

Flex Applications' Architecture

Why architecture is important?



Handling Complexity

- **choosing the right *data structures***
- **developing *algorithms***
- **applying the concept of *separation of concerns***
- ***abstraction***
- **emergence of *design patterns***
- ***best practices***

*the ideas of **abstraction** and **separation of concerns** are central to software architectures*

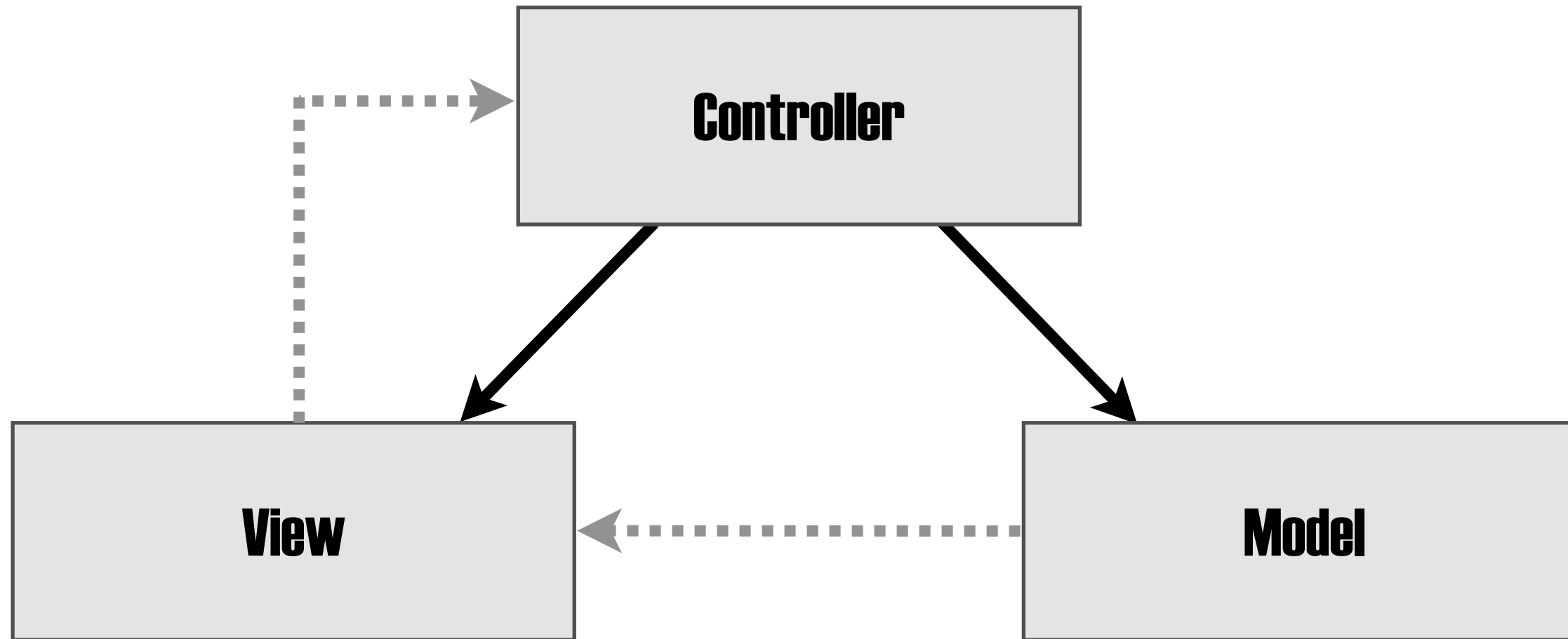
What's the goal?

