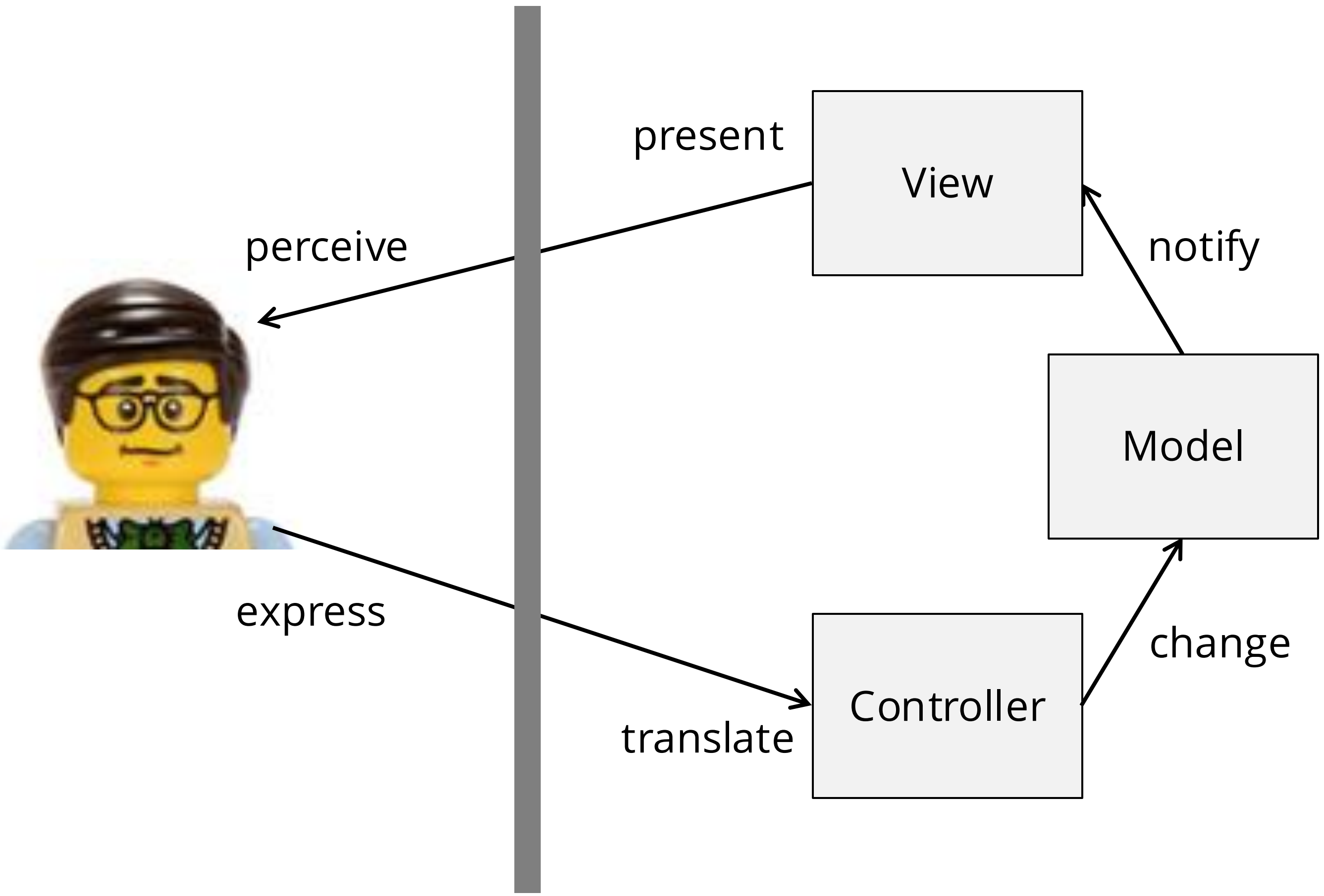
Model View Controller (MVC)

- Benefits of MVC
- Basic Implementation
- Todo Example
- MVC Variants

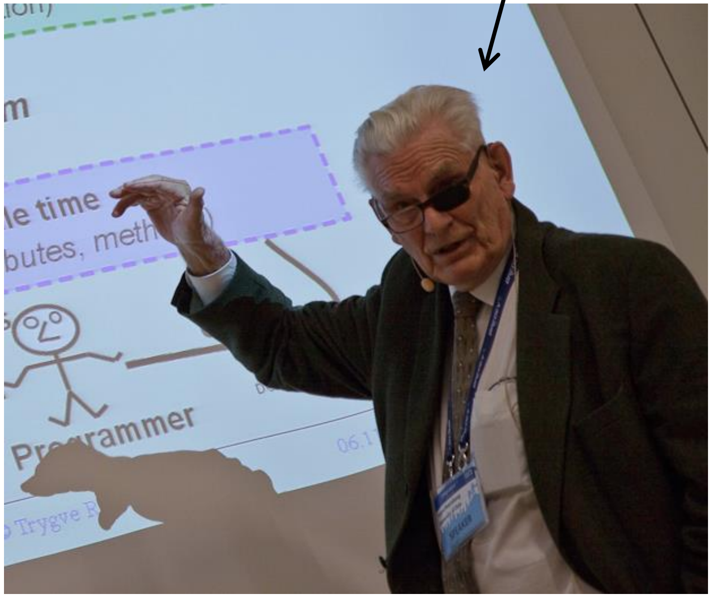
Model-View-Controller (MVC)

