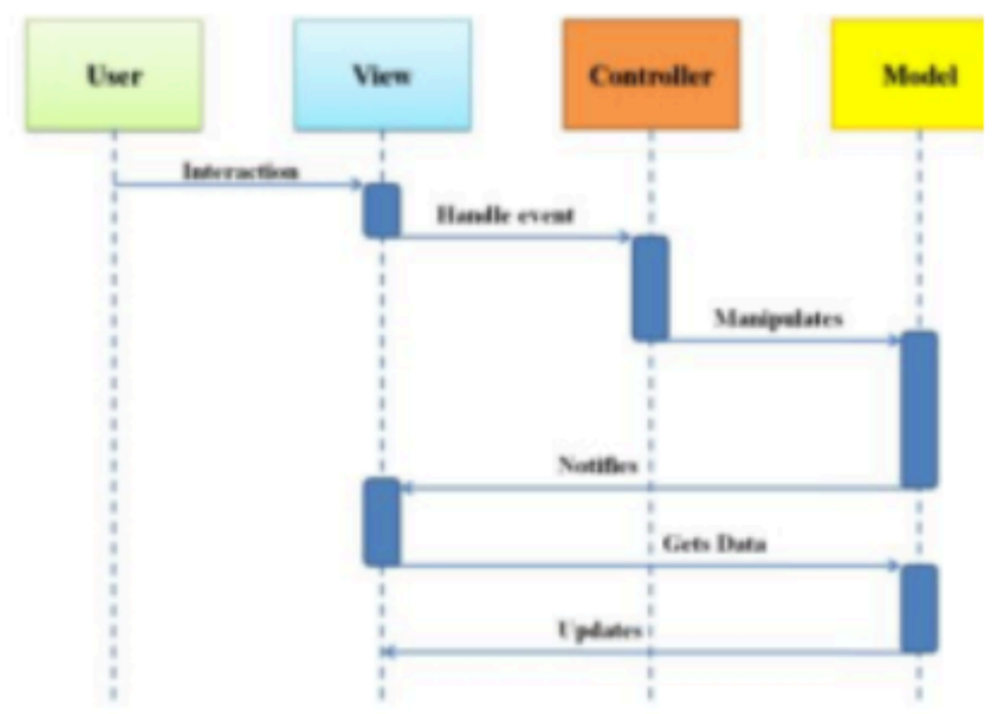
UI Patterns

UI Patterns



MVC & MVVM are patterns that layout potential approaches to GUI application development.

Agenda

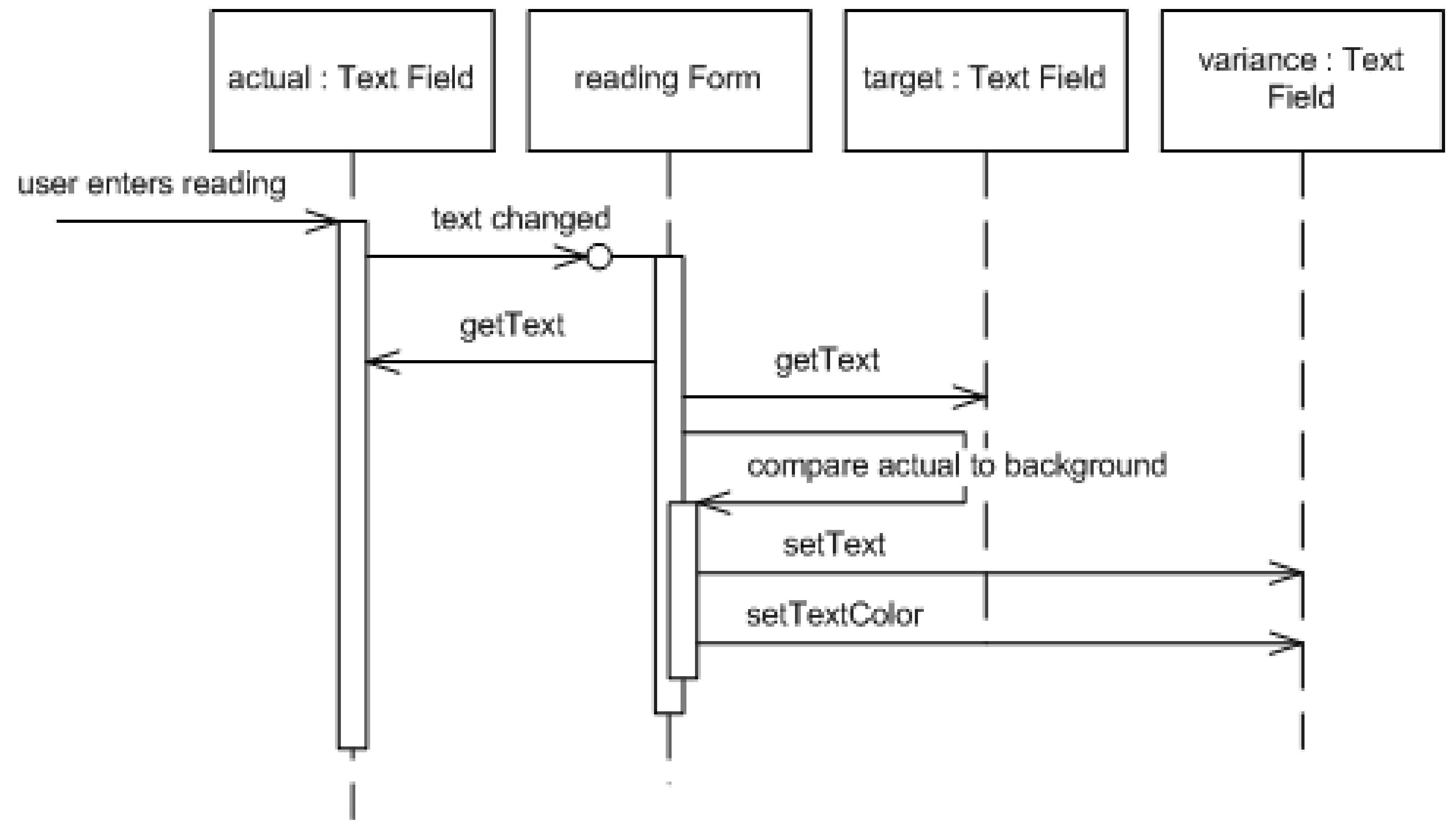
- GUI Patterns
- Model View Controller - MVC
- Model View View-Model - MVVM

