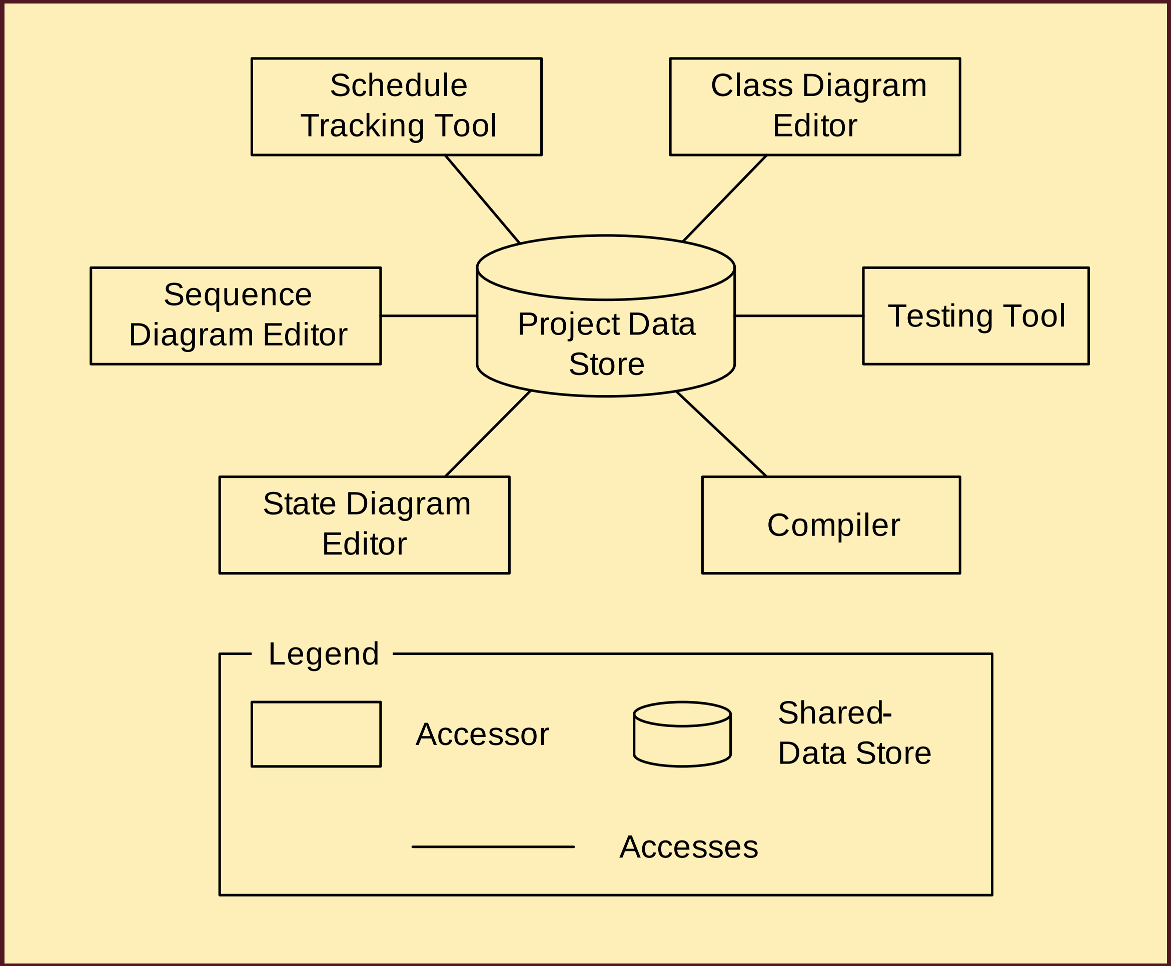
## Pipe-and-Filter Disadvantages

- Filters communicate only through pipes, which makes it difficult to coordinate them.
- Filters usually work on simple data streams, which may result in wasted data conversion effort.
- Error handling is difficult.
- Gains from concurrency may be illusory.

## Shared-Data Style

- One or more *shared-data stores* are used by one or more *shared-data accessors* that communicate solely through the shared-data stores.
- Two variants:
  - **Blackboard style**—The shared-data stores activate the accessors when the stores change.
  - **Repository style**—The shared-data stores are passive and manipulated by the accessors.
- This is a dynamic model only.