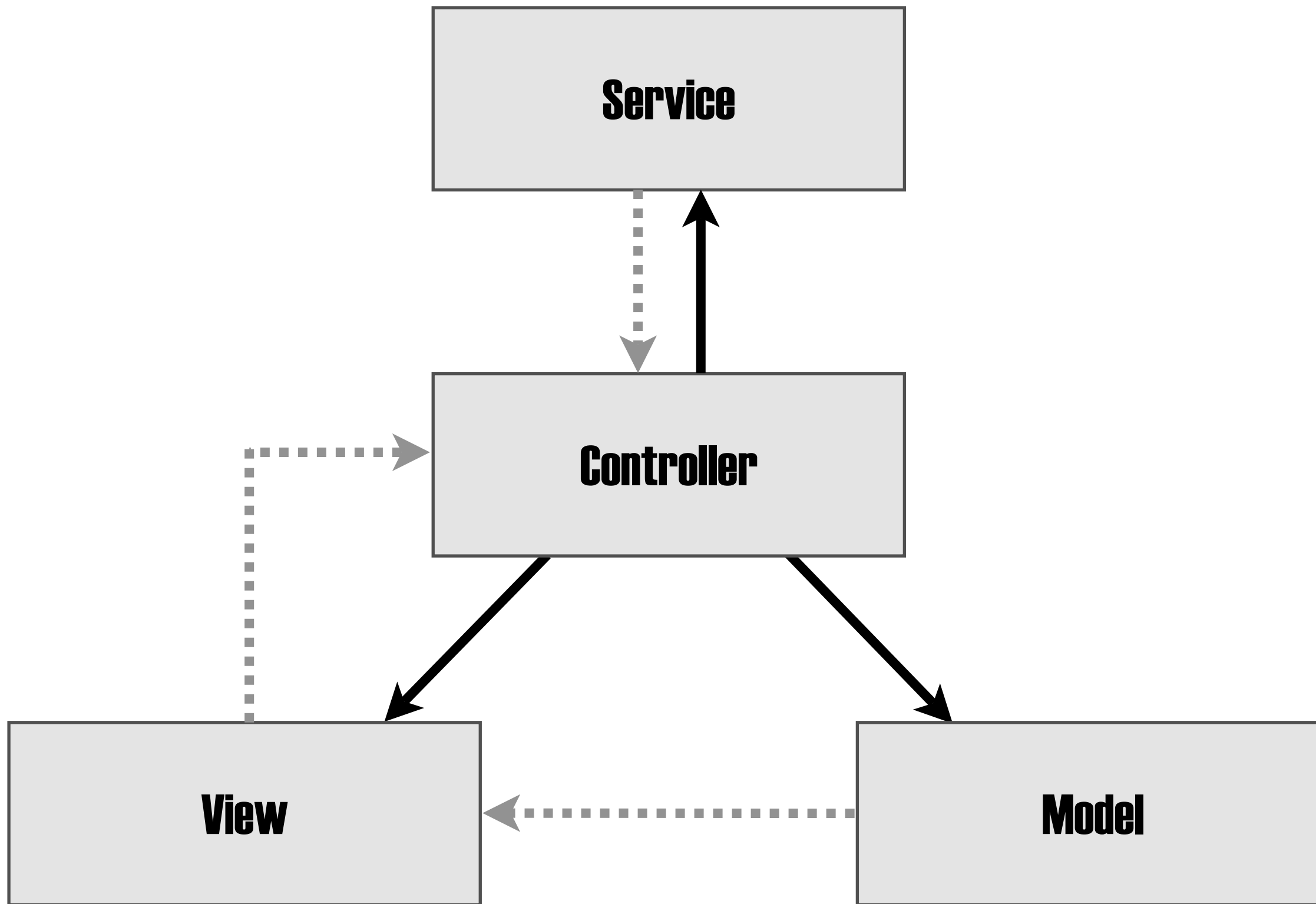
The ultimate goal is to cover a set of quality attributes:

- **extensibility**
- **maintainability**
- **scalability**
- **usability**
- ...

