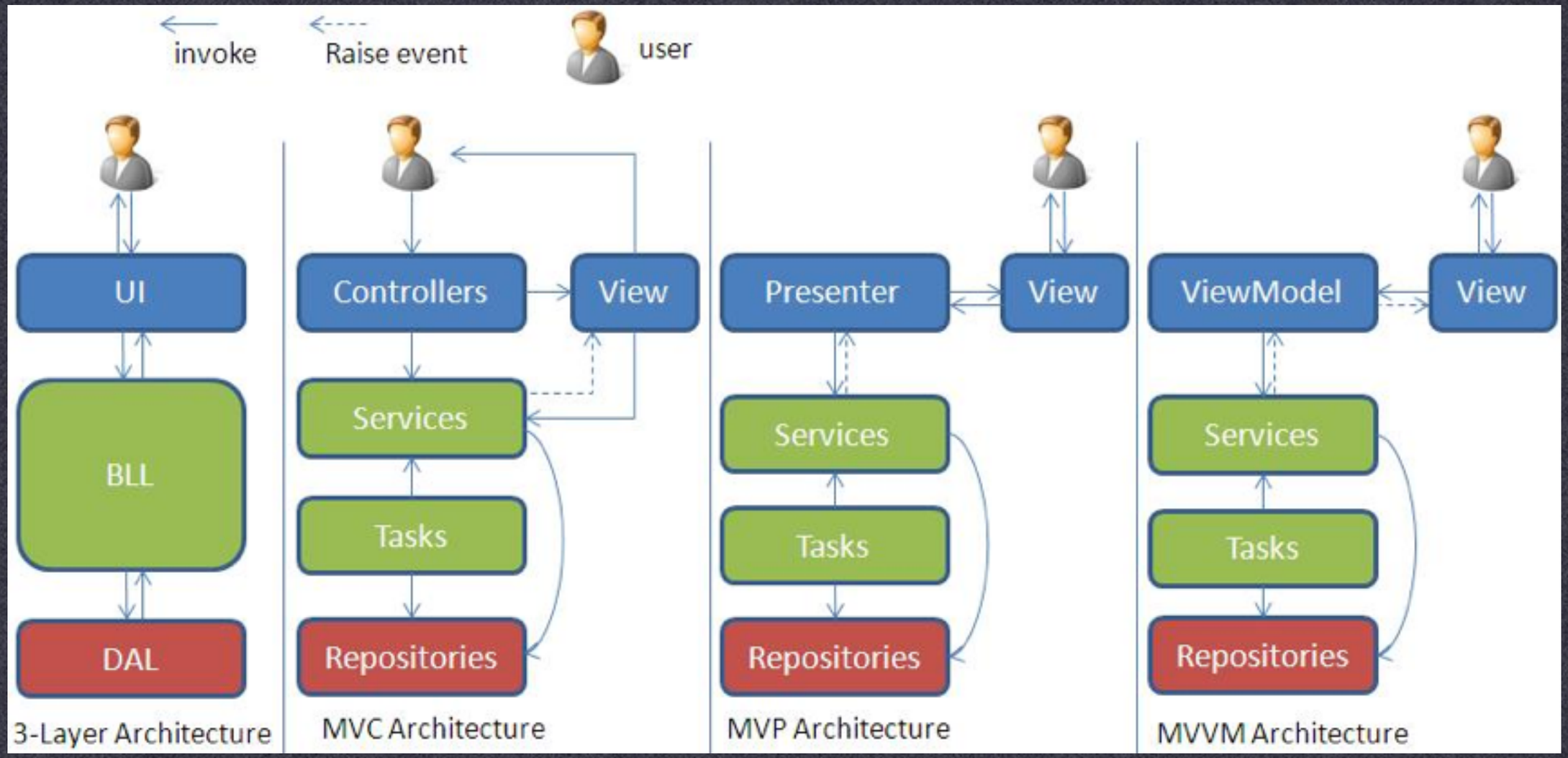


MVC / MVP

Reid Holmes

[Image from: http://merroun.wordpress.com/2012/03/28/mvvm-mvp-and-mvc-software-patterns-againts-3-layered-architecture/]