MVC was the first MV* interactive system architectures



Model-View-Controller (MVC)

- Developed at Xerox PARC in 1979 by Trygve Reenskaug
  - for Smalltalk-80 language, the precursor to Java
- Became a standard *design pattern* for GUIs
- Used at many levels
  - Overall application design
  - Individual components
- Many variations of MVC (MV*):
  - Model-View-Adaptor (MVA)
  - Model-View-Presenter (MVP)
  - Model-View-ViewModel (MVVM)



Trygve Reenskaug

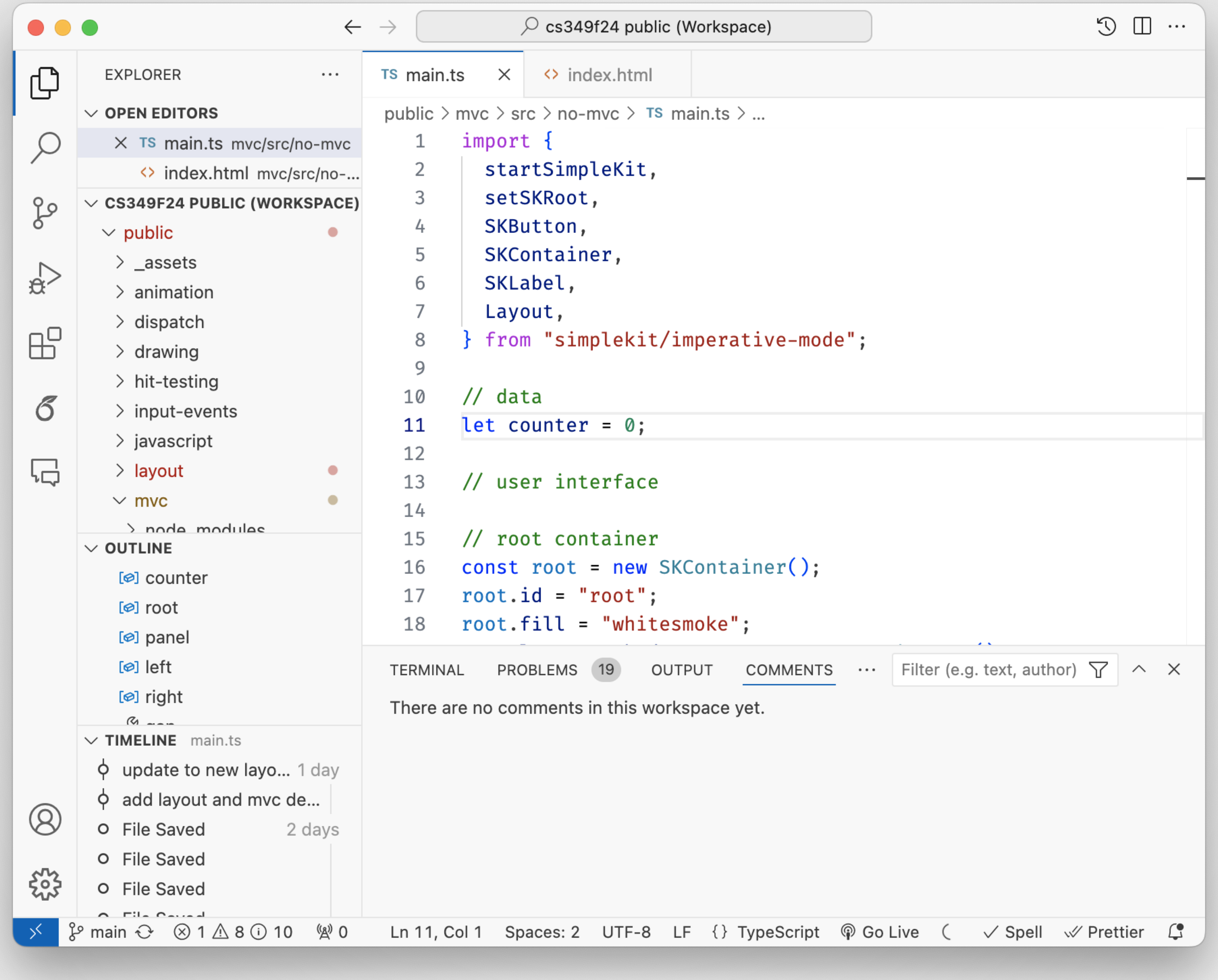
Why use MVC?

1. Separate data, state, and "business logic" from user-interface

- Ideally, View and Controller implementations can change without changing Model implementation, e.g.:
  - Add support for a new interface (e.g. different device)
  - Add support for a new input device (e.g., touchscreen)

2. Supports multiple views of same data, e.g.
  - View numeric data as a table, a line graph, a pie chart, ...
  - Present simultaneous “overview” and “detail” views
  - Distinct “edit” and “preview” views