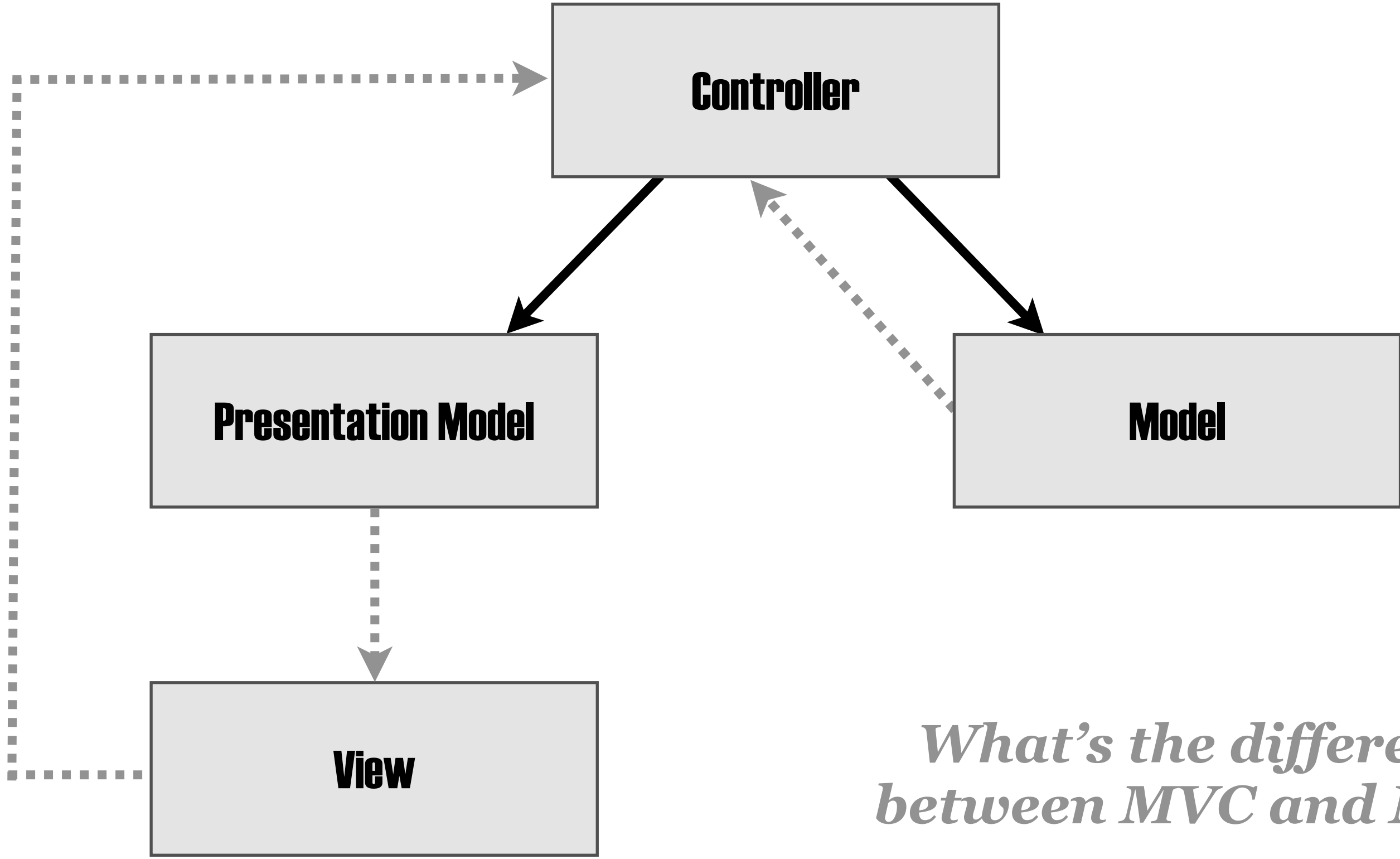
there are many more other '-ilities', but we'll focus mostly on these

Model-View-Controller





Model-View-Presenter



What's the difference between MVC and MVP?

MVC

- the Model is both presenter model and a data model
- the Controller can control both the Model and the View

MVP

- the model is divided into a Presenter Model and a Data Model
- the Controller can control the View only indirectly through changes to the Presenter Model

What's if we need to coordinate the work of several views?