# Shared-Data Style Example



## Shared-Data Style Advantages

- Shared-data accessors communicate only through the shared-data store, so they are easy to change, replace, remove, or add to.
- Accessor independence increases robustness and fault tolerance.
- Placing all data in the shared-data store makes it easier to secure and control.

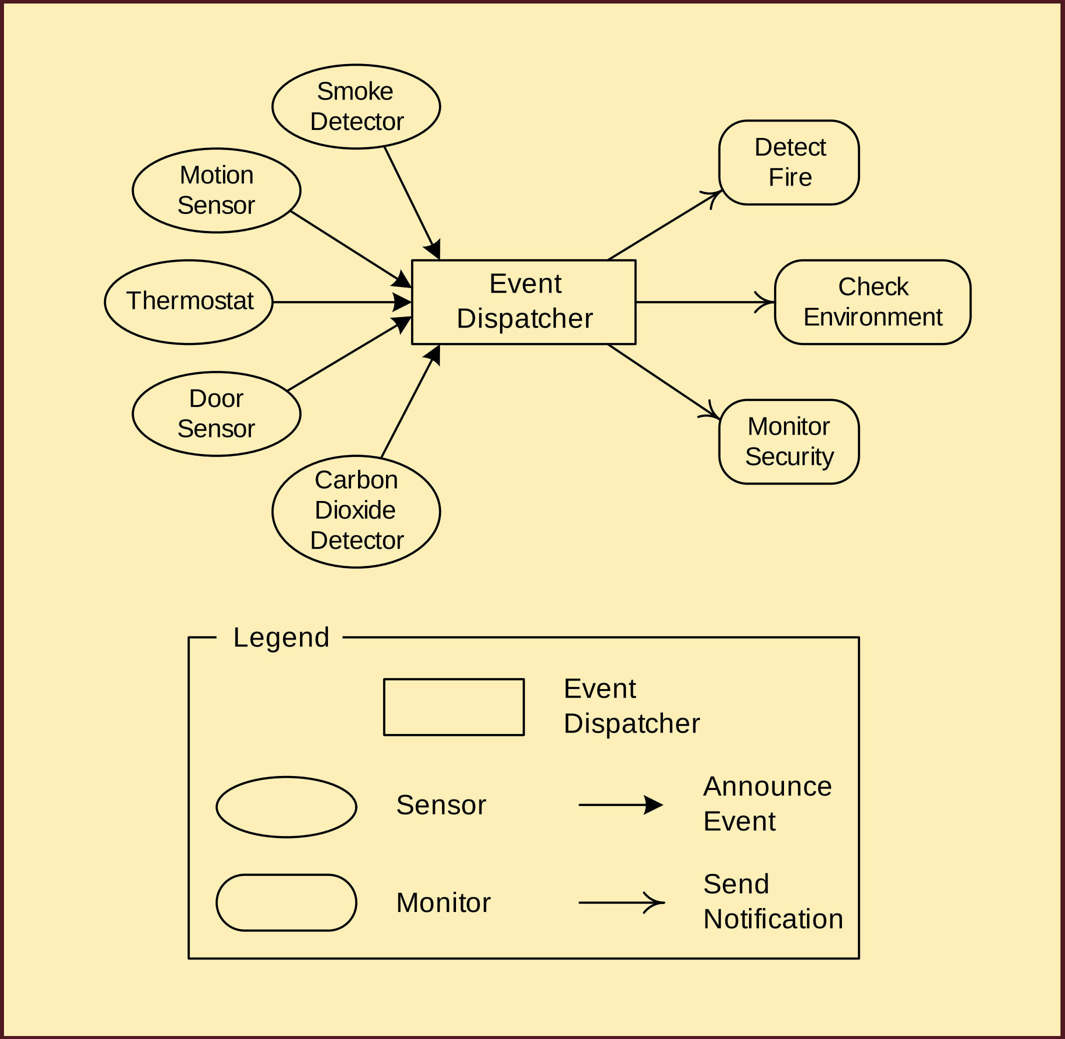
## Shared-Data Style Disadvantages

- Forcing all data through the shared-data store may degrade performance.
- If the shared-data store fails, the entire program is crippled.