3. Separation of concerns in code
  - code reuse
  - unit testing



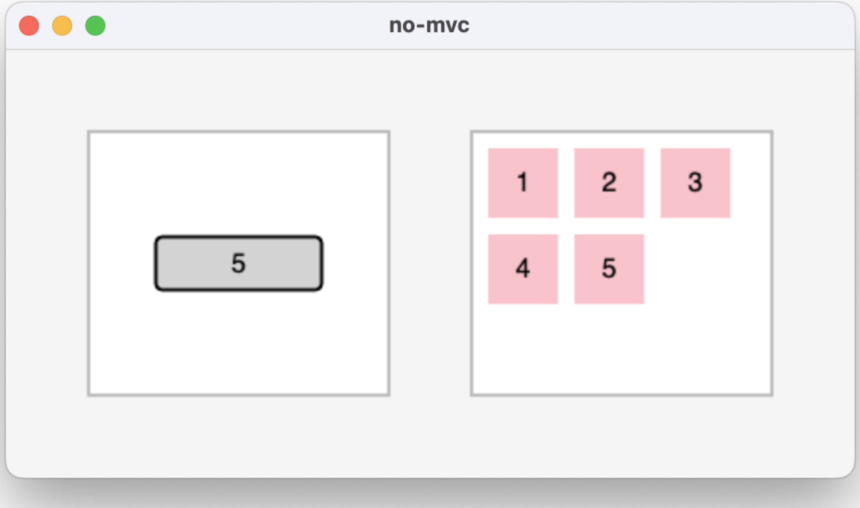
```typescript
import {
  startSimpleKit,
  setSKRoot,
  SKButton,
  SKContainer,
  SKLabel,
  Layout,
} from "simplekit/imperative-mode";

// data
let counter = 0;

// user interface

// root container
const root = new SKContainer();
root.id = "root";
root.fill = "whitesmoke";
```

There are no comments in this workspace yet.

How to Architect VS Code with MVC?

no-mvc

- Motivating example with no MV
  - no formal separation of model, view, controller
  - very simple counter

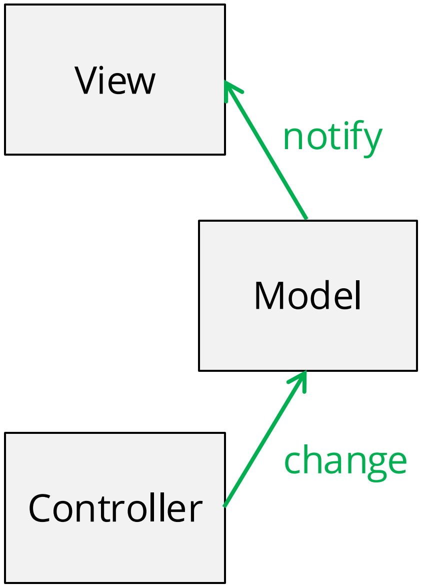


inspired by Joseph Mack: http://www.austintek.com/mvc/

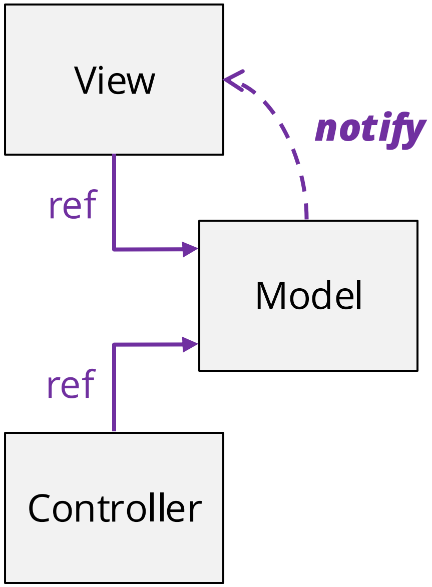
MVC Implementation

Interface architecture decomposed into three parts:

- **Model**: manages application data and logic
- **View**: manages interface to present data
- **Controller**: manages interaction to modify data