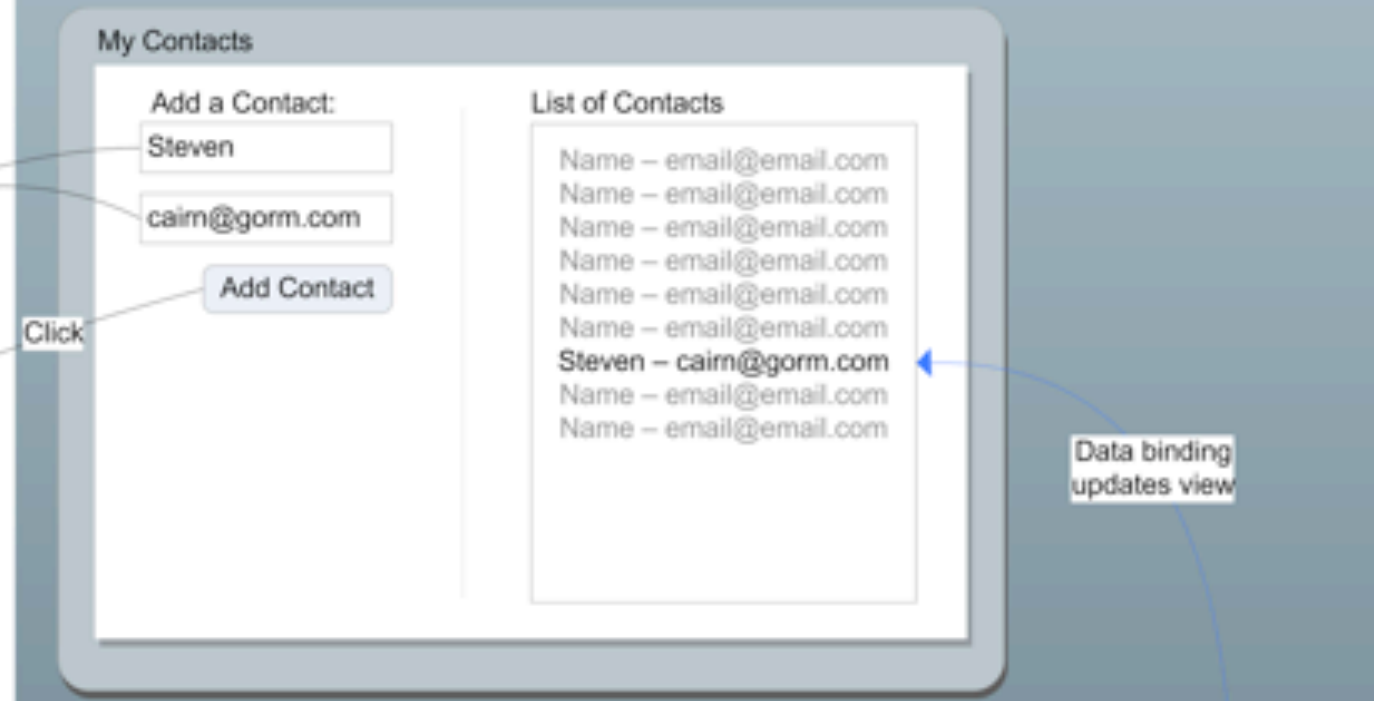
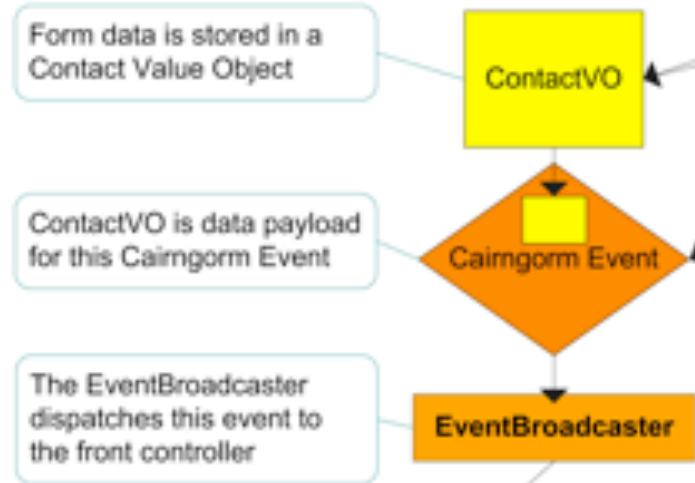
Mediator

- mediators encapsulate how a set of objects interact
- mediators can control the coordination between several views - sometimes the code behind is used for such coordination (which results in coupling the View with use-case handling and reacting to state)
- in some cases the mediators are used in the form of a translator responsible for translating the application/interaction logic into business logic

