Model-View Controller MVVM, State, Props, Container-Presentation

Cristian Bogdan

Model

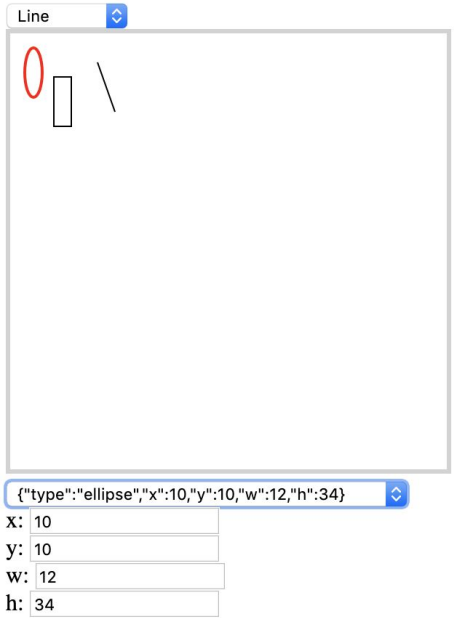
Keeps the application data

Discussion: more crucial and less crucial model properties

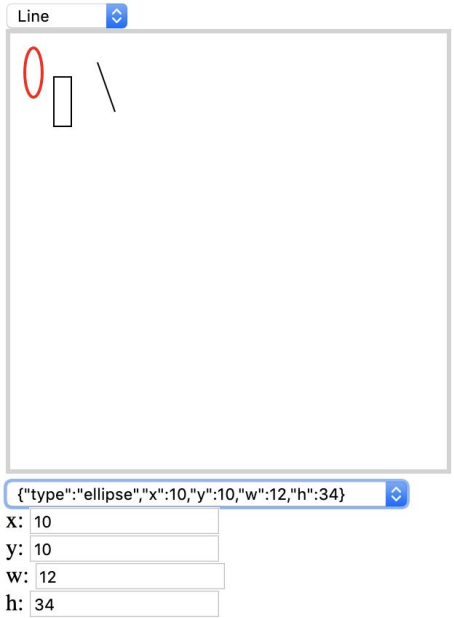
dishes, guests Vs currentDish

shapes Vs currentShape

- The Basic Draw editor displays the current shape in both views but it is not crucial for the application use case



View



Visualizes the data

Not necessarily graphical (e.g. PostScript text for printer)

Typical implementation: Observer pattern

View is Alice, Model is Bob

Basic Draw:

- Canvas View (with shape type selector)
- Shape editor View (with shape selector, shape property editors)

Incremental update: changes only the view parts specified in the update payload.

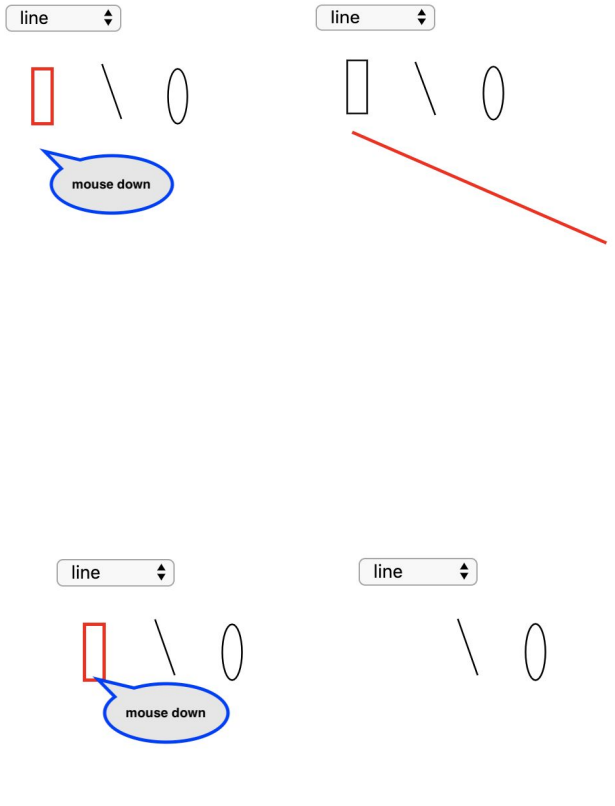
Nowadays the frameworks take care of optimizing this, so it’s enough to do **full update** (re-render)

Controller

1. Listens to events from a view (Bob is View element, Alice is Controller)
2. **Interprets** the events
3. Changes the model (this will lead to model-view notification)
4. (also shows, hides the view)

The same event can be interpreted differently by different controllers of the same view. Example: **mouse down+ mouse drag** on Basic Draw Canvas view

- If the mouse-down is on the Canvas directly, we activate the “Create Controller” to create a Shape, which becomes the current shape (red)
- If the **mouse down** is on a shape we activate a “Move Controller”, which will move the respective shape on mouse drag
- A “boss” controller needed to activate one of them...