Background

- MVC started w/ Smalltalk-80
- Java UI frameworks & EJBs reignited interest
- Also prevalent in GWT and .NET development

MVC Motivation

- UI changes more frequently than business logic
  - e.g., layout changes (esp. in web applications)
- The same data is often displayed in different ways
  - e.g., table view vs chart view
  - The same business logic can drive both
- Designers and developers are different people
- Testing UI code is difficult and expensive
- Main Goal: Decouple models and views
  - Increase maintainability/testability of system
  - Permit new views to be developed

Model

- Contains application data
  - This is often persisted to a backing store
- Does not know how to present itself
- Is domain independent
- Are often Subjects in the Observer pattern

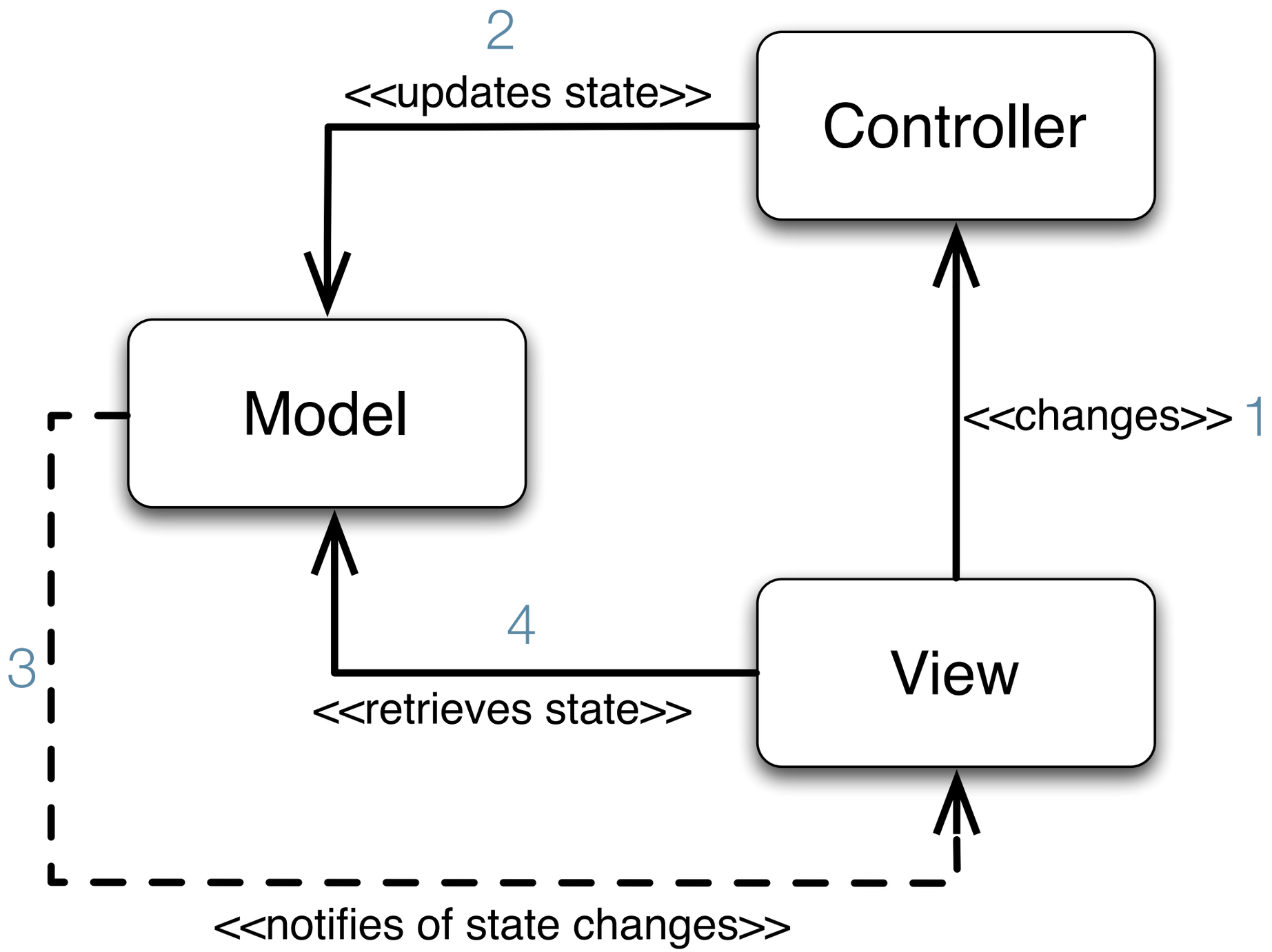
View

- Presents the model to the user
- Allows the user to manipulate the data
- Does not store data
- Is configurable to display different data

Controller

- Glues Model and View together
- Updates the view when the Model changes
- Updates the model when the user manipulates the view
- Houses the application logic
- Loose coupling between Model and others
- View tightly cohesive with its Controller