Architectural Frameworks

Cairngorm

View

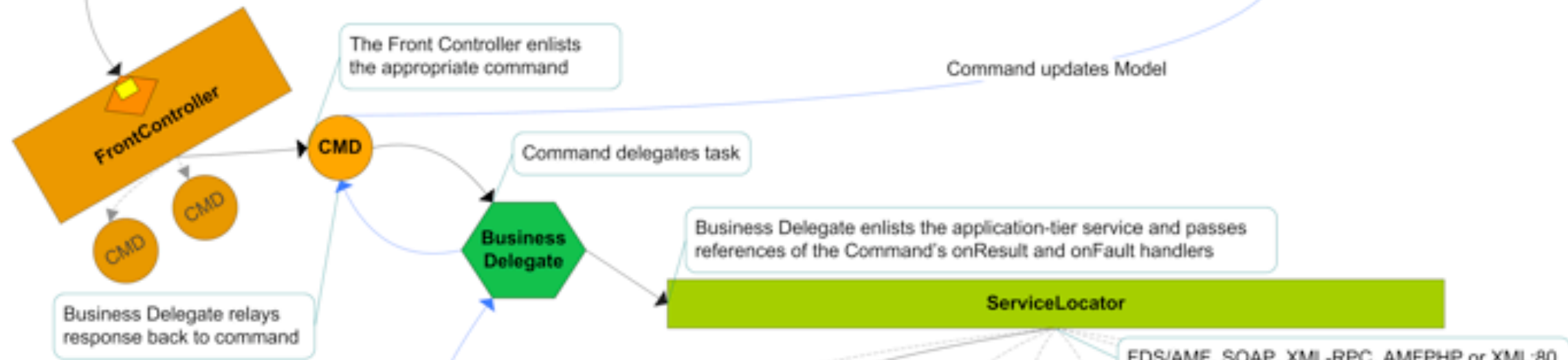


Model

Event and Data Payload are passed together to Front Controller



Controller



Application Tier

Response is sent back to Business Delegate



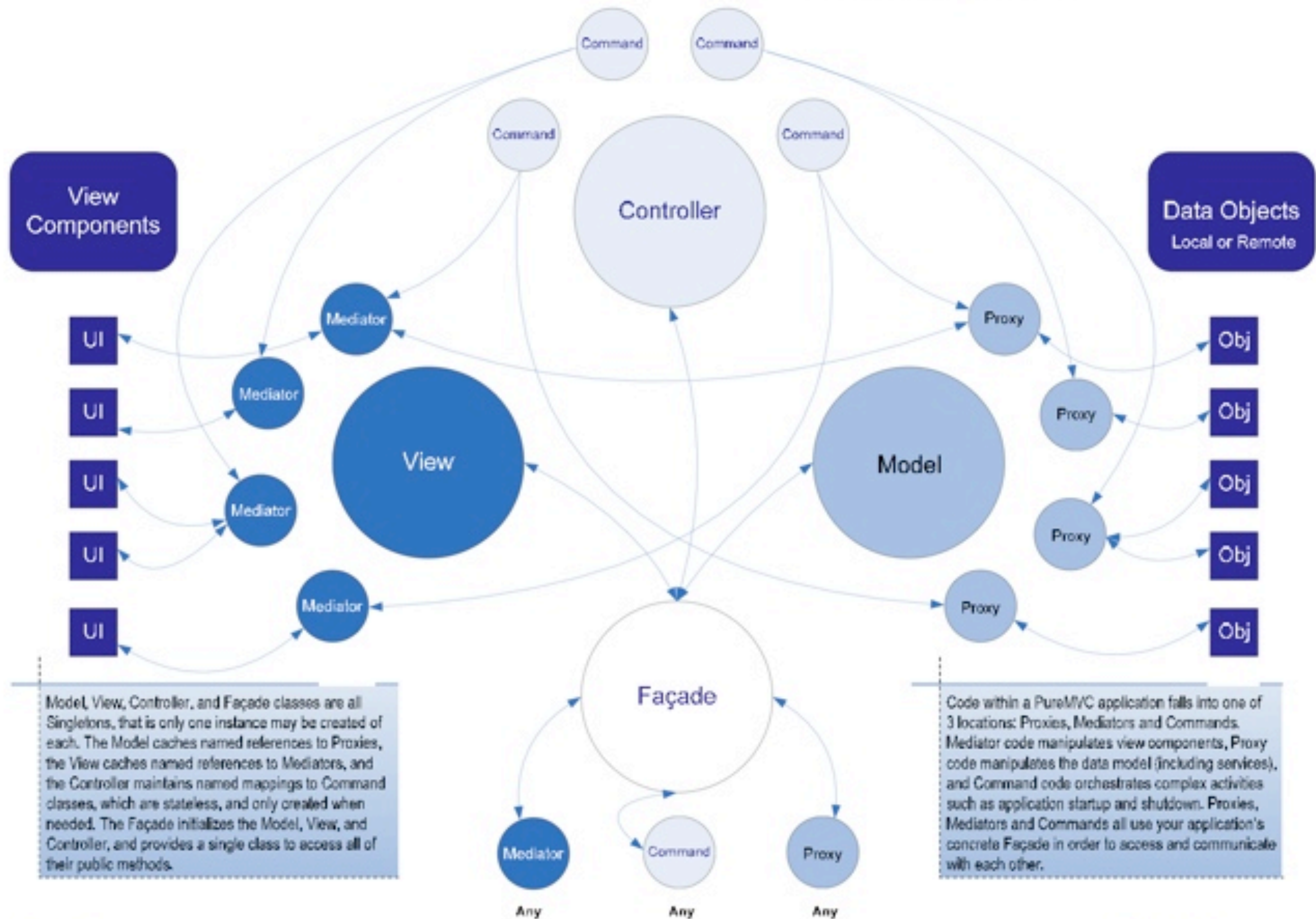
FDS/AMF, SOAP, XML-RPC, AMFPHP or XML:80

