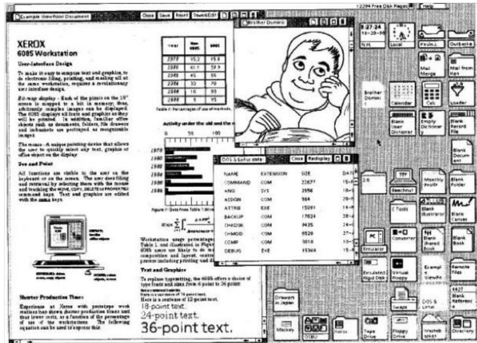
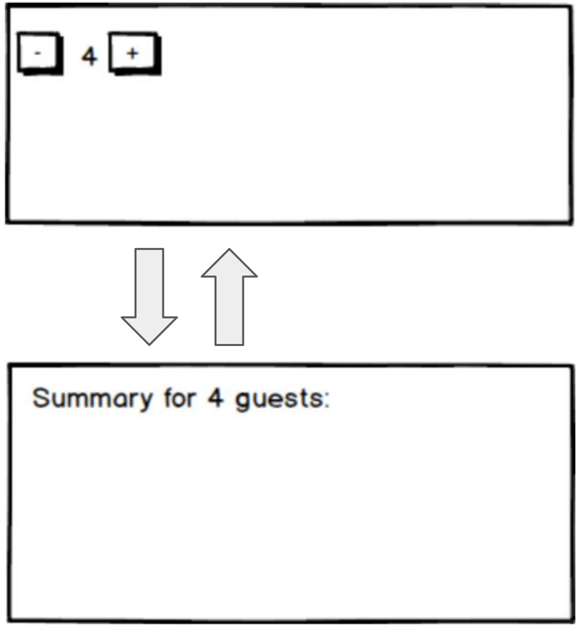
Architectures: MVP, MVC, MV-VM

DH2642

Most concepts presented here are valid for most Graphical User Interface technologies and can be implemented in many ways.

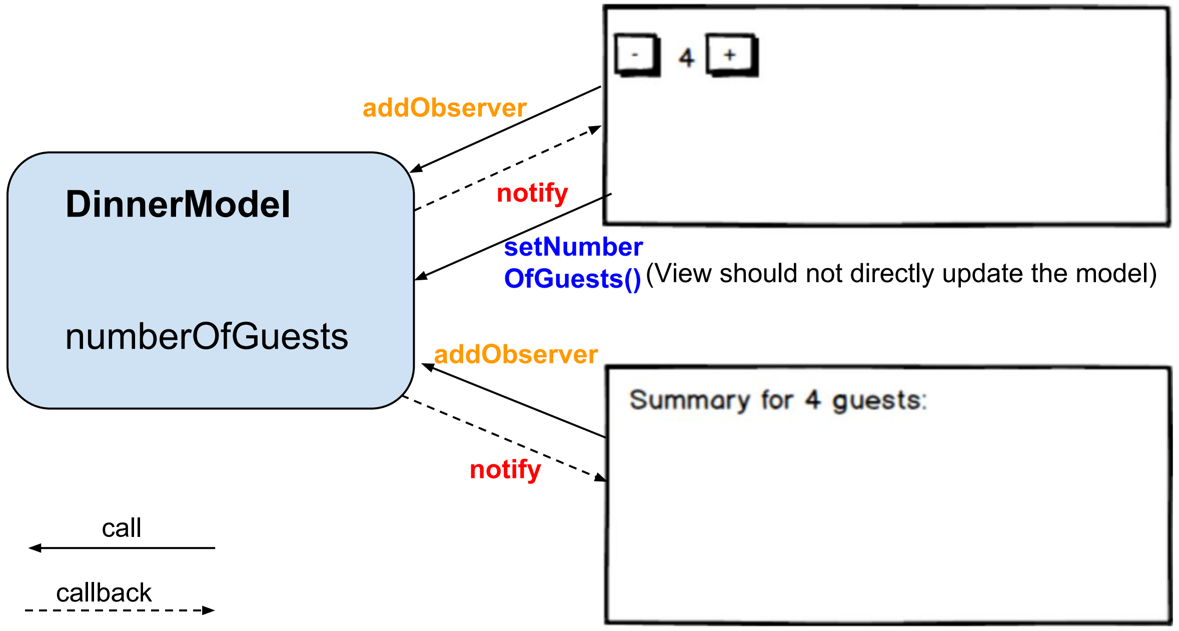


Given two representations of the same data, where to keep the Data??