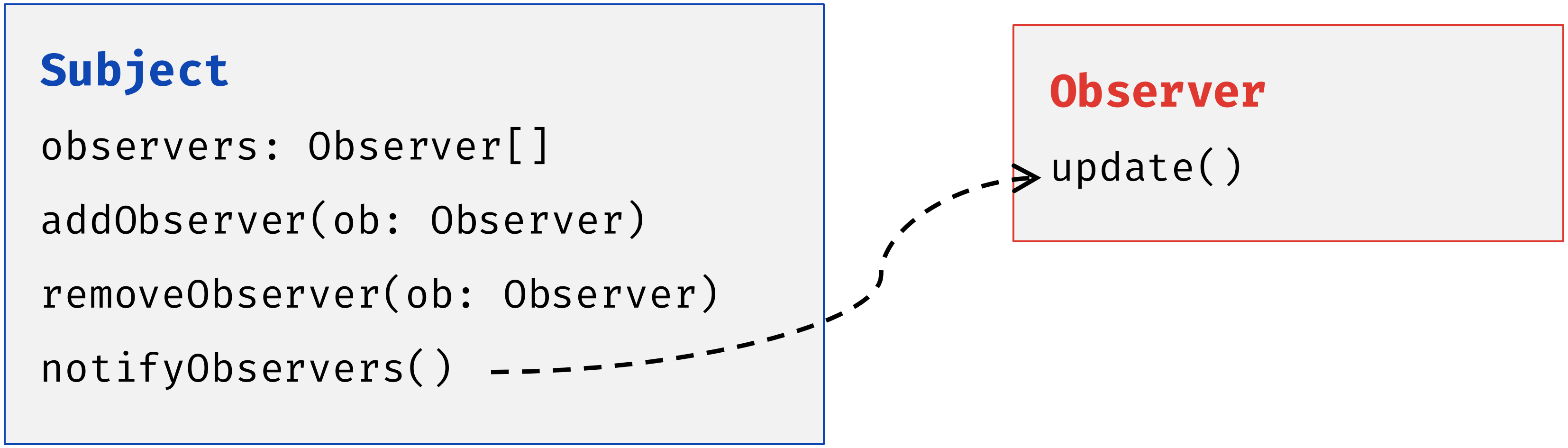


Conceptual



Implementation

Observer Pattern



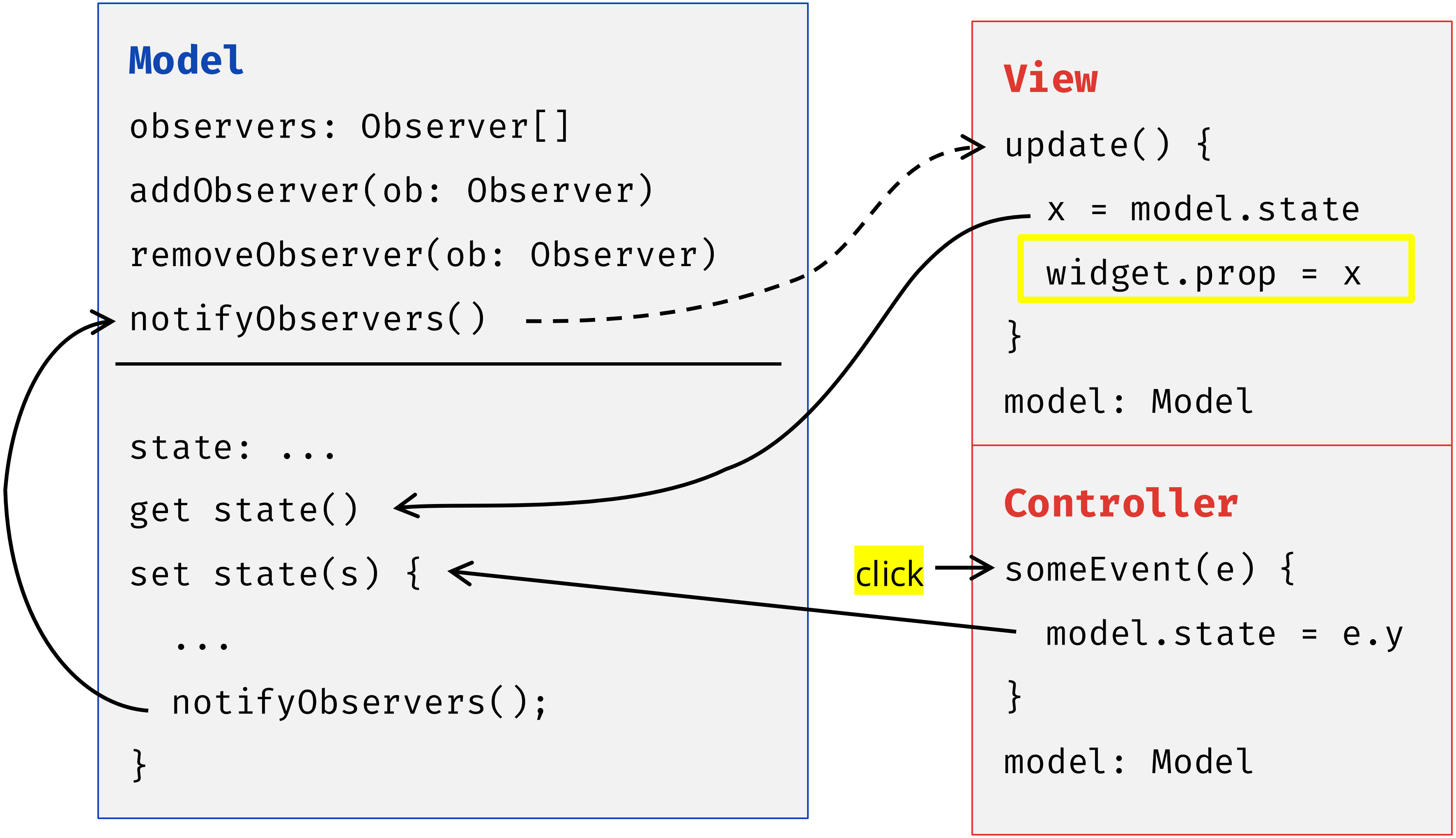
Subject

```
observers: Observer[]
addObserver(ob: Observer)
removeObserver(ob: Observer)
notifyObservers()
```