Data binding updates view

Cairngorm Drawbacks

- **controller logic spreads into a large set of commands that very often differ only trivially from each other, thus we end up with lots of repetitive code**
- **controller logic is too far from the view**
- **model can get easily very big and complicated**
- **the lack of Mediator pattern leads to coupling reusable components with dependencies on Cairngorm (e.g. dispatching Cairngorm events, rather than abstract events)**

PureMVC



Model, View, Controller, and Façade classes are all Singletons, that is only one instance may be created of each. The Model caches named references to Proxies, the View caches named references to Mediators, and the Controller maintains named mappings to Command classes, which are stateless, and only created when needed. The Façade initializes the Model, View, and Controller, and provides a single class to access all of their public methods.

Code within a PureMVC application falls into one of 3 locations: Proxies, Mediators and Commands. Mediator code manipulates view components, Proxy code manipulates the data model (including services), and Command code orchestrates complex activities such as application startup and shutdown. Proxies, Mediators and Commands all use your application's concrete Façade in order to access and communicate with each other.

PureMVC Drawbacks

- **controller logic is too far from the view - it's hard to handle more complex interactions**
- **the model is too far from the view - since the framework is not designed for Flex, it is difficult to benefit from the data binding mechanism in Flex**
- **same as in Cairngorm we end up with a big monolithic front controller - no way to dedicate separate controllers for logically-related concerns**

Cairngorm

- **easy to streamline development**
- **no way to have multiple Cairngorm instances**
- **model is very monolithic**
- **introduces a ServicesLocator**