## Event-Driven Style

- Also called the **Implicit Invocation** style
- An **event** is any noteworthy occurrence.
- An *event dispatcher* mediates between components that announce and are notified of events.
- This is a dynamic model only.

# Event-Driven Style Example



## Stylistic Variations

- Events may be notifications or they may carry data.
- Events may have constraints honored by the dispatcher, or the dispatcher may manipulate events.
- Events may be dispatched synchronously or asynchronously.
- Event registration may be constrained in various ways.

## Event-Driven Style Advantages

- It is easy to add or remove components.
- Components are decoupled, so they are highly reusable, changeable, and replaceable.
- Systems built with this style are robust and fault tolerant.

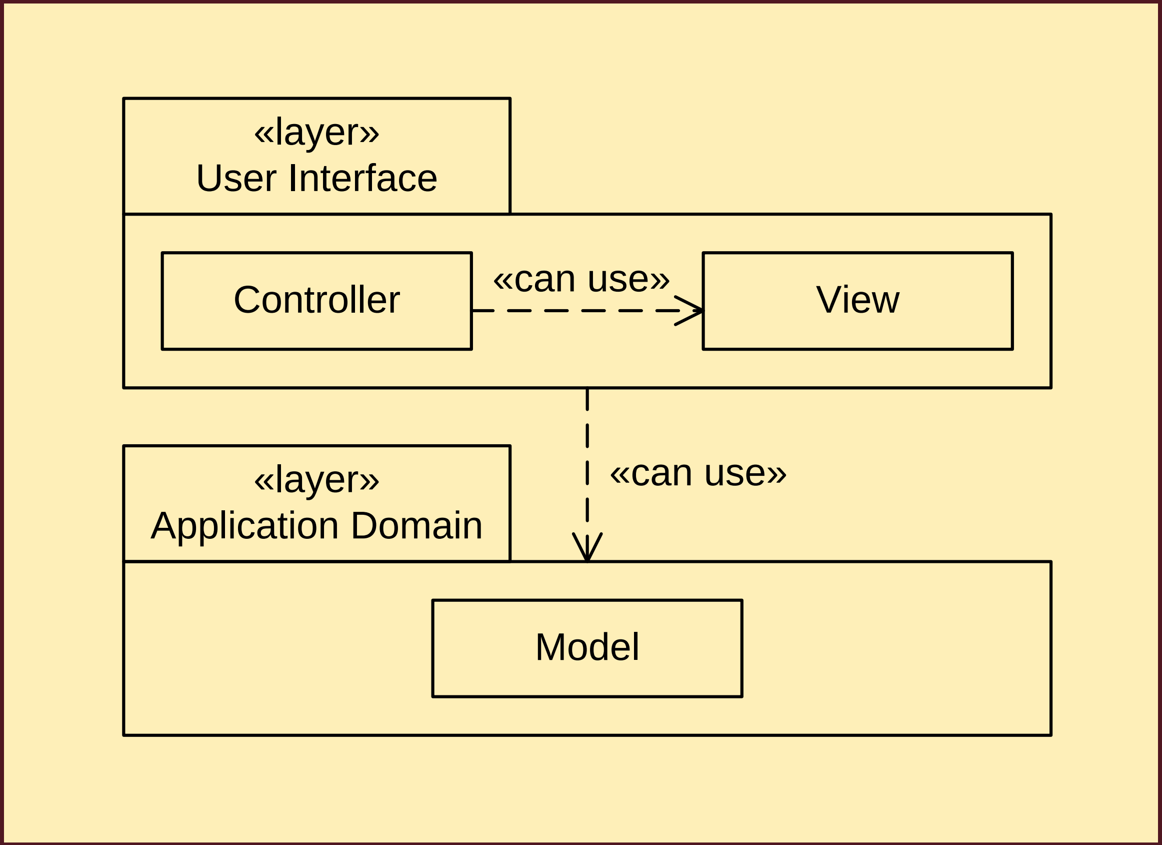
## Event-Driven Style Disadvantages

- Component interaction may be awkward when mediated by the event dispatcher.
- There are no guarantees about event sequencing or timing, which may make it difficult to write correct programs.
- Event traffic tends to be highly variable, which may make it difficult to achieve performance goals.

