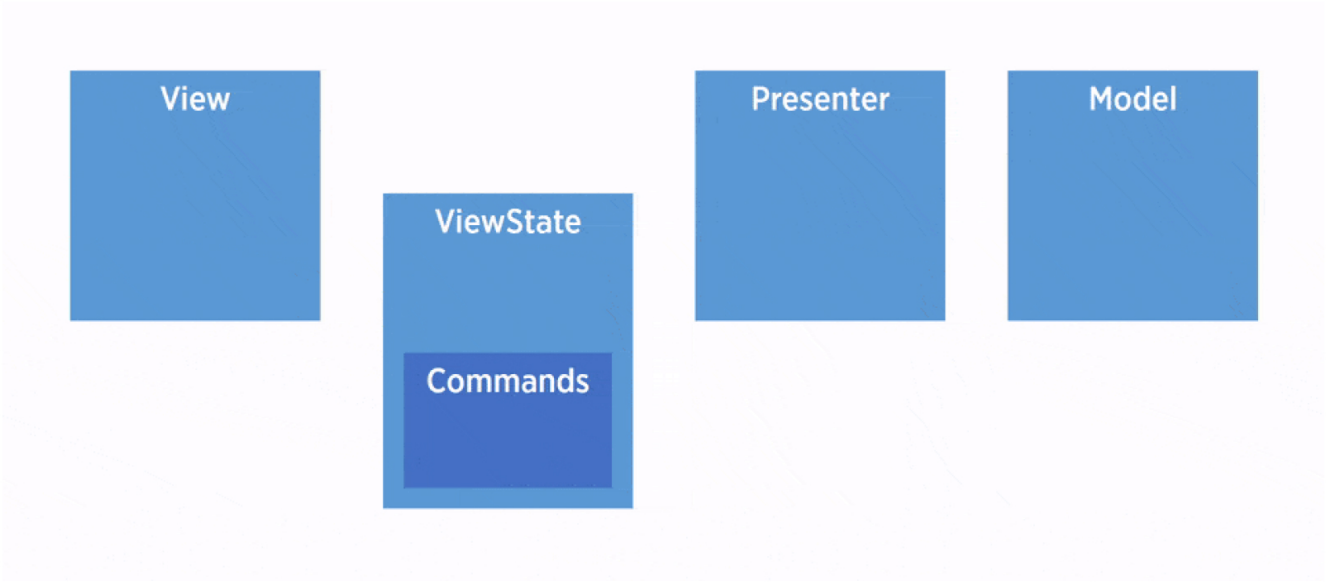
Reactive Apps with Model-View-Intent - Part 1: Model

2. Screen orientation changes



What's happened here:

1. At View happened action orange ▲ , that is passed to Presenter
2. Presenter sends command orange ● to ViewState
3. ViewState adds a command orange ● to commands queue, and then passes it to View
4. View brings itself into a state specified in the command ●
5. Presenter starts async request green ■ to Model
6. Presenter receives result of request ■ from Model
7. Presenter sends command ● to ViewState
8. ViewState adds a command ● to commands queue, remove existing command ● and then passes it to View
9. View brings itself into a state specified in the command ●
10. Presenter sends command ● to ViewState
11. ViewState adds a command ● to commands queue and then passes it to View
12. View brings itself into a state specified in the command ●

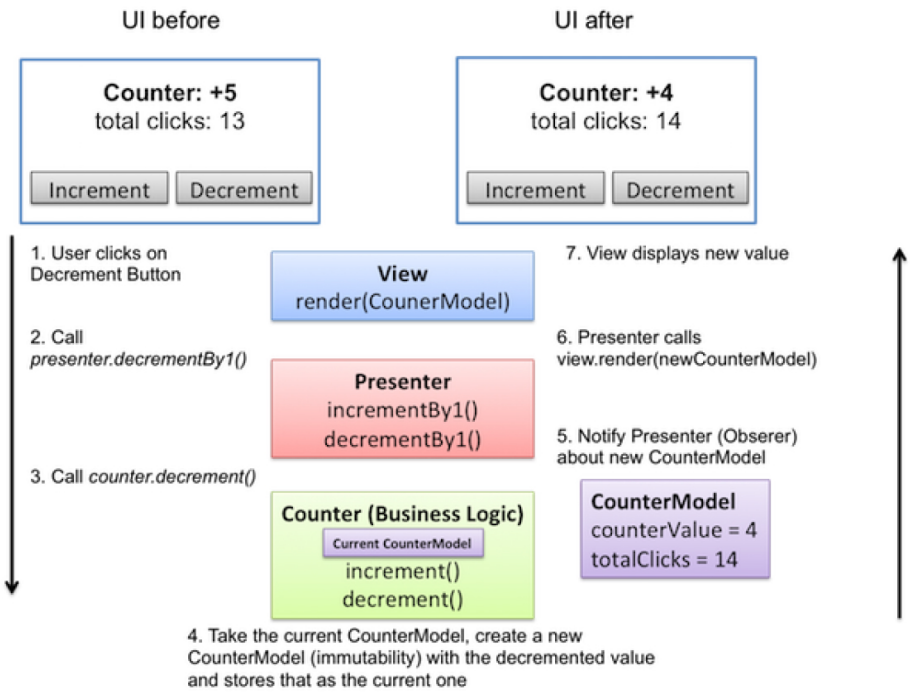
New/recreated View attach to existing Presenter

13. ViewState sends queue of saved commands to new/recreate View
14. New/recreated View brings itself into a state specified in the commands ● and ●