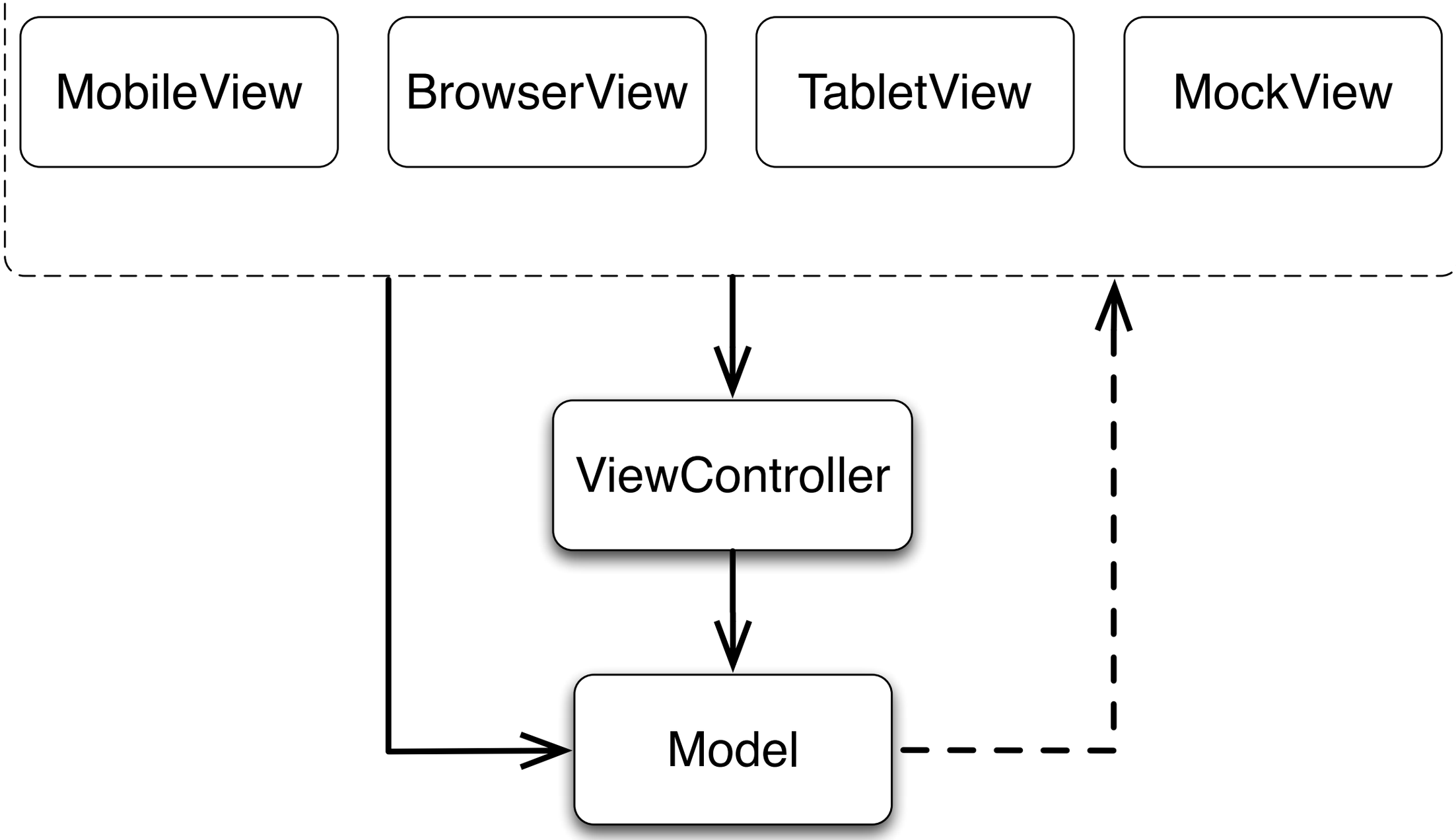
Abstract topology



Concrete topology

```
ViewController c = new ViewController();
IView v = c.createView(c);
IModel m = c.loadModel();
m.addListener(v);
```

[Dependency injection would remove explicit new in c.createView()]



Interaction mechanism

- User interacts with the UI (View)
- UI (View) notifies controller of changes
- Controller handles notifications, processing them into actions that can be performed on the model
- Controller modifies the model as required
- If the model changes, it fires modification events
- The view responds to the modification events

Benefits and tradeoffs

- Pro:
  - Decouple view from model
    - Support multiple views [collaborative views]
    - Maintainability [add new views]
    - Split teams [relieve critical path]
  - Testability [reduce UI testing]
- Con:
  - Complexity [indirection, events]
  - Efficiency [frequent updates, large models]

Compound Pattern

- MVC (and other similar patterns) rely upon several more basic design patterns
- In MVC:
  - View (strategy) / Controller (context) leverage the strategy pattern
  - View is often uses a composite pattern (for nested views)
  - View (observer) / Model (subject) interact through the observer pattern
- Other meta-patterns rely upon similar lower-level design patterns

MVP Motivation

- Take MVC a tiny bit further:
  - Enhance testability
  - Further separate Designers from Developers
- Leveraged by both GWT and .NET

Model

- Contains application data
  - This is often persisted to a backing store
- Does not know how to present itself
- Is domain independent
- Often fires events to an Event Bus