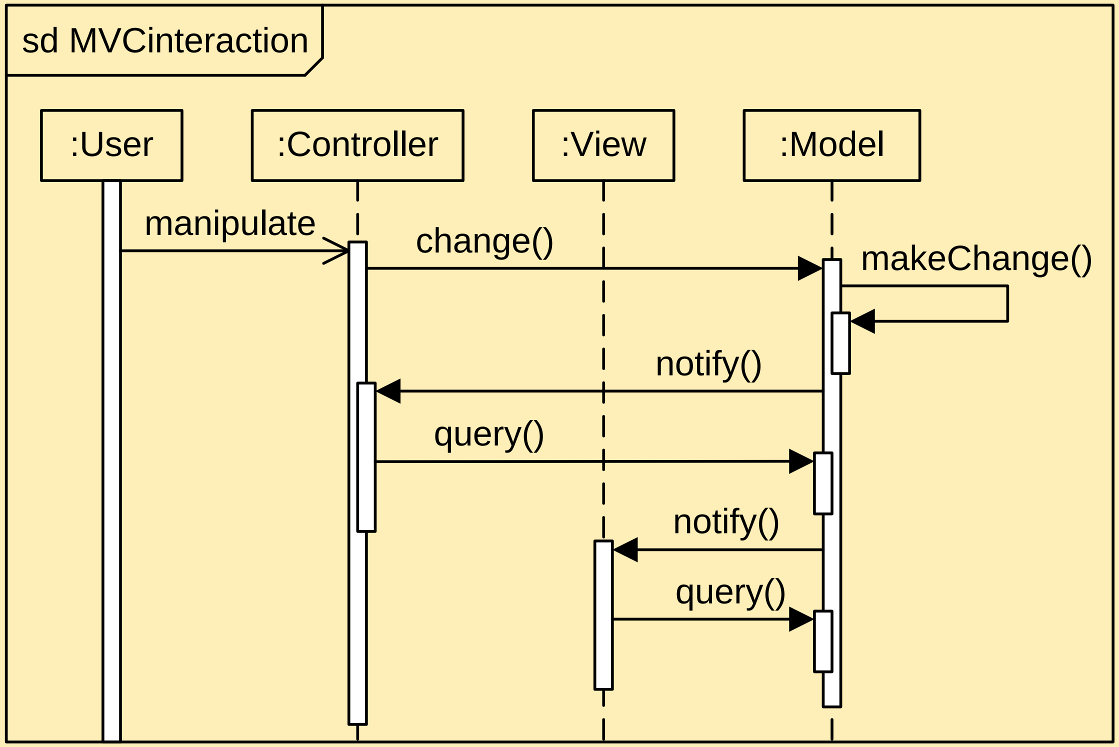
## Model-View-Controller (MVC) Style

- This style models how to set up the relationships between user interface and problem-domain components.
- *Model*—A problem-domain component with data and operations for achieving program goals independent of the user interface
- *View*—A data display component
- *Controller*—A component that receives and acts on user input

# MVC Static Structure



# MVC Behavior



## MVC Advantages

- Views and controllers can be added, removed, or changed without disturbing the model.
- Views can be added or changed during execution.
- User interface components can be changed, even at runtime.

## MVC Disadvantages

- Views and controller are often hard to separate.
- Frequent updates may slow data display and degrade user interface performance.
- The MVC style makes user interface components highly dependent on model components.

## Hybrid Architectures

Most systems of any size include several architectural styles, often at different levels of abstraction.

- An overall a system may have a Layered style, but the one layer may use the Event-Driven style, and another the Shared-Data style.
- An overall system may have a Pipe-and-Filter style, but the individual filters may have Layered styles.

## Summary 1

- In the Layered style program components are partitioned into layers and each layer is constrained to use only the layer or layers beneath it.
- In the Pipe-and-Filter style components are filters connected by pipes.
- In the Shared-Data style program components are modeled as one or more shared-data stores manipulated by one or more shared-data accessors.

## Summary 2

- In the Event-Driven style program components register with an event dispatcher than accepts announcement of events and notifies interested components that events have occurred.
- In the Model-View-Controller style user interface view and controller components can use problem-domain model components that notify them when they change.

## Summary 3

- Architectures are usually composed of several styles, often at different levels of abstraction, that is, architectures are often hybrids of styles.