Graphical User Interfaces

- User Interface development - can be notoriously complex.
- Inherent complexity: the GUI component set is at varying levels of abstraction with sophisticated event mechanisms:
  - Controls
  - Containers/Dials/Widgets
  - Panels/Windows
  - Menus/Buttons/Dropdowns
- Accidental complexity: domain logic can easily become hopelessly intermingled with the GUI specific logic.

GUI Events

- A significant source of complexity
- Fine-grained events
  - Mouse entered, exited
  - Mouse pressed
  - Radio button pressed, armed, rollover
- Coarse-grained events:
  - Radio button selected
  - Action performed
  - Domain property changed
- Managing the flow of these events requires careful consideration if design coherence is to be preserved.



GUI Patterns

- Reusable designs that can be realised with different toolkits:
  - Model View Controller (MVC)
  - Model View View Model (MVVM)
- Other patterns (http://martinfowler.com/eaaDev/)
  - Notification
  - Supervising Controller
  - Model View Presenter (MVP)
  - Passive View
  - Presentation Model
  - Event Aggregator
  - Window Driver
  - Flow Synchronization
  - Observer Synchronization
  - Presentation Chooser
  - Autonomous View

In particular, read
http://martinfowler.com/eaaDev/uiArchs.html
for background to these patterns

Key Principle: Separation of Concerns

WIKIPEDIA
The Free Encyclopedia

“In computer science, separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. A concern can be as general as the details of the hardware the code is being optimized for, or as specific as the name of a class to instantiate. A program that embodies SoC well is called a modular program.”

https://en.wikipedia.org/wiki/Separation_of_concerns

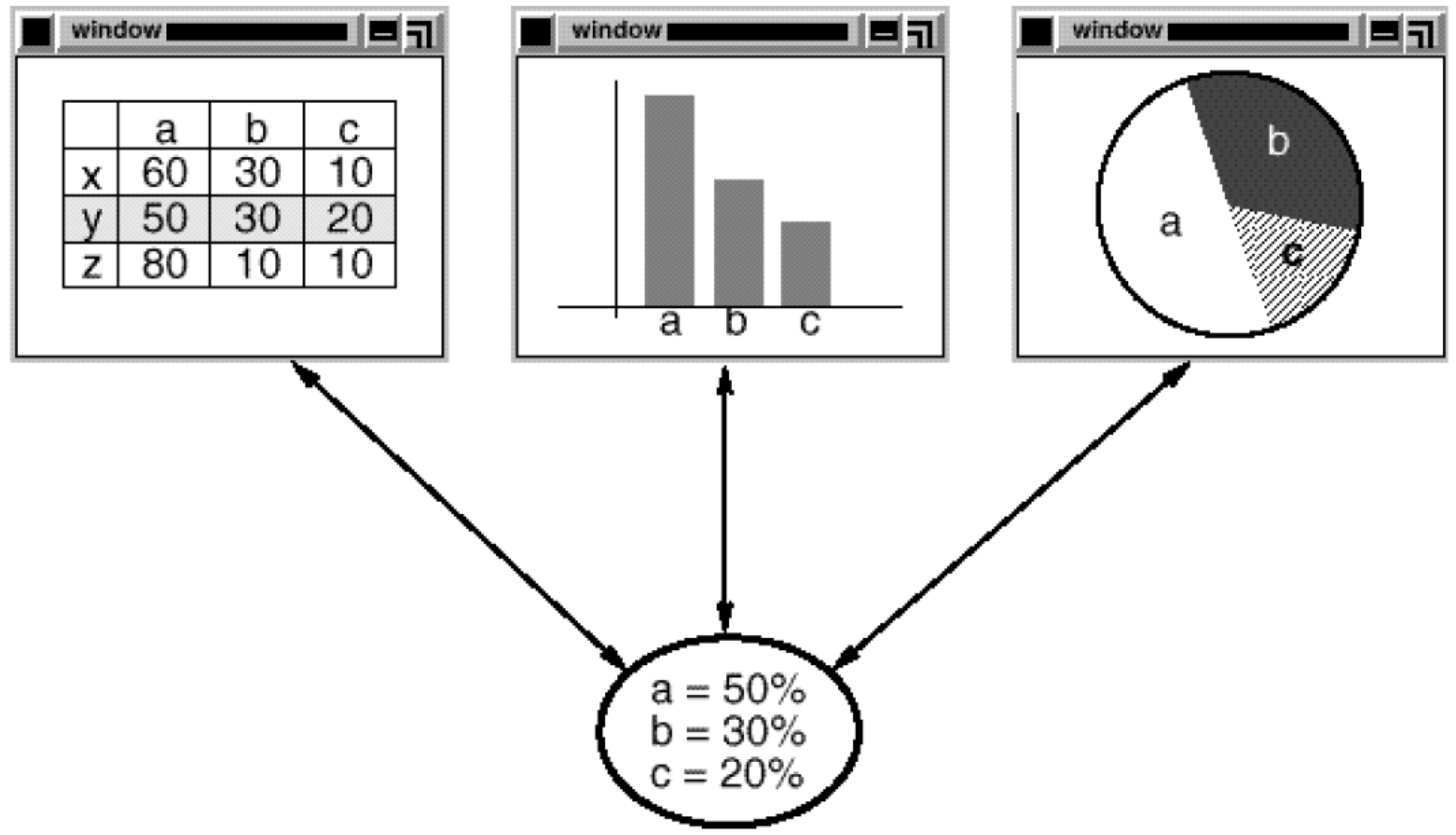
Model View Controller

- The Model/View/Controller (MVC) triad of classes is used to build user interfaces in Smalltalk-80.
- MVC consists of three kinds of objects:
  - Model is the application object
  - View is its screen presentation
  - Controller defines the way the user interface reacts to user input
- Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse

Synchronization

- MVC synchronizes views and models via Observer Synchronization.
  - A view must ensure that its appearance reflects the state of the model.
  - Whenever the model's data changes, the model notifies views that depend on it.
  - In response, each view gets an opportunity to update itself.
- This approach allows multiple views to be attached views to a model to provide different presentations.