Unexperienced interaction programmers tend to keep data in both representations, ending up with a lot of interdependency

Observer-**Observable**



Graphical (usually) representations **observe** the Data Model. They are called Observers

Model is also called **Observable**.

New Observers can be added at any time to an **Observable**. They can also be removed.

Observer-**Observable** is an Alice-**Bob** situation. It is known as “the Observer pattern”

Observer pattern implementations

JavaScript Proxy

- Gives notifications of changes in any Object. Used by Vue

JavaScript (handmade)

- Each Observer subscribes by giving the Observable (model) a function (callback)
- The Observable (model) keeps an array of Observer callbacks.
- When it wants to notify, the Observable simply calls() each Observer function

Java

- `java.util.Observable` class, extend it to create your own model
- `java.util.Observer` interface (a single method notify())

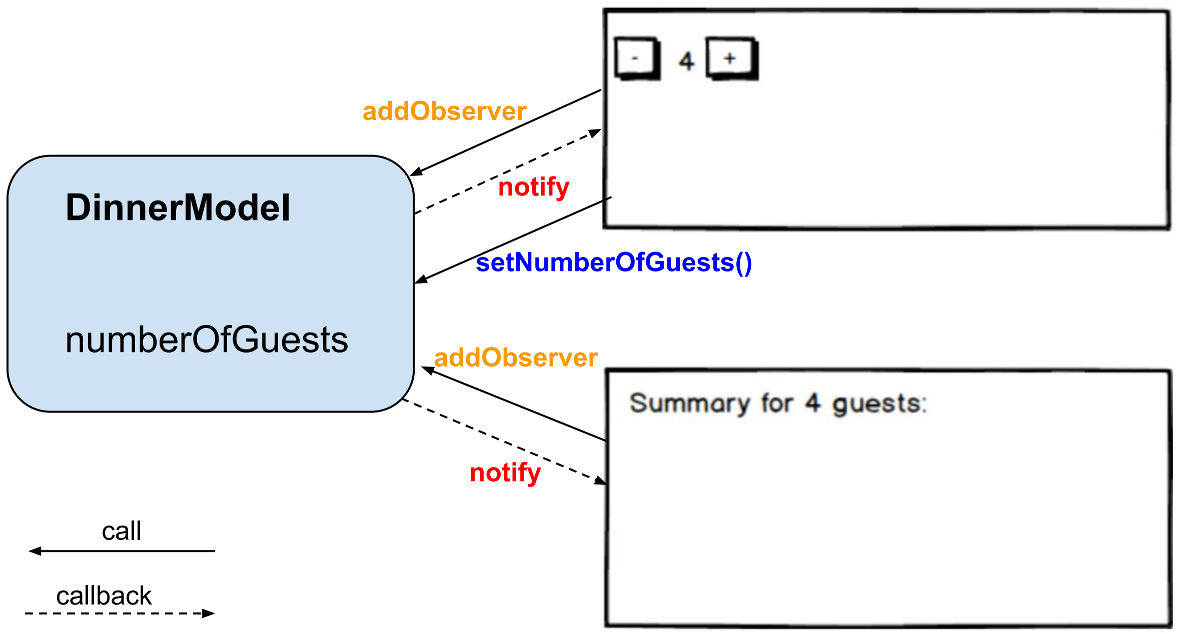
Bob the model

Who notifies? **(Bob)**	**DOM Element (INPUT** box)	**Custom component (`NumberEditor`)**	**Observable (model)**
Who subscribes? (Alice)	e.g. `NumberEditor`	Code using `NumberEditor`	**Observer**
subscription	`addEventListener("change", f)`	`{ setNumber: f }`	addObserver(f)
notifications	`f(change1) f(c2) f(c3)`	`f(3) f(4) f(5) f(2)`	`f() f() f() f() f()`