MVC Discussion

The M, V, C concerns, as well as Observer (M-V), Listener (V-C) will be recognized by most interaction programmers

There are cross-cutting concerns like remote data, that muddle the M, V, C separation

The M-V, and C-M separation works well in most cases

The V-C separation is a bit more problematic

- If the Controller needs to add listeners to the relevant input controls, it needs to know the View internal structure and graphical layout. Separation breaks...
- Or it can listen to bubbling events, but then its code becomes a string of if(event.target…) which also need to know a bit about the view layout (e.g. button labels)
- The modern way is to make a bit of a compromise and let the view listen to the event, but then delegate it to the controller, by calling a controller callback

Controller passes callback to View (custom event)

```
class MyView{
  constructor(model, root, changeInteger){
     this.root=root; this.model=model; this.changeInteger=changeInteger;
  }
  render(){ h("div", {}
     , h("button", {onClick: event=>this.changeInteger(-1)},"-")
     , model.getNumberOfGuests()
     ).render(this.root);
  }
}

class MyController{
  constructor(model, root){
      new MyView(model, root,
              x=> model.setNumberOfGuests(model.getNumberOfGuests()+x)
      ).render();
  }
}
```

This is as if the view would fire a "changeInteger" **custom event. Controller is Alice, View is Bob**
The original event is usually not needed...

Note that the View does not care what the Controller is doing with the integer it sends.
We could decouple the view totally from the model by sending it an integer, and update signals. We will do that later.

Binding

With binding there is typically no need for a Controller at all.
Drawback: too much interaction code (Controller concern) in the View.

Rather than observe the whole Model, we observe just a variable. For example an integer. Then mentioning the variable (or an expression) in a View will **keep it updated**. Examples in **Vue.js Templates (details later).**

```
{{numberOfGuests}}
```

A separate observer (binding) for e.g.the dishes array. Array rendering:

```
<div v-for="(dish) in dishes" >{{dish.title}}</div>
```

Changing the CSS class of a BasicDraw shape depending on whether it is the current shape or not. Also **conditional rendering** depending on the shape type.

```
<rect v:if="shape.type=rectangle" v-bind:class="shape==currentShape?current:normal" >
```

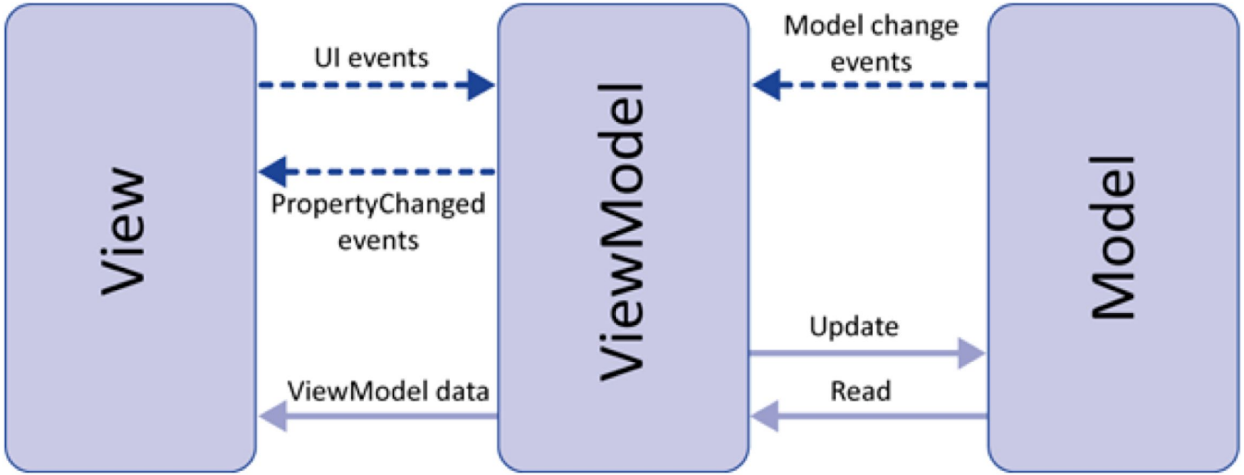
Two-way binding will both render the value **and** change it from the user input. This is an INPUT editor for number of guests. v-model means that the model of our <input> is numberOfGuests...

```
<input type="number" v-model="numberOfGuests">
```