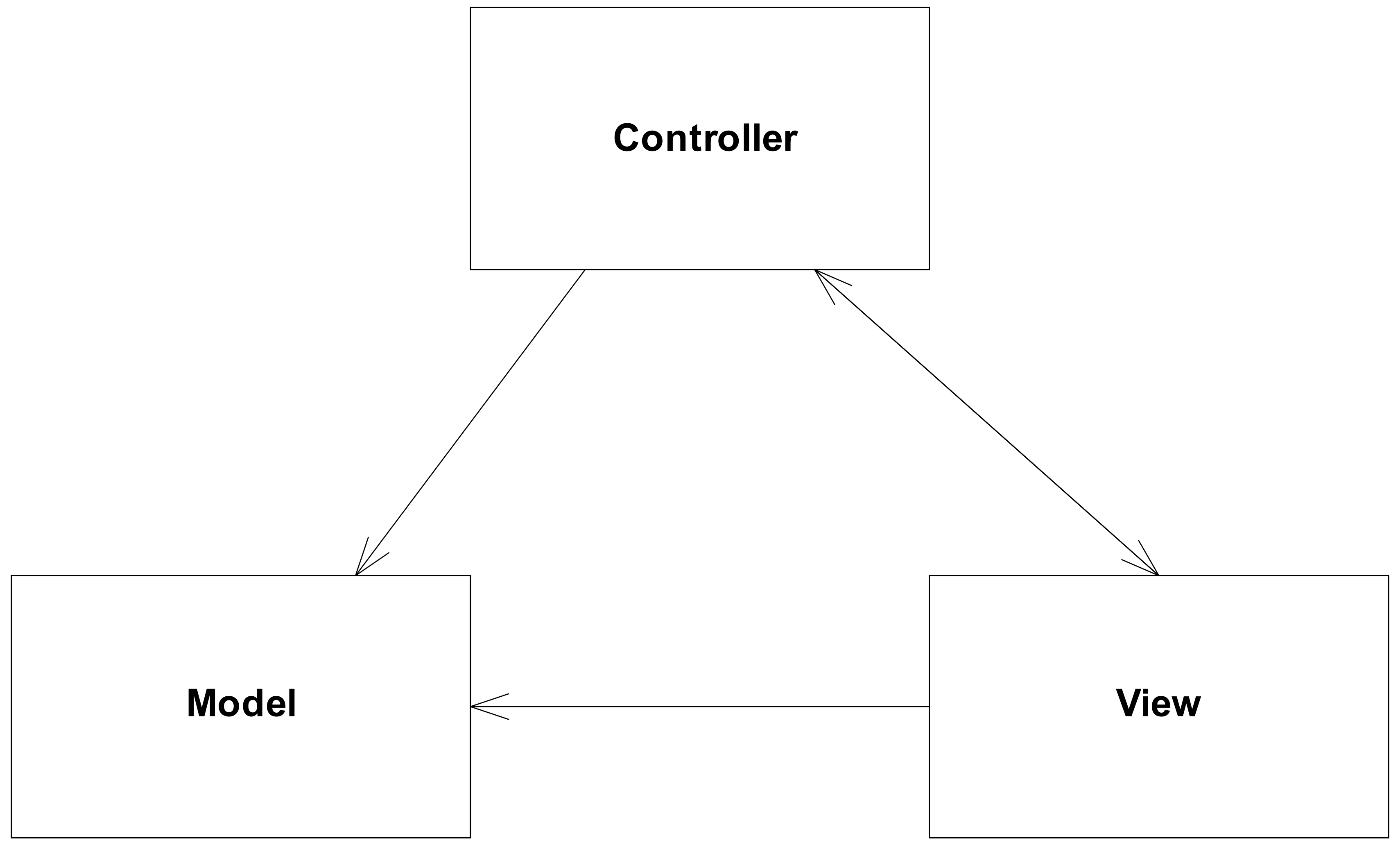
View / Model



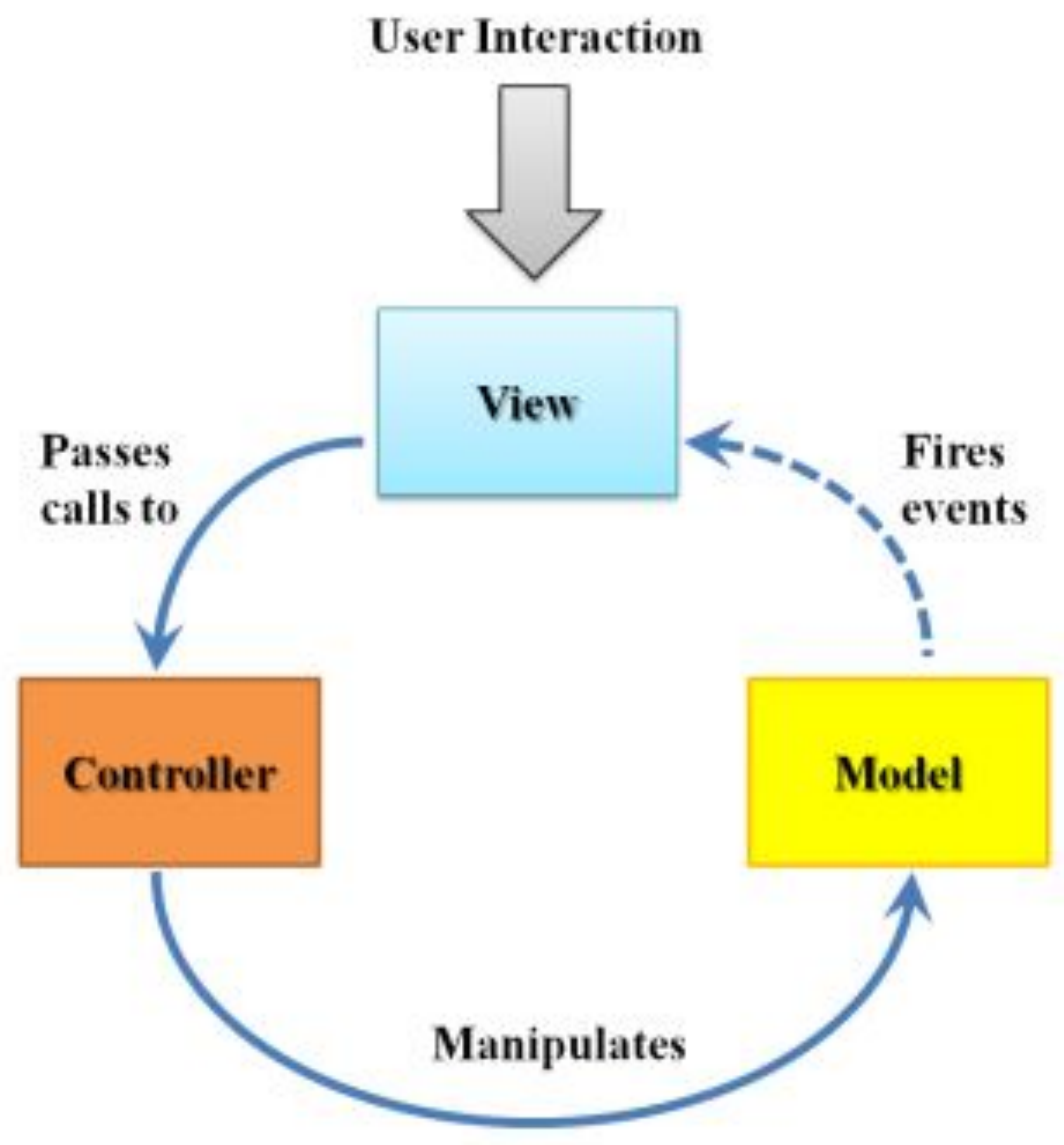
	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Controller

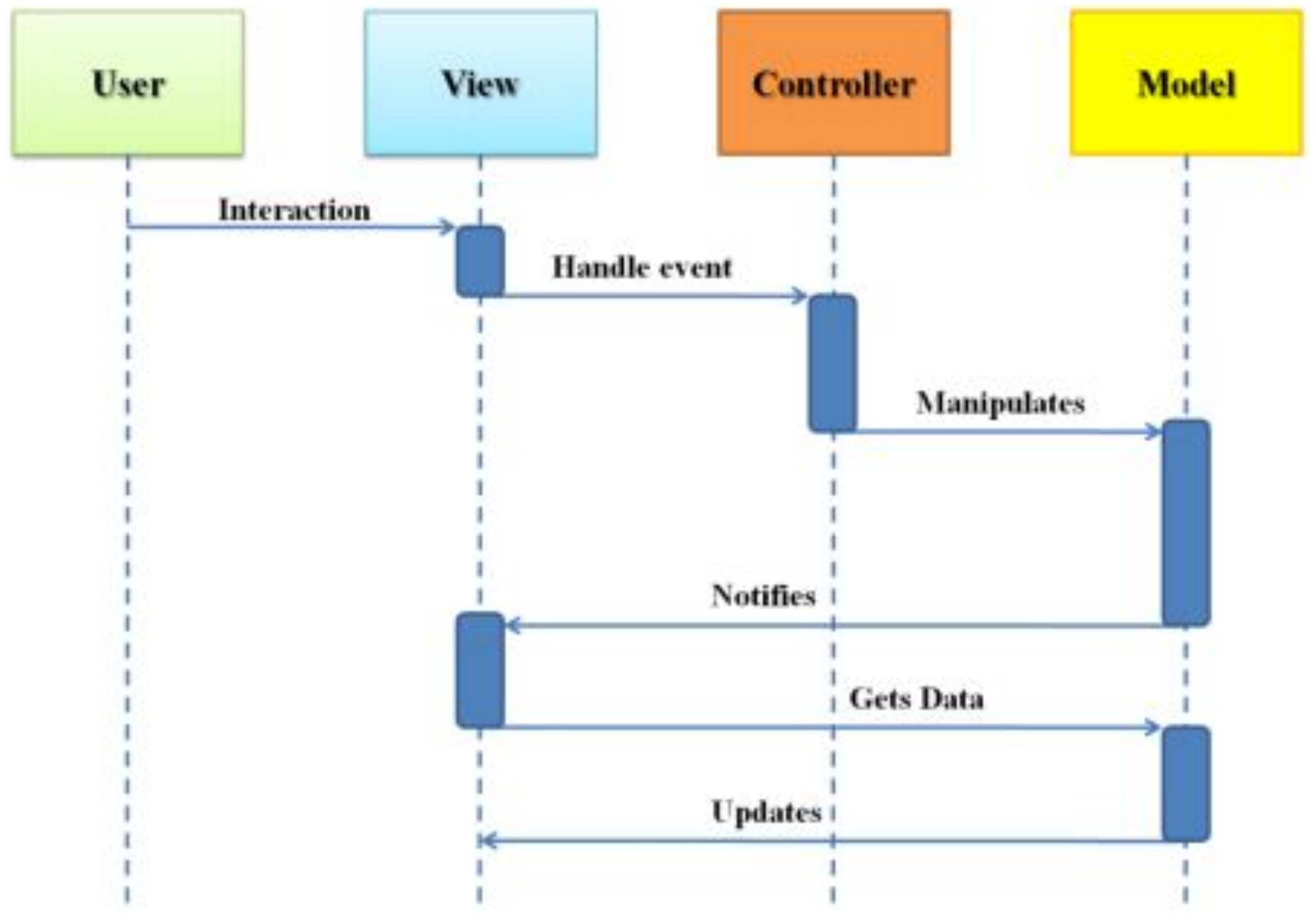
- MVC encapsulates response and model update mechanisms in a Controller object.
- The Controller is the "glue" between the Model and the View.
- The View renders model updates on the screen, but is not permitted to modify the model.
- The View forwards events to the controller
- The Controller does not have access to the screen but can modify the model.