3. Navigation on the back stack

4. Process death

5. Immutability and unidirectional data flow



6. Debuggable and reproducible states

7. Testability

Just load a list of persons from a backend. A “traditional” MVP implementation could look like this:

```
class PersonsPresenter extends Presenter<PersonsView> {

  public void load(){
    getView().showLoading(true); // Displays a ProgressBar on the screen

    backend.loadPersons(new Callback(){
      public void onSuccess(List<Person> persons){
        getView().showPersons(persons); // Displays a list of Persons on the screen
      }

      public void onError(Throwable error){
        getView().showError(error); // Displays a error message on the screen
      }
    });
  }
}
```

From my point of view there should be a “Model” class like this:

```
class PersonsModel {
  // In a real application fields would be private
  // and we would have getters to access them
  final boolean loading;
  final List<Person> persons;
  final Throwable error;

  public(boolean loading, List<Person> persons, Throwable error){
    this.loading = loading;
    this.persons = persons;
    this.error = error;
  }
}
```

And then the Presenter could be implemented like this:

```
class PersonsPresenter extends Presenter<PersonsView> {

  public void load(){
    getView().render( new PersonsModel(true, null, null) ); // Displays a ProgressBar

    backend.loadPersons(new Callback(){
      public void onSuccess(List<Person> persons){
        getView().render( new PersonsModel(false, persons, null) ); // Displays a list of Persons
      }

      public void onError(Throwable error){
          getView().render( new PersonsModel(false, null, error) ); // Displays a error message
      }
    });
  }
}
```

1. The State Problem

```
class PersonsViewModel {
  ObservableBoolean loading;
  // ... Other fields left out for better readability

  public void load(){

    loading.set(true);

    backend.loadPersons(new Callback(){
      public void onSuccess(List<Person> persons){
      loading.set(false);
      // ... other stuff like set list of persons
      }

      public void onError(Throwable error){
        loading.set(false);
        // ... other stuff like set error message
      }
    });
  }
}
```

In MVVM with RxJava we don't use the data binding engine but bind Observable to UI Widgets in the View, for example:

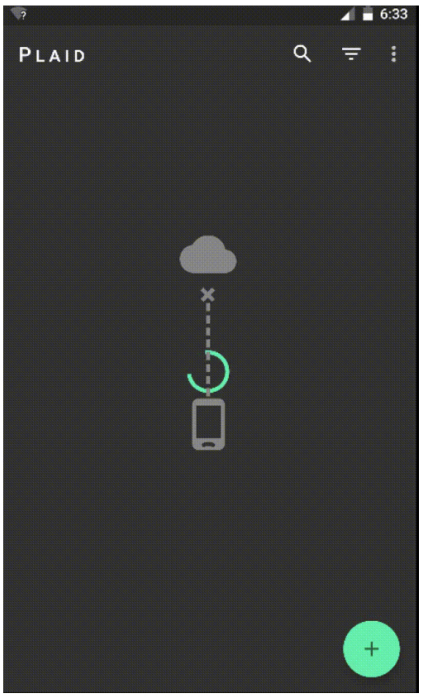
```
  private PublishSubject<List<Person> persons;
  private PublishSubject loadPersonsCommand;

  public RxPersonsViewModel(){
    loadPersonsCommand.flatMap(ignored -> backend.loadPersons())
      .doOnSubscribe(ignored -> loading.onNext(true))
      .doOnTerminate(ignored -> loading.onNext(false))
      .subscribe(persons)
      // Could also be implemented entirely different
  }

  // Subscribed to in View (i.e. Activity / Fragment)
  public Observable<Boolean> loading(){
    return loading;
  }

  // Subscribed to in View (i.e. Activity / Fragment)
  public Observable<List<Person>> persons(){
    return persons;
  }

  // Whenever this action is triggered (calling onNext() ) we load persons
  public PublishSubject loadPersonsCommand(){
    return loadPersonsCommand;
  }
}
```

Of course these code snippets are not perfect and your implementation may look entirely different. The point is that usually in MVP and MVVM the state is driven by either the Presenter or the ViewModel.

This leads to the following observations:

1. The business logic has its own state, the Presenter (or ViewModel) has its own state (and you try to sync the state of business logic and Presenter so that both have the same state) and the View may also have its own state (i.e. you set the visibility somehow directly in the View, or Android itself restores the state from bundle during recreation).
2. A Presenter (or ViewModel) has arbitrarily many inputs (the View triggers an action handled by Presenter) which is ok, but a Presenter also has many outputs (or output channels like view.showLoading() or view.showError() in MVP or ViewModel is offering multiple Observables) which eventually leads to conflicting states of View, Presenter and business logic especially when working with multiple threads.

In the best-case scenario, this only results in visual bugs such as displaying a loading indicator ("loading state") and error indicator ("error state") at the same time like this:



Understanding what a “Model” is and how to model it properly is important, because at the end a Model can solve the “State Problem”

```
class PersonsModel {
  // In a real world application those fields would be private
  // and we would have getters to access them
  final boolean loading;
  final List<Person> persons;
  final Throwable error;

  public(boolean loading, List<Person> persons, Throwable error){
    this.loading = loading;
    this.persons = persons;
    this.error = error;
  }
}
```

Having a “Model” solves a lot of issues we quite often struggle with in Android development:

1. The State Problem
2. Screen orientation changes
3. Navigation on the back stack
4. Process death
5. Immutability and unidirectional data flow
6. Debuggable and reproducible states
7. Testability