Observer

```
update()
```

MVC as Observer Pattern



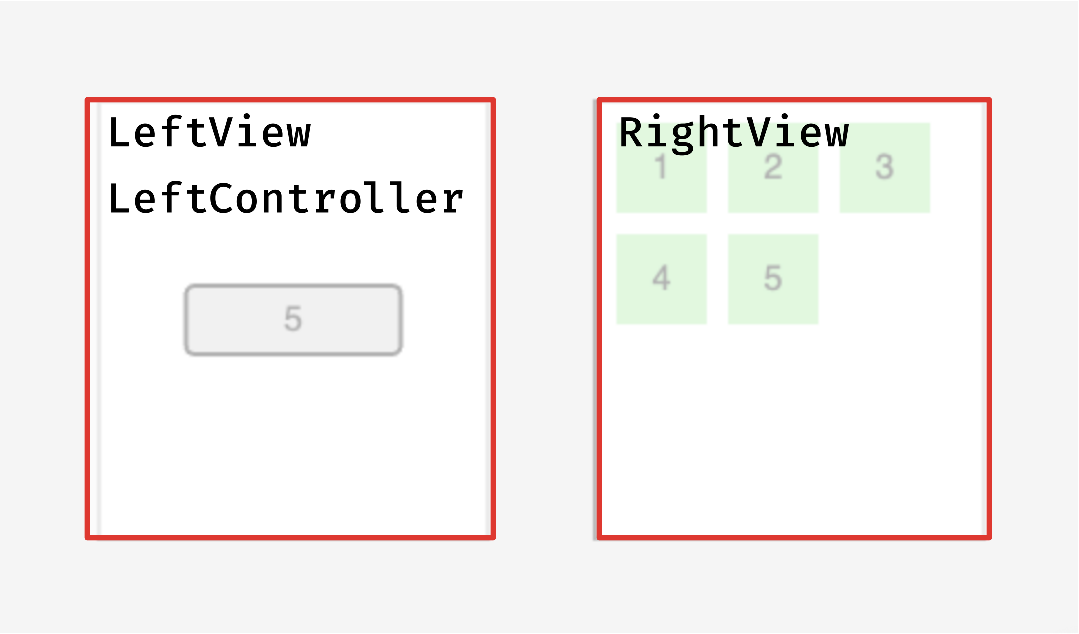
```
Model
observers: Observer[]
addObserver(ob: Observer)
removeObserver(ob: Observer)
notifyObservers()

state: ...
get state()
set state(s) {
  ...
  notifyObservers();
}
```

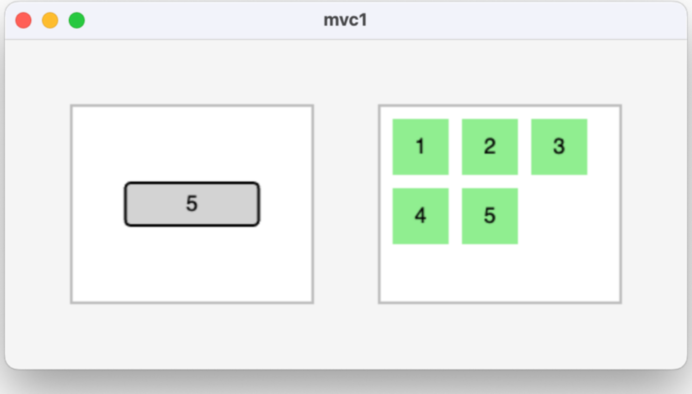
```
View
update() {
  x = model.state
  widget.prop = x
}
model: Model
```

```
Controller
someEvent(e) {
  model.state = e.y
}
model: Model
```

mvc1

- Classic MVC with separate View and Controller





inspired by Joseph Mack: http://www.austintek.com/mvc/

Observer interface and Subject base class

```
export interface Observer {
  update(): void;
}

export class Subject {
  private observers: Observer[] = [];

  protected notifyObservers() {
    for (const o of this.observers) { o.update(); }
  }

  addObserver(observer: Observer) {
    observer.update();
    this.observers.push(observer);
  }

  ...
}
```

single generic update notification

call this every time state changes

first view update

View

```
export class LeftView extends SKContainer implements Observer {

  update(): void {
    this.button.text = `${this.model.count}`;
  }

  button: SKButton = new SKButton({ text: "?" });

  constructor(private model: Model, controller: LeftController) {
    super();

    this.addChild(this.button);

    // set an event handler for button "action" event
    this.button.addEventListener("action", () => {
      controller.handleButtonPress();
    });

    // register with the model when we're ready
    this.model.addObserver(this);
  }
}
```

when model changes

references to model and controller

connect to controller

Controller

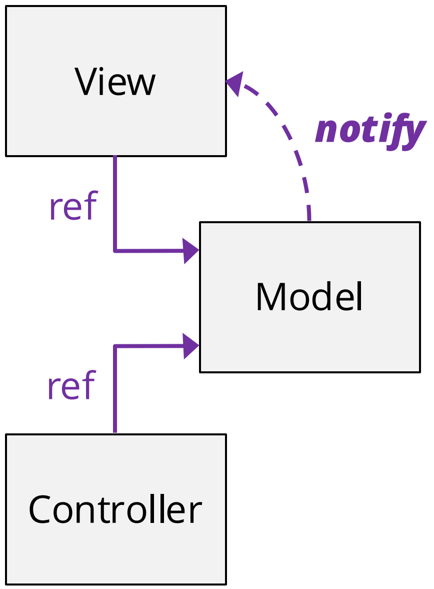
```
export class LeftController {

  constructor(private model: Model) {}

  handleButtonPress() {
    this.model.increment();
  }
}
```