eg, addEventListener("mouse up", listener)

probably an object variable in the component NumberEditor

View-Presenter separation



ie, View

The graphical representations now need to

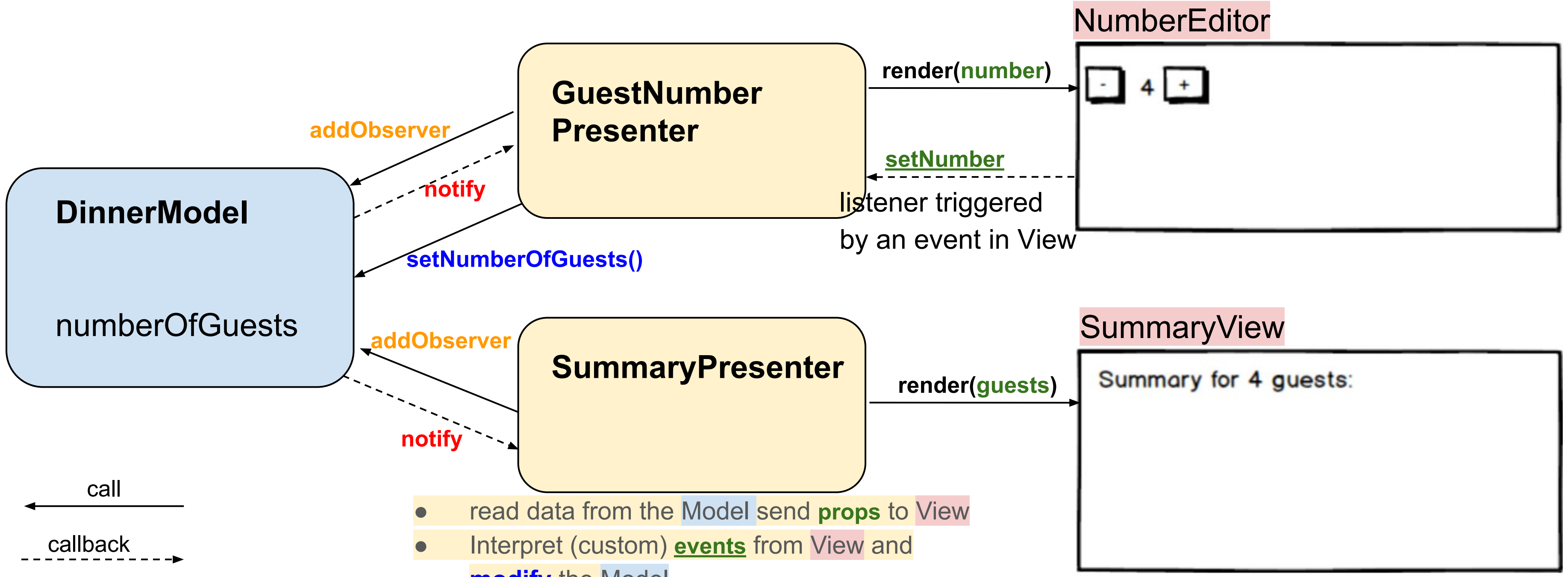
- read data from the Model
- Represent data
- Detect user events eg: input change
- Interpret events and modify the model eg: read the current input number and call setNumberOfGuests(number)
- When notified by the model, read data again and re-render

Alternatively, the yellow tasks will be done by the presenter.

Model-View-Presenter

Views are aware of the Model/ use case to different degrees. The NumberEditor is generic, while the SummaryView is more tied to the dinner use case.

It is the job of the Presenter to **mediate** between Model and View: **adapt** the data formats to **props** expected by the View, transform (or not) **custom events** into Model **changes** etc.