Model-View-ViewModel

In MVVM the View does not observe the Model directly but it is re-rendered (updated) by a ViewModel, which is also the Controller. The ViewModel, in turn, observes the Model.

Used often in connection with one-way or two-way binding (PropertyChanged events are the binding notifications).



Controller as ViewModel that adds view as observer

```
class MyView{
  constructor(integer, changeInteger){
     this.changeInteger=changeInteger; this.integer= integer;
  }
  render(){ return h("div", {}   // we return h() instead of rendering
       , h("button", {onClick: event=>this.changeInteger(-1)},"-")
       , this.integer
     );
  }
}
class MyViewModel{
  constructor(model, root){
     const update=()=>new MyView( model.getNumberOfGuests(),
            x=> model.setNumberOfGuests(model.getNumberOfGuests()+x)
     ).render().render(root);
     model.addObserver(update);
     update();  // initial rendering
  }
}
```

Note that the view became totally oblivious of the type of integer that it shows and edits.

Also the view does not do much with its state so it can become a function. See next slide.

Even simpler: functional View

```
function MyView(integer, changeInteger){
    return h("div", {}
       , h("button", {onClick: event=> changeInteger(-1)},"-")
       , integer
       , h("button", {onClick: event=> changeInteger(1)},"+")   // a little extra
     );
}

class MyViewModel{
  constructor(model, root){
      const update=()=> MyView(model.getNumberOfGuests(),  //no new!
             x=> model.setNumberOfGuests(model.getNumberOfGuests()+x)
      ).render(root);
      model.addObserver(update);
      update();  // initial rendering
  }
}
```

Since the view code is basically its render() function, we can transform it into a function. We also let the controller call render(root)

View/Component function names start with a capital letter as a matter of style (or as requirement for some frameworks)

Even simpler: functional ViewModel and View!

```
function MyView(integer, changeInteger){
    return h("div", {}
       , h("button", {onClick: event=> changeInteger(-1)},"-")
       , integer
       , h("button", {onClick: event=> changeInteger(1)},"+")
     );
}

function MyViewModel(model, root){
      const update=()=> MyView(model.getNumberOfGuests(),
            x=> model.setNumberOfGuests(model.getNumberOfGuests()+x)
      ).render(root);
      model.addObserver(update);
      update();  // initial rendering
}
```

Transforming single-method classes in functions does not always work, especially when a View or viewModel needs to keep state. But it often works and is preferred by programmers.

React makes special provisions for supporting functional components, for example the State hook.

Hierarchical MVC / MVVM

Consider an **<input type="text" value="initialValue">** . It does MVC internally!

- View: the box with focus mark around it, with text inside it, cursor,
- Model: has focus? Is mouse over? cursor Position, Current Value
- Controller: mouse down (move cursor), mouse drag (select), mouse up, key typed, focus, blur. All these events **change the model**, which update the View.
- Initial values (e.g. attributes) come **from the outside world**
- Sometimes the controller fires events (like <input> fires **oninput** and **onchange**), **to the outside world**

The modern component. State and props

<input type="text" value="initialValue" >

- Internal component mini-model is called **state**. It is different from the Application State (model)
  - Or from the state of the parent (enclosing) components
- Initial value properties/attributes are called **props. Read only.** Used to get data **from** the outside world
- A few special props are used to fire **events to the outside world**. These are called **custom events**

```
const MyInputComponent = (initialValue, onValueChange) =>
     h("input", {type:"text", value:initialValue,  // event.target is the <input> !
                  onInput: event=> onValueChange(event.target.value) })
```

Just like HTML elements take a single attribute parameter (h("input", {attrs})), modern components take one single **props parameter**, which they typically access through **destructuring** (though not necessarily)

```
const MyInputComponent = ({initialValue, onValueChange}) =>
     h("input", {type:"text", value:initialValue,
                  onInput: event=> onValueChange(event.target.value) })
```

Which components need to keep state?