Model

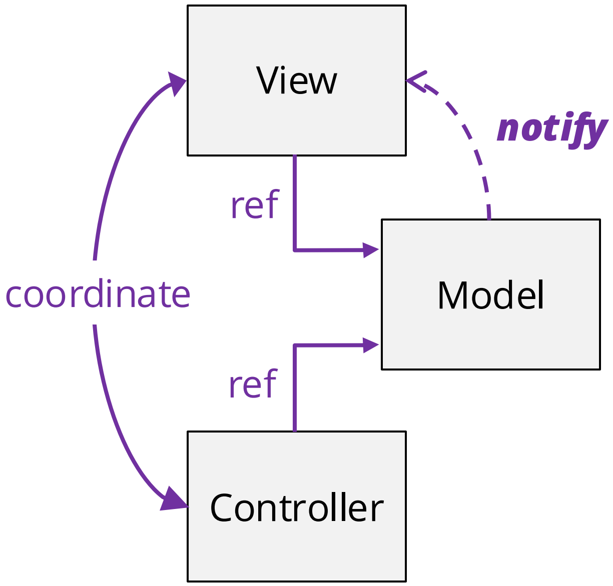
```
export class Model extends Subject {

  // model data (i.e. model state)
  private _count = 0;
  get count() {
    return this._count;
  }

  // model "business logic"
  increment() {
    this._count++;
    // need to notify observers anytime the model changes
    this.notifyObservers();
  }
}
```

called whenever state changes

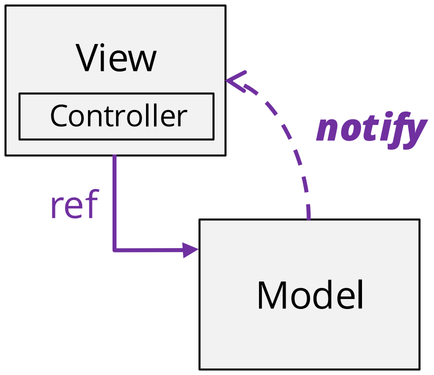
MVC in Theory and Practice



In theory,:
View and Controller ***are loosely coupled***.



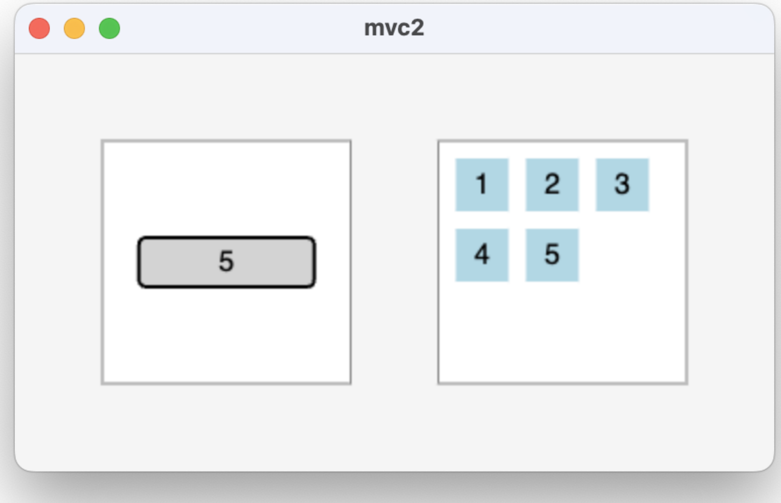
In practice:
View and Controller ***are often tightly coupled***.



Approach:
View integrates the Controller.

mvc2

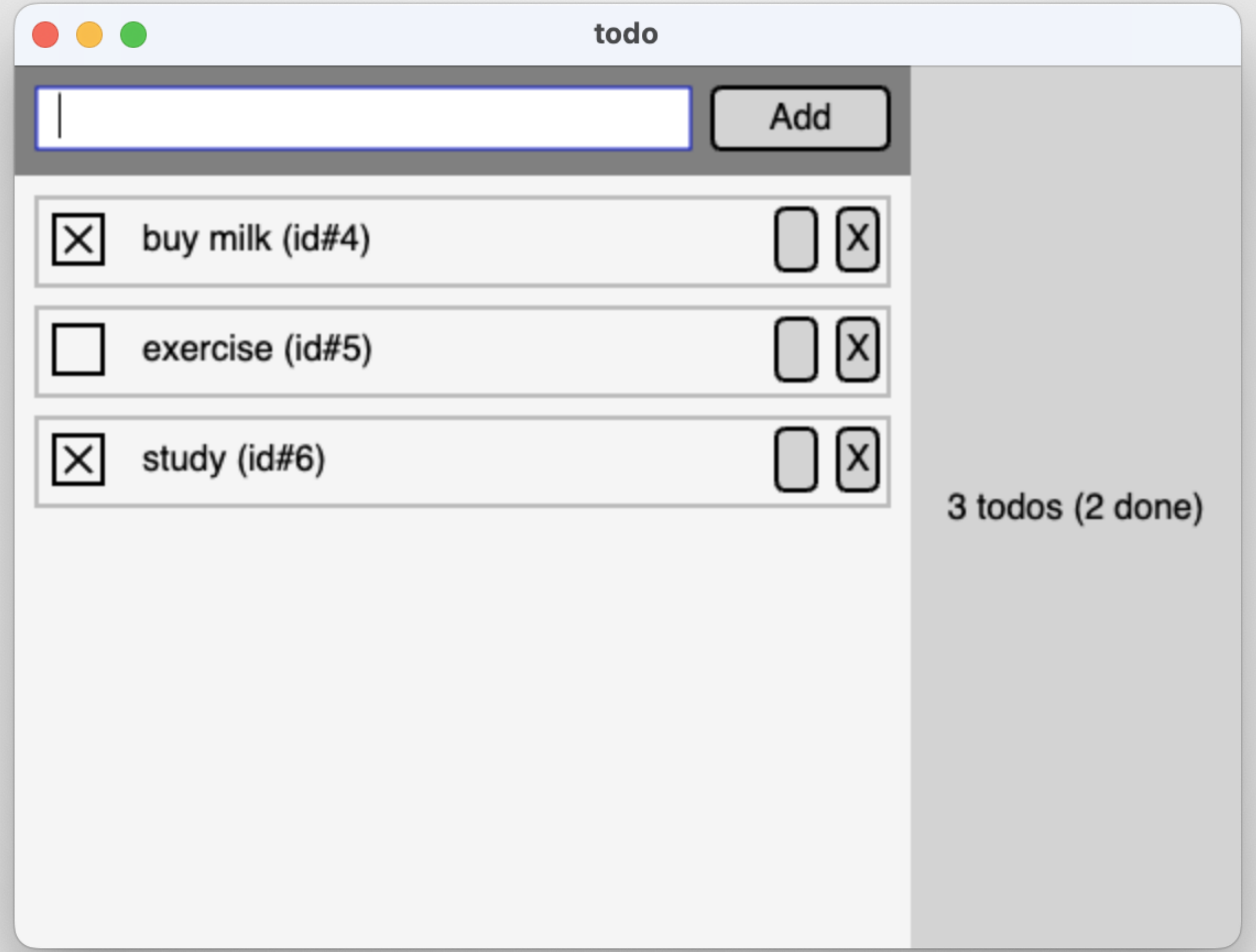
- MVC with Controller *integrated* into View
- This is the most typical MVC approach in practice