- read data from the Model send **props** to View
- Interpret (custom) **events** from View and **modify** the Model
- When notified by the Model, read data again and re-render View

- Represent data
- Detect user events

Example Presenter implementations

The Model application logic will not allow setting wrong **`number`** *values. The Presenter may need to treat such cases.*

```
function GuestNumberPresenter({model}){
   const renderView =()=> h(NumberEditor, {
                                  number:    model.getNumberOfGuests(),
                                  setNumber: x=> model.setNumberOfGuests(x)});

   let myView= renderView();   // first render
   model.addObserver(()=> {                  // View re-render and DOM update
         const updated= renderView();       // re-render
         myView.parentElement.replaceChild(updated, myView);   // replace DOM
         myView= updated;     // see JavaScript closure
   });
   return myView;
}
```

return a HTML component

API route analogy:

-> similar as the response to a client (`number: model.getNumberOfGuests()`)

-> similar as the fetch request call by the client (`setNumber: x=> model.setNumberOfGuests(x)`)

Similar for `SummaryPresenter` but

```
    const renderView=()=> h(SummaryView,{guests:model.getNumberOfGuests()});
//test
const model= new DinnerModel();
document.body.append(h(GuestNumberPresenter, {model:model}));
document.body.append(h(SummaryPresenter, {model}));   // short for {model:model}
```

Presenters own/contain Views

Presenters can be implemented as Components but they should not render any UI (or very little). One Presenter can **combine** several Views

Views are contained into Presenters, **only one** Presenter should know/care about a View therefore Views are "lower down" in the hierarchy

Data (like **`number`**) travel **down** from Presenter to View. **Props down**
Custom events (like **`setNumber`**) notifications travel **up** from View to Presenter. **Events up**
This is known as **Props down, Events up**, and can be applied separately from MVP

Presenter components are often called Containers, and Views are known as Presentational :) components. See React evangelist older text.

Not in Lecture, Skip!

react-redux
connect() returns a glue function!

Generate Presenters with “glue” function

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

See also react-redux, Vue-redux, Vuex.