MVC Sequence Diagrams



Model-View-Controller



Sequence Diagram of MVC

Potential Advantages

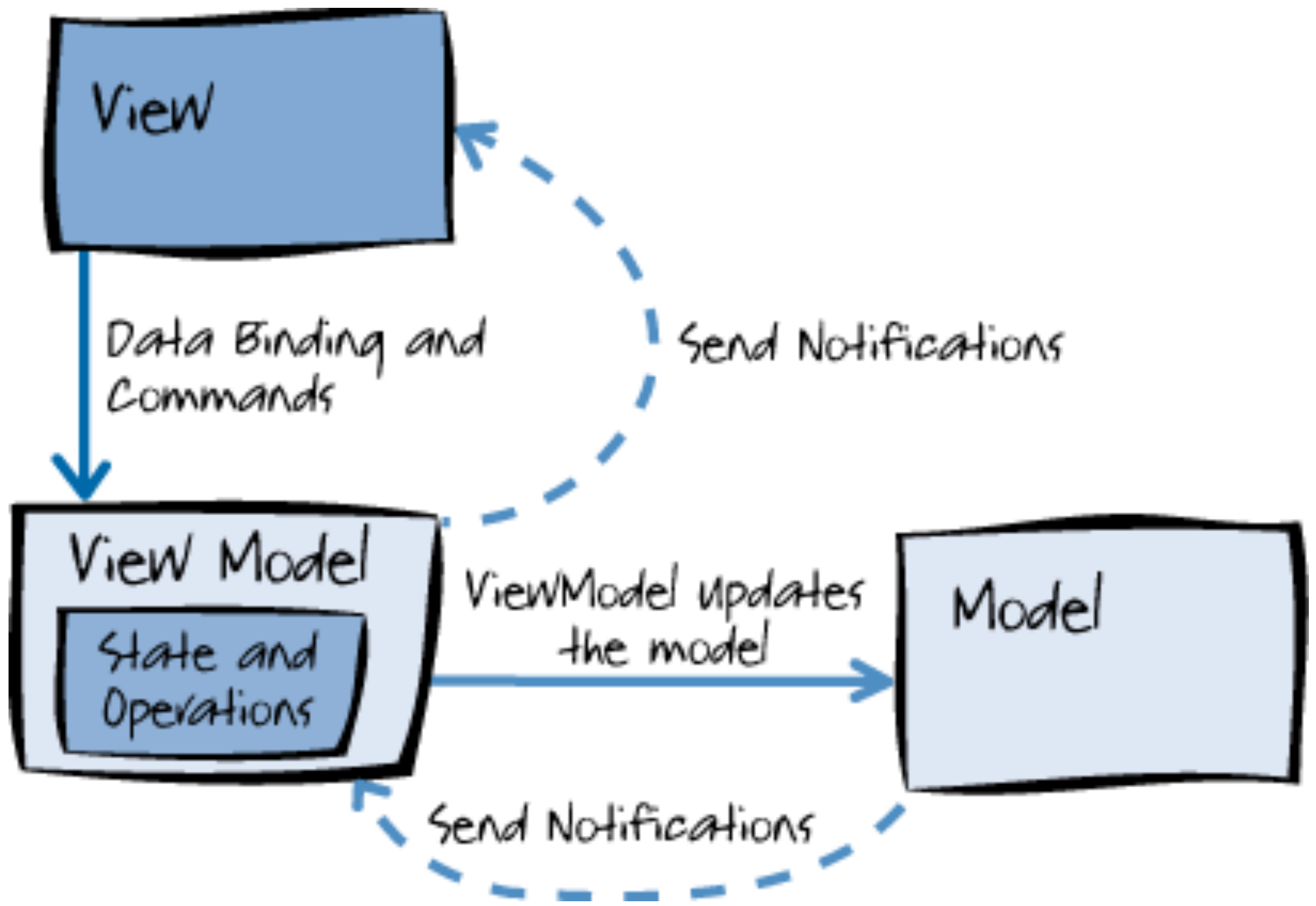
- Since MVC handles the multiple views using the same enterprise model it is easier to maintain, test and upgrade the multiple system.
- It will be easier to add new clients just by adding their views and controllers.
- Since the Model is completely decoupled from view it allows lot of flexibilities to design and implement the model considering reusability and modularity.
- This makes the application extensible and scalable