View with Integrated Controller

```
export class LeftView extends SKContainer implements Observer {

  update(): void {
    this.button.text = `${this.model.count}`;
  }

  button: SKButton = new SKButton({ text: "?" });

  constructor(private model: Model) {
    super();

    this.addChild(this.button);

    //  Controller
    this.button.addEventListener("action", () => {
       model.increment();
    });

    // register with the model when we're ready
    this.model.addObserver(this);
  }
}
```

this is the controller



☒ buy milk (id#4)

□ exercise (id#5)

☒ study (id#6)

3 todos (2 done)

todo

- **Model**
  - Private array of todos, each is a Todo type with unique id
  - CRUD methods: CUD must notify observers
  - information methods: no need to notify observers
- **FormView**
  - Button and Textfield text changes based on whether a todo is “selected” (selected edits the todo, not selected adds a new todo)
- **ListView**
  - TodoView children; each update clears them and creates new ones
- **InfoView** displays different messages based on model state
- **TodoView** displays a single todo with buttons to edit and delete

todo

- Modify code to immediately update form edits in todo list
  - Uncomment additional controller code in FormView
  - Think about notifications happening each time

- Instrument with debug information to see notifications
  - switch model include to “observer-debug”
  - Uncomment code in main.ts to notifyObservers with Esc key

Optimizing View Updates

- Each viewUpdate, *everything* in every view is refreshed from model
- Could add parameters to viewUpdate to indicate what changed
  - if view knows it isn't affected by change, can ignore it
- **But: simpler is often better**
  - early optimization only introduces extra complexity that causes bugs and adds development time
- Advice: don't worry about efficiency until you have to:
  just update the entire interface

todo

- Add a simple optimization to *only* recreate list of TodoViews when a todo was added or deleted.

MVC Variants

- Model-View-Adaptor
- Model-View-Presenter
- Model-Model-ViewModel

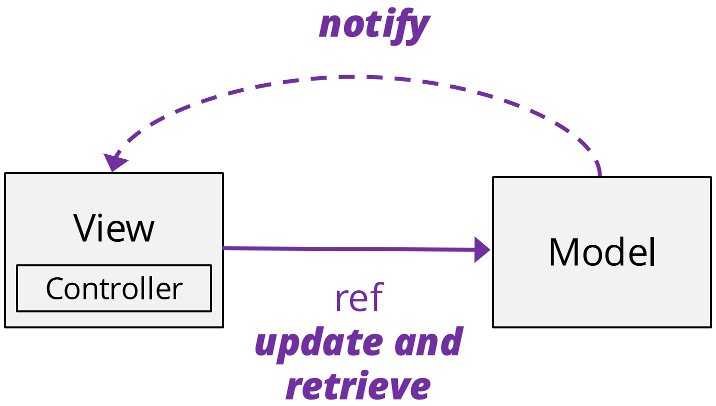
Model-View-Controller (MVC)