Keeping state= have other internal models, different from the application Model

- SidebarView?
- SummaryView?
- SearchView?

- Basic Draw Canvas?
- Basic Draw shape edit form?
- Basic Draw shape property editor?

As shown before, HTML Elements keep state. Sometimes (but not in React!) one can use them as “memory” and not define own state properties. Examples

- The Shape Type chooser of the Basic Draw Canvas
- The search terms of the Dinner planner search views

React overrides the HTML Element values from its component props and state

Props down, events up. Custom components

Typically “the outside world” is another component, that uses our component. **Props down (from the enclosing component), events up (to the enclosing component)**

```
const MyInputComponent = ({initialValue, onValueChange}) =>
     h("input", {type:"text", value:initialValue,
                    onInput: event=> onValueChange(event.target.value) }) // events up!
```

The enclosing component can then use MyInputComponent by calling the function:

```
MyInputComponent({initialValue: model.getNumberOfGuests(),       // props down
                    onValueChange: x=> model.setNumberOfGuests(x)});
```

Or use MyInputComponent in hyperscript like an element! (custom component)

```
const EnclosingComponent= ({model})=>
     h(MyInputComponent, {initialValue: model.getNumberOfGuests(),
                             onValueChange: x=> model.setNumberOfGuests(x)}
```

Container and Presentational components

Presentational components are generic and independent of any application model (aka Application State)**.** They can be written by somebody else, with no knowledge of your application

- number editor (can be used in Sidebar)
- table display (sortable, etc) (can be used in Sidebar, Summary, DishDetails)
- “folder” icon+label display(s) (can be used in Search results, maybe Sidebar, Summary…)

Even if you need to write a presentational component (e.g. **DishDetailsPresentation**), you can still apply the main principle: **no knowledge of the Application State**! Just knowledge of the Spoonacular structure and number of food portions.

Container components do the “glue” work between **Presentational components** and Application State

- Map the props of Presentational to Application state props
- Map the **custom events** of the Presentational to callbacks that change the App state
- Typically render no graphical appearance, just use one or more Presentational
- Can be generated automatically using e.g. react-redux

Container and Presentational components

```
const EnclosingComponent= ({model})=>
     h(MyInputComponent, {initialValue: model.getNumberOfGuests(),          // props down
                          onValueChange: x=> model.setNumberOfGuests(x)})   // events up
```

Note that the **Enclosing** (**container**) component **renders no graphical elements** (no DIV etc) but just connects (**glues**) the inner (**MyInputComponent**) component to the application state (model).

MyInputComponent Presentational component props and implementation have no knowledge of the Model! (application state)

The previously introduced view-model is also a container, but not a component that can be used with h(). It glues to the Model:

```
function MyViewModel(model, root){
      const update=()=> MyView(model.getNumberOfGuests(),                // props down
            x=> model.setNumberOfGuests(model.getNumberOfGuests()+x)     // events up
      ).render(root);
      model.addObserver(update);
      update();  // initial rendering
}
```

react-redux
connect() returns a glue function!

View and Presenter: "glue" function

Instead of defining Container components explicitly, one often *generates* them using an ordinary function that "glues" a Presentational component, which it gets as parameter, to a certain App state. For example for the DinnerModel.

```
const glueToModel= (View) => h(View,
       { guests: model.getNumberOfGuests(), setGuests: x=> model.setNumberOfGuests(x),
         dishes: model.getMenu(), dishSelected: d=>model.removeDish(d)} );

const Sidebar= glueToModel(SidebarView);
const Summary= glueToModel(SummaryView);
```

You will typically have several glue functions, depending on where your Presentational components come from, what props need to be glued, etc.

Example: possible Presentational components in the dinner planner

SidebarView

- an integer editor (glued by the Container to model.mumberOfGuests)
- a sortable table (glued by the Container to model.dishes)

SummaryView

- a sortable table (glued to model.getIngredients())

SearchView

- a list of icons, maybe sortable, like a file explorer folder (glued by Container to the search results)

Once the container-presentation separation is done, you can plug in any component from the internet for the number editor, table, or icon list).

In DOM using h() you can e.g. use Bootstrap DataTable

In React or Vue.js (TW4, project) you will find many such generic components for tables, number editors, icon lists etc.