Model View View-Model

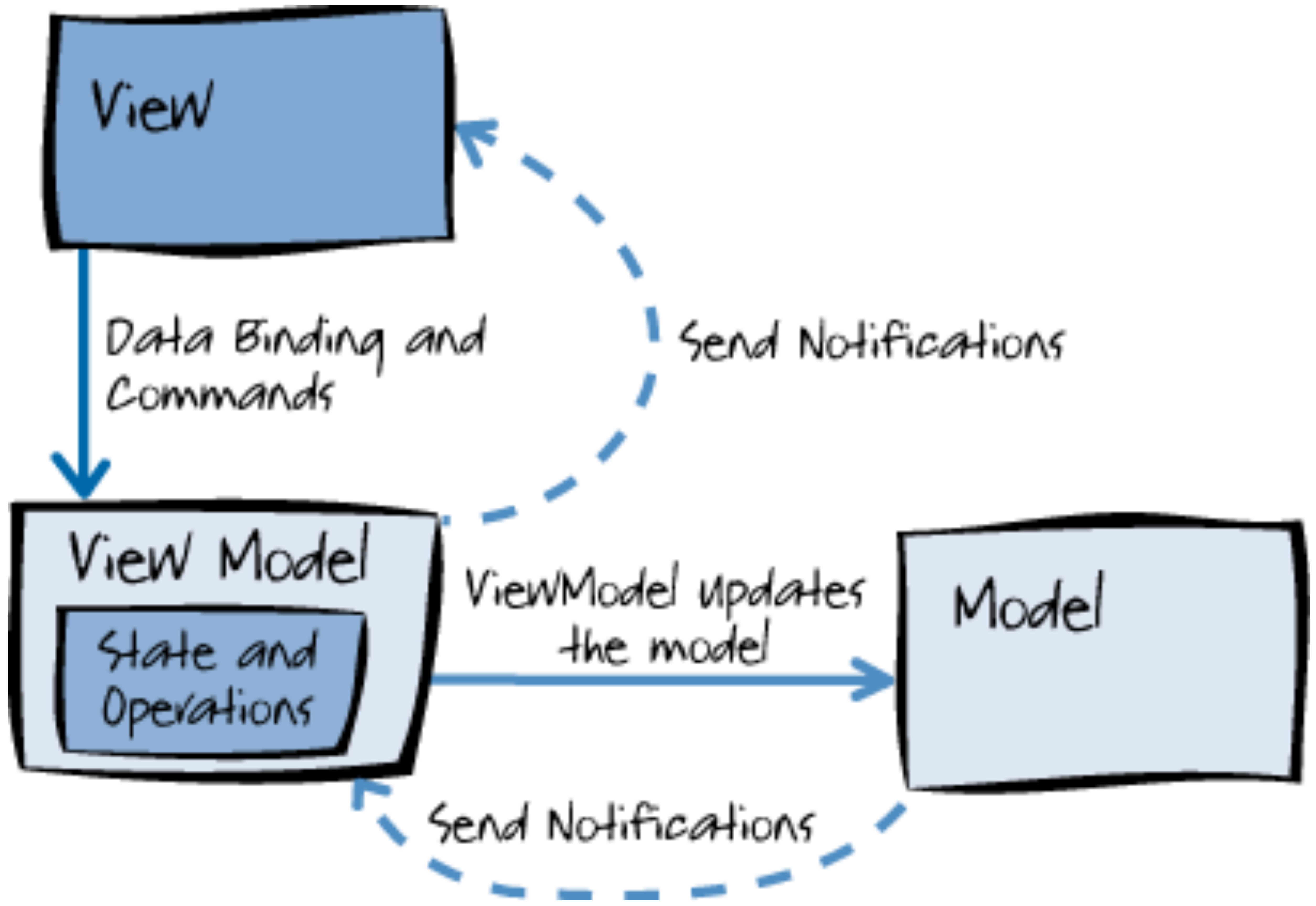
- Model: refers to a domain model, which represents real state content

- View: As in the MVC, the view is the structure, layout, and appearance of what a user sees on the screen.

- View model : an abstraction of the view exposing public properties and commands. Instead of the controller of the MVC pattern, MVVM has ‘bound’ properties - automatically synchronised with the view



MVVM


View
Data Binding and Commands
Send Notifications
View Model
State and Operations
ViewModel updates the model
Model
Send Notifications

Benefits of MVVM

1. Separation of Skills: This enables a separation of responsibilities on teams have a designers and programmers

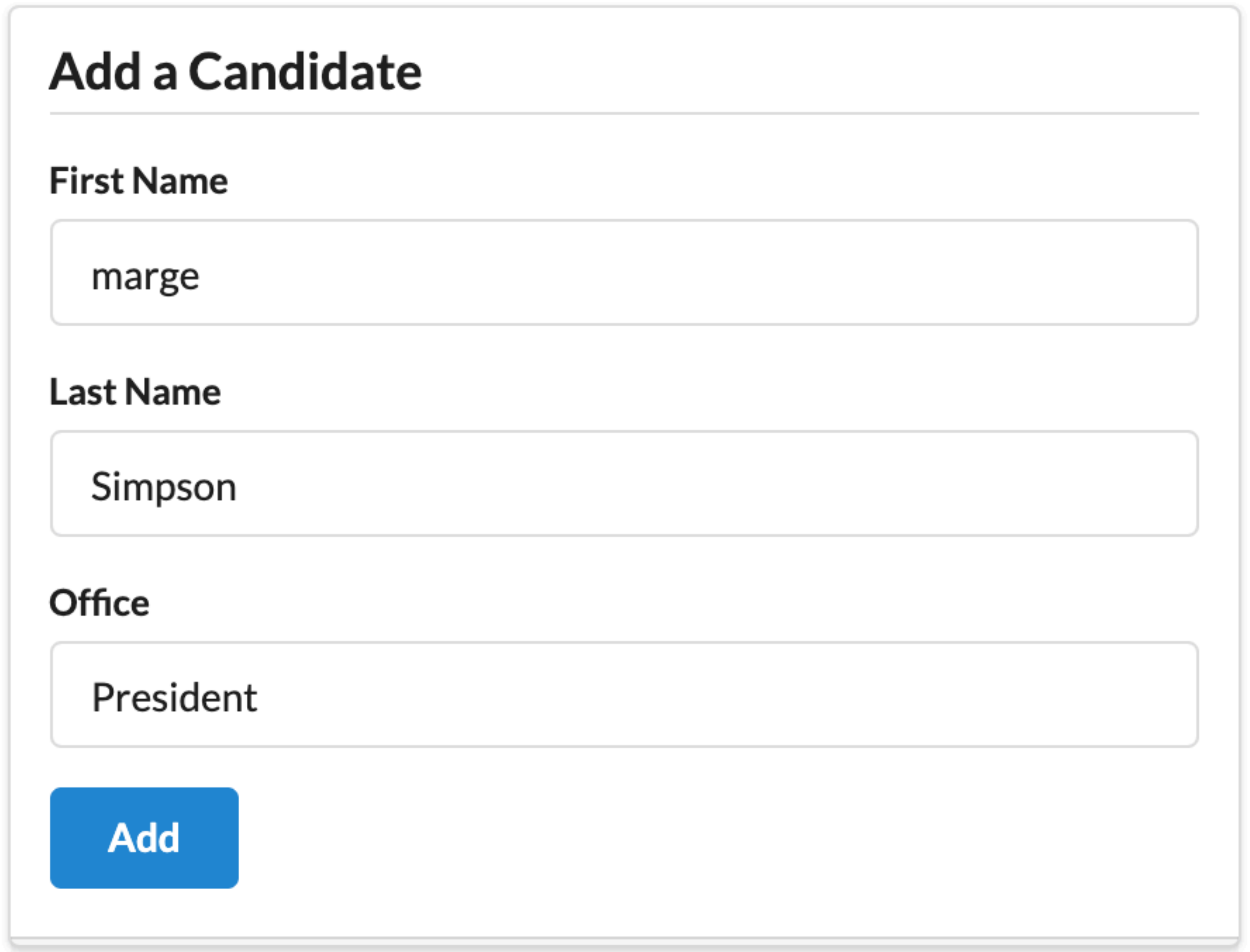
2. Views are agnostic from the code that runs behind them, enabling the same view-models to be reused across multiple views