Model: manages application data and logic

View: manages interface to present data

Controller: manages interaction to modify data

- common approach is to integrate Controller in the View



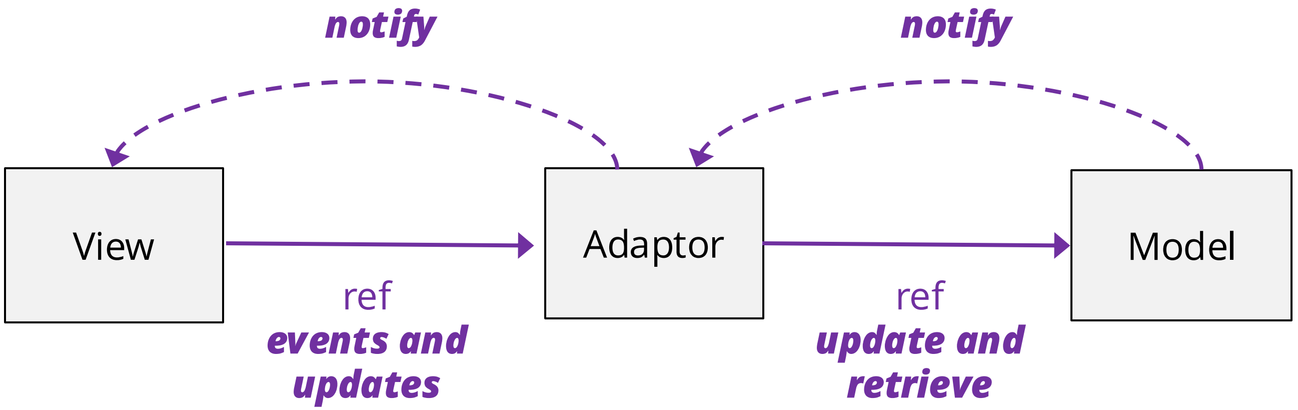
Model-View-Adaptor (MVA)

Model: manages application data and logic.

View: manages interface to present and interact with data.

Adaptor: translates or "adapts" the Model into a form that the View can use.

- An Adaptor can support multiple Views



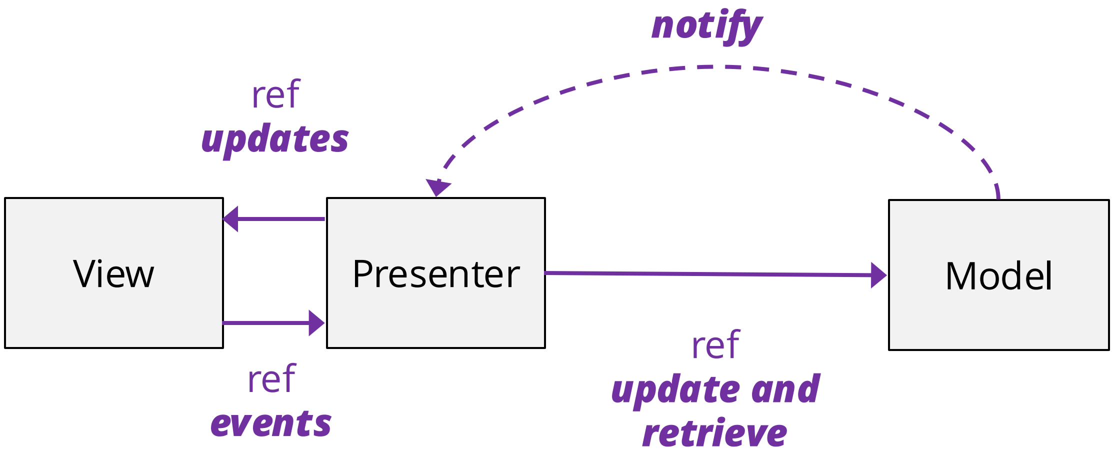
Model-View-Presenter (MVP)

Model: manages application data and its modification.

View: manages interface to present data.

Presenter: middle layer to retrieve data from Model and format it for the View, handles user input and updates Model

- Presenter and View are tightly coupled



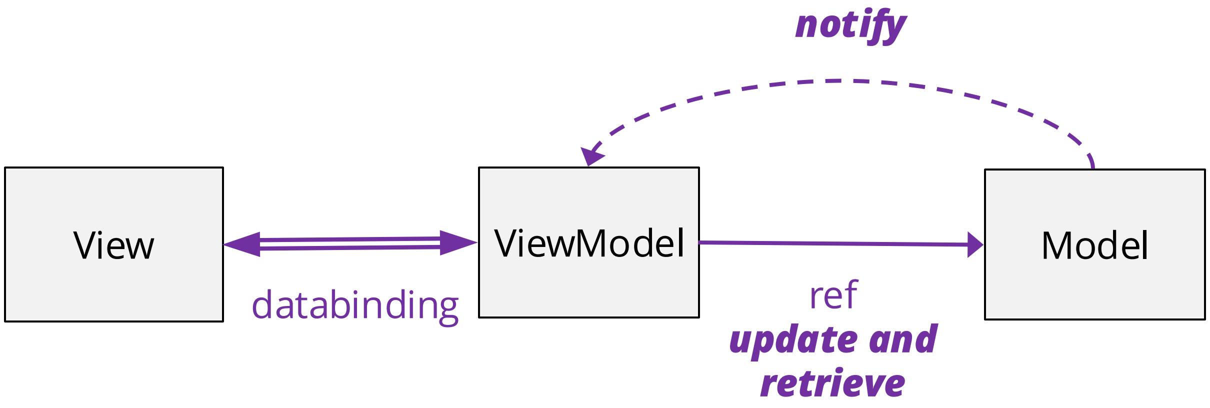
Model-View-ViewModel (MVVM)

Model: manages application data and its modification

View: manages interface to present data.

ViewModel: mediator that exposes data from the Model in a way that's directly usable by the View using *data-binding*

- Data-binding means changes to View automatically trigger changes to Model (and vice-versa)


notify
View
ViewModel
Model
databinding
ref
update and retrieve