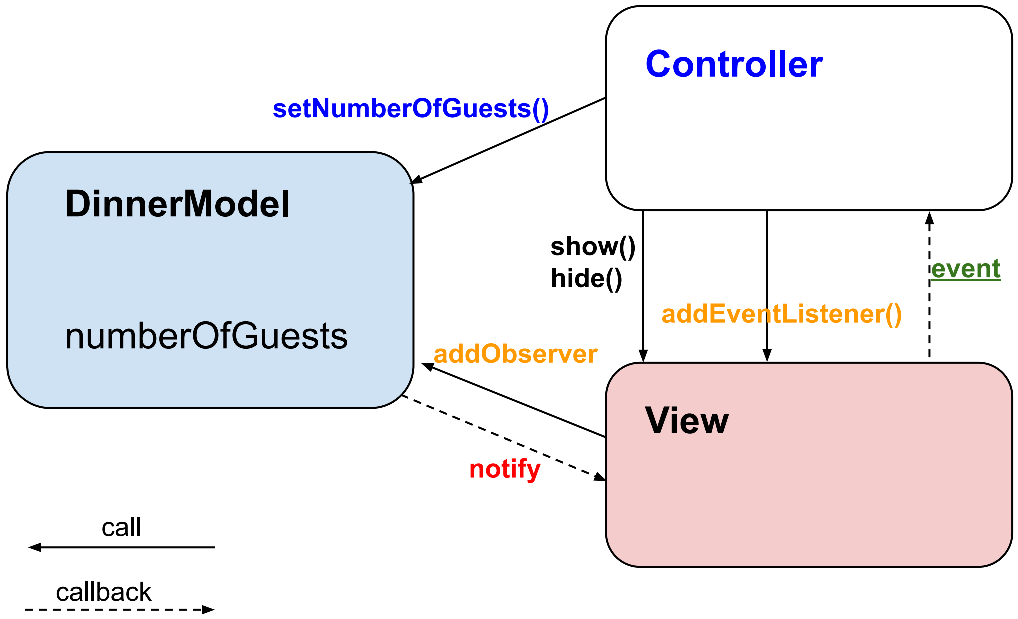
```
const glueToModel= (View) =>
h(View, {
            guests: model.getNumberOfGuests(),
            setGuests: x=> model.setNumberOfGuests(x),
            dishes: model.getMenu(),
            dishSelected: d=>model.removeDish(d)
        }

);

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

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

Model-View-Controller



MVC terminology will be recognized by most interaction programmers

One MVC View can have more controllers depending on its mode (e.g. graphical editor: same event, different model semantic)

In MVP

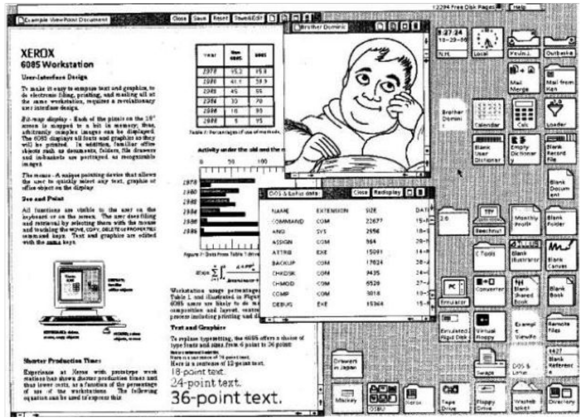
- Presenter does the Controller job (change Model), and more.
- A part of the Controller is in the View (listen to events, convert to **custom events**) ie, NumberChange

M→V→C→M is circular

M←→P←→V is linear

Most component-oriented architectures end up with MVP because the component hierarchy requires a linear relationship between P components and V components

MVC in the History of interactive computing



MVC was introduced by Trygve Reenskaug (1979) at Xerox Palo Alto Research Center (PARC)

With MVC and the Smalltalk language, PARC achieved one of the first WIMP (windows, icons, menus, pointers) GUIs: Xerox Star

In the 1980s the Star inspired Apple Lisa and the slightly more famous Apple Macintosh. Microsoft Windows was designed with the help of former PARC engineers.

Other PARC inventions that we use today: Ethernet, PostScript, laser printer, WYSIWYG editors, bitmap graphics,...

Model-View-ViewModel

MV-VM can be regarded as a MVP variant.

The difference is that there is **data binding** between View and Presenter (ViewModel)

Data binding examples in Vue or Angular: The View is composed by a **“template string”**

- `"Summary for {{guests}} guests"`
  One-way binding (aka interpolation) to the **`guests`** value stored in the ViewModel.

- `'Enter number: <input type="text" v- model="number" />'`
  will display the **`number`** value stored in the ViewModel, but will also be able to change it (which will lead to a Model modification by the ViewModel). This is **two-way binding.**

Vue translates its templates to **JSX** nowadays. JSX is much more flexible than template strings.
Before React and JSX, templates and binding were the biggest wonder in Web frameworks.