3. No duplicated code to update views - rely on databinding to keep view and view-model in sync.

4. Since view-model is separated from view view-model classes can be tested independently

5. The Model can be shared across multiple view-models, and can be used to centralise resource access (e.g. Remote API access).

View


Add a Candidate

First Name

marge

Last Name

Simpson

Office

President

```
<template>
  <form submit.trigger="addCandidate()" class="ui form stacked segment">
    <h3 class="ui dividing header"> Add a Candidate </h3>
    <div class="field">
      <label>First Name </label> <input value.bind="firstName">
    </div>
    <div class="field">
      <label>Last Name </label> <input value.bind="lastName">
    </div>
    <div class="field">
      <label>Office </label> <input value.bind="office">
    </div>
    <button class="ui blue submit button">Add</button>
  </form>
</template>
```

View-Model

```
import { bindable } from 'aurelia-framework';
import { Candidate } from '../../services/donation-types';

export class CandidateForm {
  firstName: string;
  lastName: string;
  office: string;
  @bindable
  candidates: Candidate[];

  addCandidate() {
    const candidate = {
      firstName: this.firstName,
      lastName: this.lastName,
      office: this.office
    };
    this.candidates.push(candidate);
    console.log(candidate);
  }
}
```

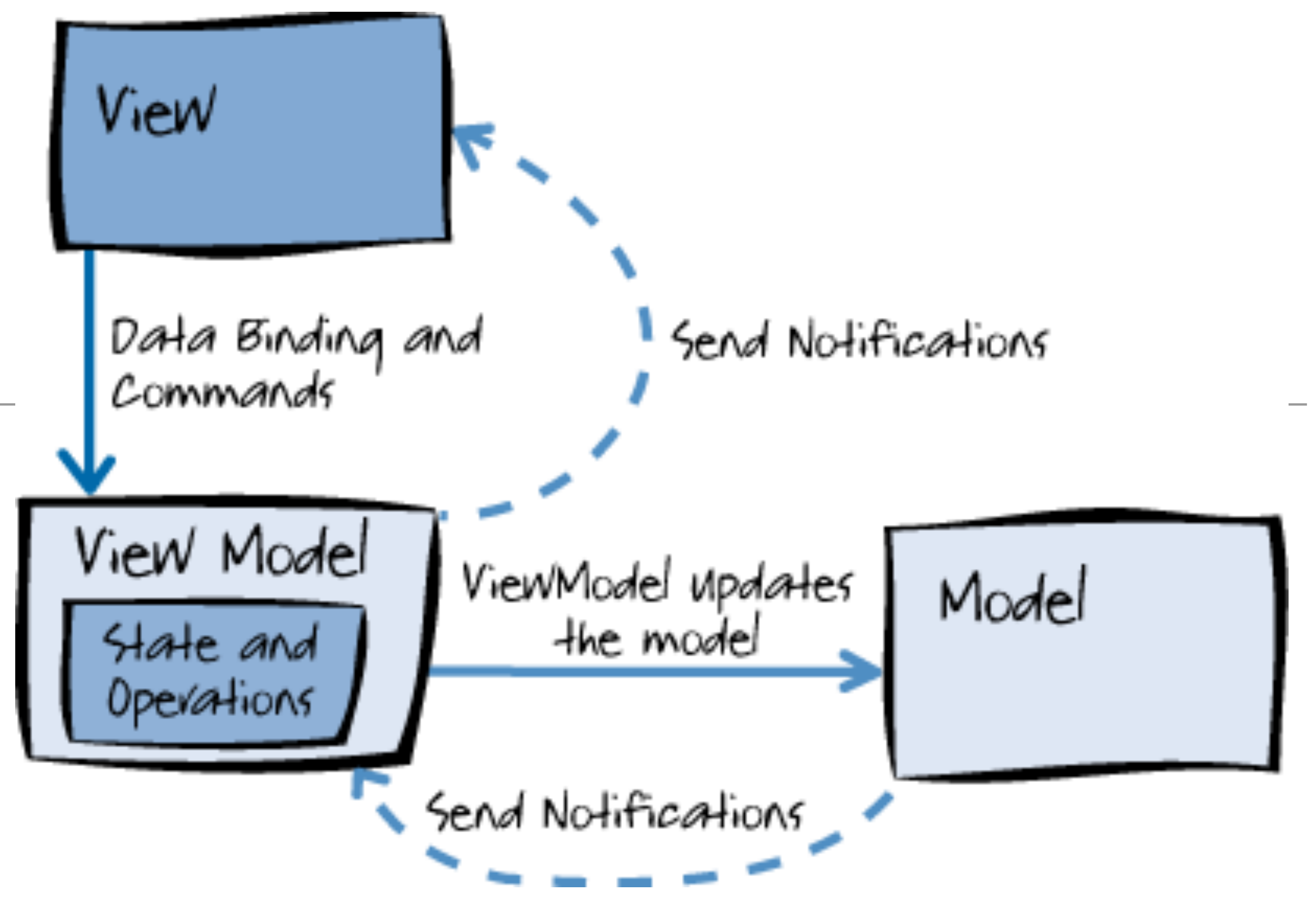
Model -> Candidates View Component

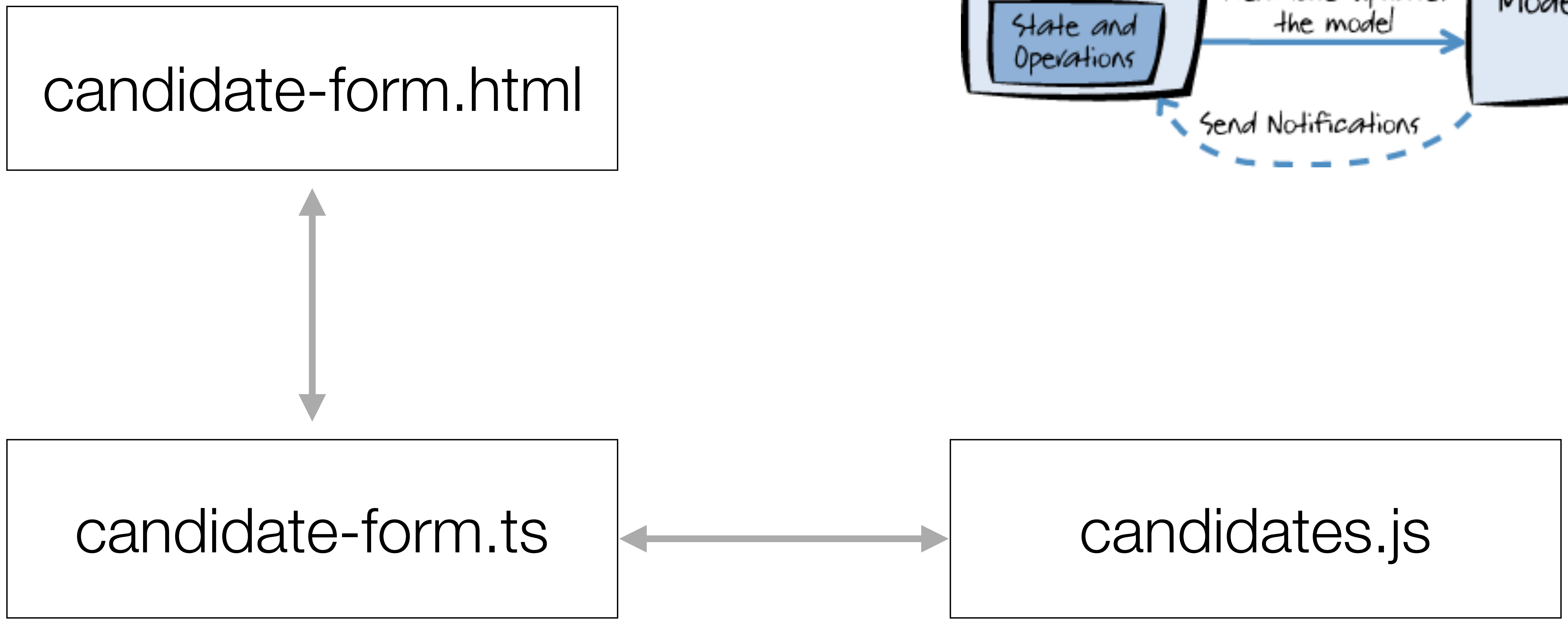
```
import { Candidate } from '../services/donation-types';

export class Candidates {
  candidates: Candidate[] = [];
}
```

```
<template>
  <require from="../resources/elements/candidate-form"></require>
  <require from="../resources/elements/candidate-list"></require>