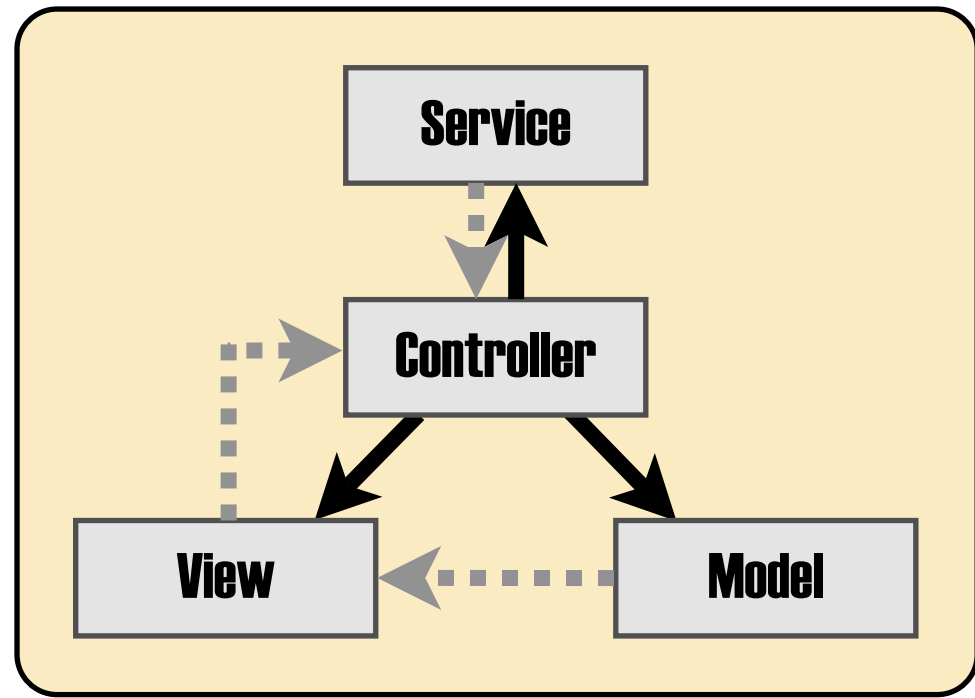
PureMVC

- **easy to streamline development**
- **supports multiple instances of the PureMVC core**
- **model is distributed between proxies**

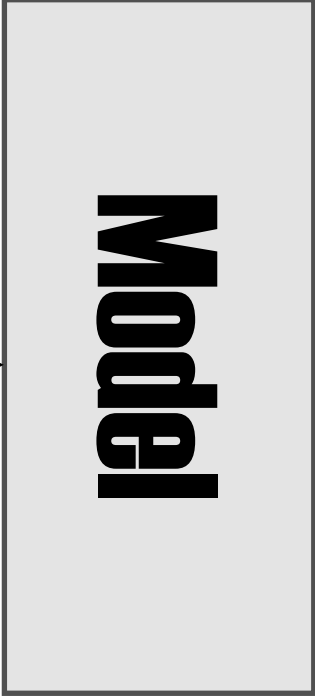
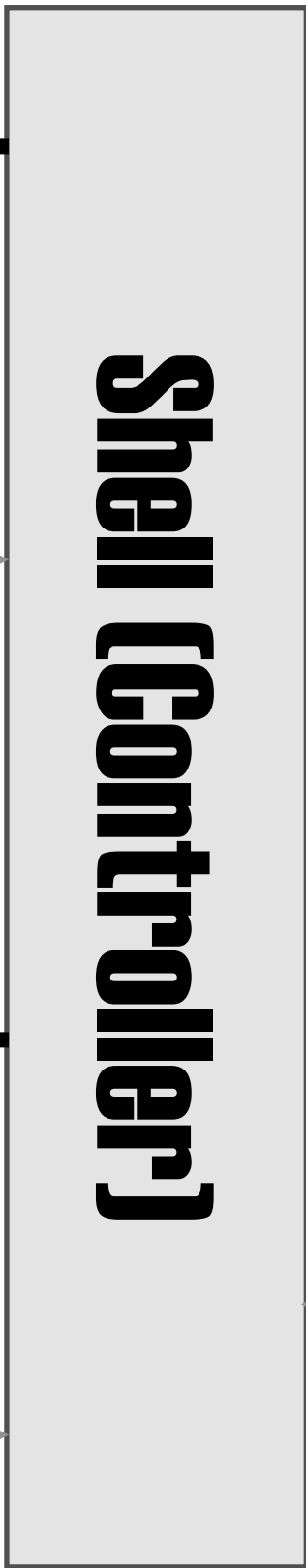
Both frameworks represent Macro patterns - limited level of granularity