MVVM was introduced by Microsoft in 2005 for e.g. Silverlight. Many other technologies support **binding** (e.g. JavaFX)

Technology evolution at Model level

One would think that the Model (abstract data) level is just an object in a given programming language

Reactive programming: ReactiveX (used by Angular, RxJS). RxJava, RxPy
Everything is an Observable (from Model to remote data access, and even DOM events)

For large applications, and lots of programmers, managing **application state** can become unscalable
=> **State managers:** Flux, Redux (by React), Vuex (by Vue) model + other state data

Frameworks work at Presenter-View level
Redux, Vuex, RxJS work *mostly* at abstract data level

=> one can use any of them with any framework (e.g. even Vuex with React…)

Or even with no-framework (e.g. the course hyperscript using Redux, etc)

Not in Lecture, Skip!

Model-View-ViewModel

Templates and Binding supported by

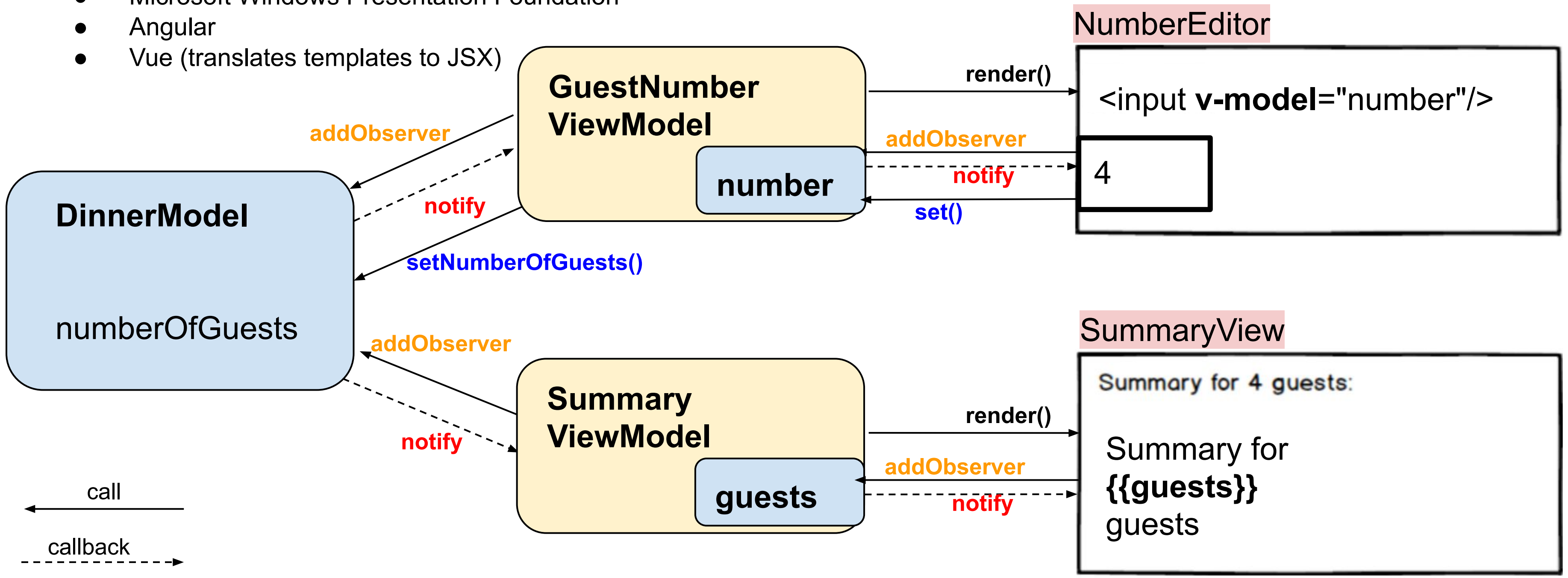
- Microsoft Windows Presentation Foundation
- Angular
- Vue (translates templates to JSX)

The ViewModel prepares a mini-model for each value displayed in the View.

SummaryView: The View prepares a **template** that has a {{mini-observer}} (aka "interpolation") for each dynamic value.

When the mini-model notifies, the dynamic value is updated. This is called **one-way binding**

NumberEditor can also change the value bound. This is **two-way binding**



Reactive programming

Observables that rely on other notifying Objects (**Bobs**)

- an Observable that observes another Observable and only notifies Observers when a certain property of the 2nd Observable has changed (say DinnerModel's currentDish)
- an Observable that observes several other Observables and notifies whenever one of them notifies
- an Observable that observes the status of a HTTP web API call, and notifies when it finished with success or error.

Combining Observables in such ways is called **Reactive Programming**, and can be done in many programming languages. See RxJS, RxJava, RxPy.
But you can also apply the ideas above in your own code, no need for an Observable library.

Often chain notification situations (e.g. currentDish change leads to HTTP call which notifies) can be solved with Reactive Programming and need no framework features (e.g. no React useEffect, or Vue watchers)