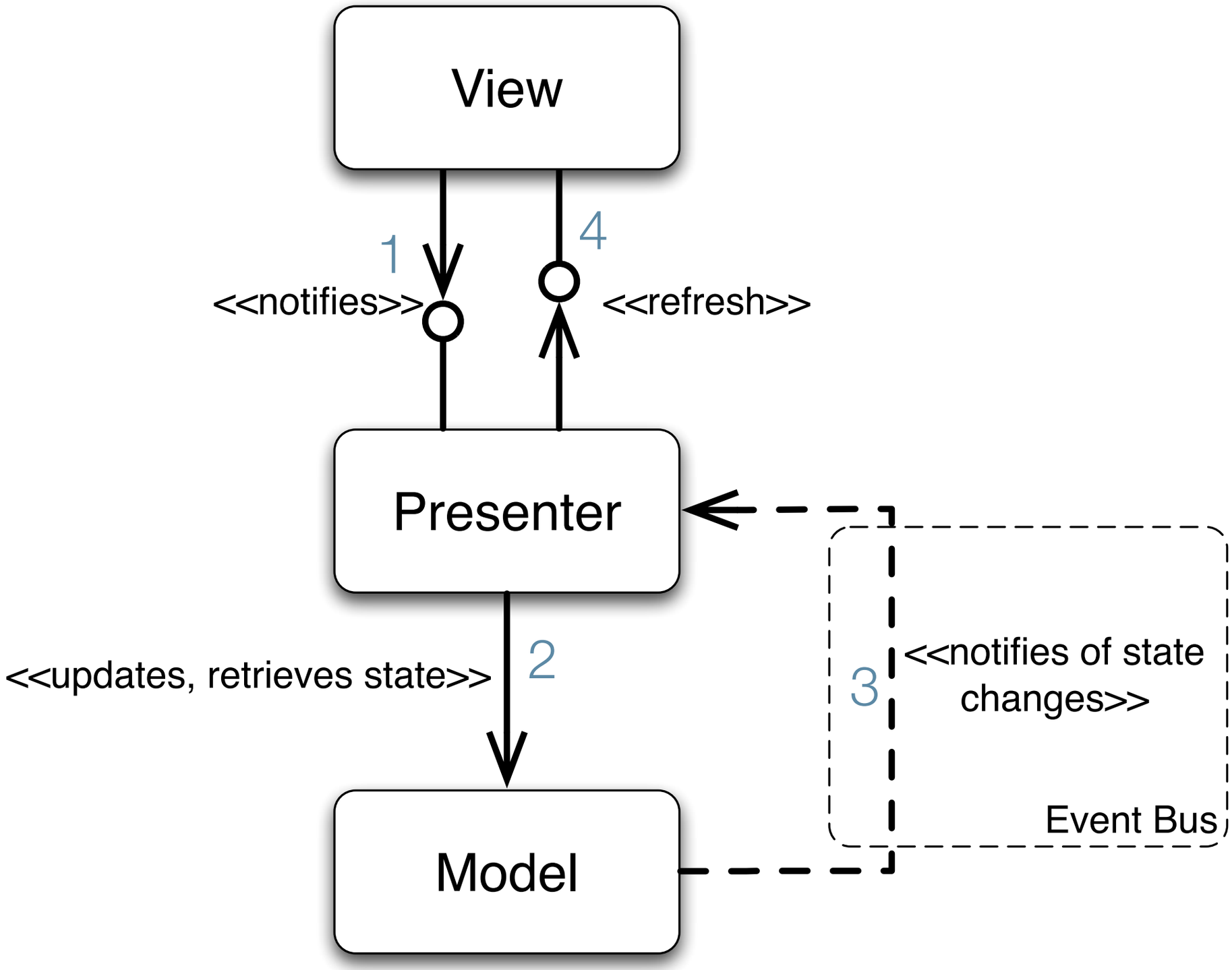
View

- Thin UI front-end for controller
- Does not store data
- Can be interchanged easily
- Does not ever see or manipulate Model objects
- Only interacts with primitives
  - e.g., (setUser(String) instead of setUser(User))