A more granular approach

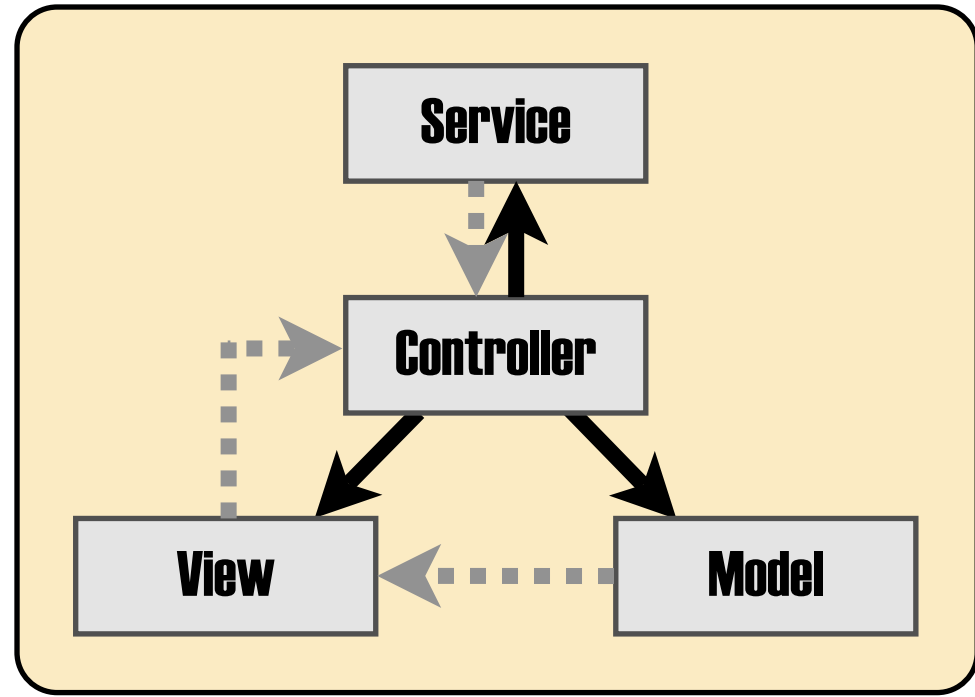
Module 1



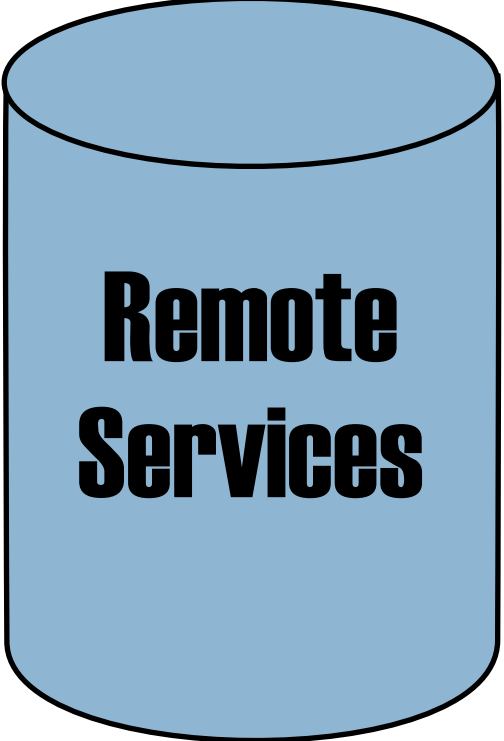
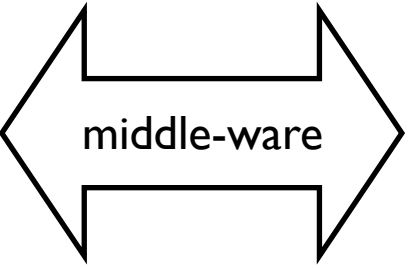
create
inject dependencies
dispatch events



Module 2



create
inject dependencies
dispatch events



A typical MVCS with IoC

```
<model:StockModel id="model"/>  
  
<control:Controller id="controller"  
  model="{model}"  
  view="{view}"  
  services="{services}"/>  
  
<view:StockDisplay id="view"  
  model="{model}"  
  buy="controller.buy(event.stocks)"/>  
  
<service:Services id="services"/>
```

*you can use data binding
for dependency injection*

Event Propagation

- **views should dispatch only view specific events**
- **view controllers handle such events and decide whether to translate them to module specific events**
- **module specific events are handled by module controllers, which decide whether to translate them to application specific events**

Component-level Architecture

What's the goal?

- **the goal is to layer the components into relatively small and specialized layers of loosely bound reference implementations**
- **the layers can be UI elements, skinning, layout, data binding, view controllers, etc.**

OpenFlux Sample

```
<flux:Button label="Button Example"
  left="20" top="20">
  <flux:view>
    <flux:CheckBoxView/>
  </flux:view>
  <flux:controller>
    <flux:ButtonController selectable="true"/>
  </flux:controller>
</flux:Button>
```

*the model is the
component itself*

```
<mx:Style source="button_styles.css"/>
<flux:Button label="Button Example"
  left="20" top="20" styleName="checkbox"/>
```

```
.checkbox {
  selectable: true;
  view: ClassReference("com.openflux.views.CheckBoxView");
}
```

Summary

- **why architecture is important**
- **MVC, MVCS, MVP**
- **architectural frameworks - Cairngorm & PureMVC**
- **micro MVCS**
- **component level architecture**
- **OpenFlux**