  <div class="ui stackable two column grid">
    <div class="column">
      <candidate-form candidates.bind="candidates"></candidate-form>
    </div>
    <div class="column">
      <candidate-list candidates.bind="candidates"></candidate-list>
    </div>
  </div>
</template>
```


View
Data Binding and Commands
Send Notifications
View Model
State and Operations
ViewModel updates the model
Model
Send Notifications


candidate-form.html
candidate-form.ts
candidates.js

```html
<template>
  <form submit.trigger="addCandidate()" class="ui form stacked segment">
    <h3 class="ui dividing header"> Add a Candidate </h3>
    <div class="field">
      <label>First Name </label> <input value.bind="firstName">
    </div>
    <div class="field">
      <label>Last Name </label> <input value.bind="lastName">
    </div>
    <div class="field">
      <label>Office </label> <input value.bind="office">
    </div>
    <button class="ui blue submit button">Add</button>
  </form>
</template>
```

$$\begin{tikzcd} \text{candidate-form.html} \arrow[d, leftrightarrow] & \\ \text{candidate-form.ts} \arrow[r, leftrightarrow] & \text{candidates.js} \end{tikzcd}$$

```typescript
import { bindable } from 'aurelia-framework';
import { Candidate } from '../../services/donation-
types';

export class CandidateForm {
  firstName: string;
  lastName: string;
  office: string;
  @bindable
  candidates: Candidate[];

  addCandidate() {
    const candidate = {
      firstName: this.firstName,
      lastName: this.lastName,
      office: this.office
    };
    this.candidates.push(candidate);
    console.log(candidate);
  }
}
```

```typescript
import { Candidate } from '../services/donation-types';

export class Candidates {
  candidates: Candidate[] = [];
}
```