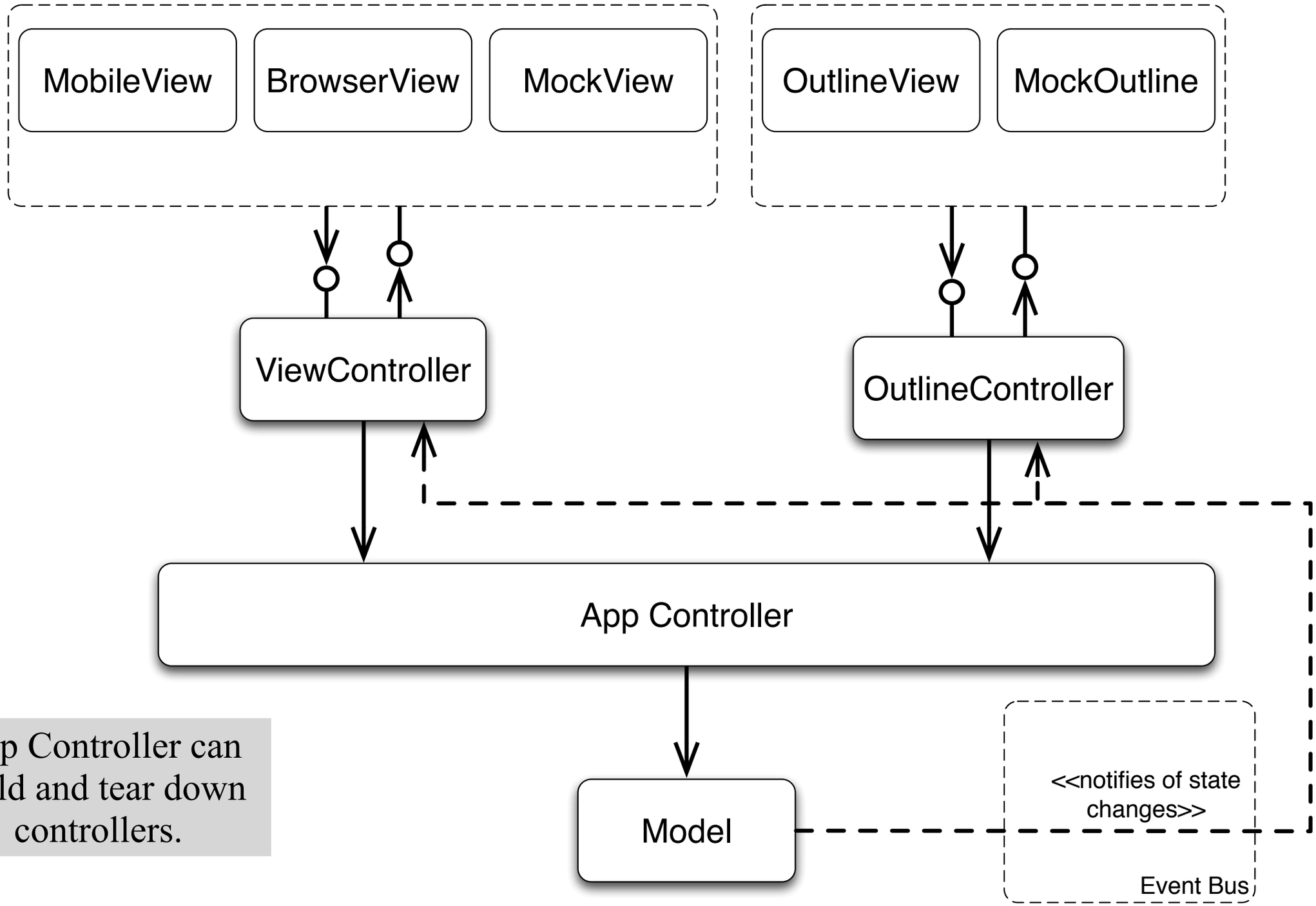
Controller

- Glues Model and View together
- Updates the view when the Model changes
- Updates the model when the user manipulates the view
- Houses the application logic

MVP Topology



Concrete MVP Topology



App Controller can build and tear down controllers.

Concrete Example

```
main(String[] args) {
   AppController ac = new AppController();
   IModel m = ac.setModel(Persist.loadModel());
   m.addListener(ac);
   ac.showMain();
}

AppController::showMain() {
    ViewController vc = new ViewController(this);
   v.showMain();
}
ViewController::ViewController(AppController ac) {
   _controller = ac;
   _controller.getModel().addListener(vc);
   IView v = createView();
   v.setPresenter();
}
```

[Dependency injection should be used in ViewController.createView()]

Views and presenters are tightly bound:

```
public interface IView {
    public void setPresenter(Presenter p);
    public void showMain();

    public interface Presenter {
        void onCancel();
        void onAction(String action);
        public IView createView();
    }
}
```

Benefits and tradeoffs

- Same as MVC with improved:
  - Decoupling of views from the model
    - Split teams [relieve critical path]
  - Testability [reduce UI testing]
  - A little less complex than MVC [fewer events]