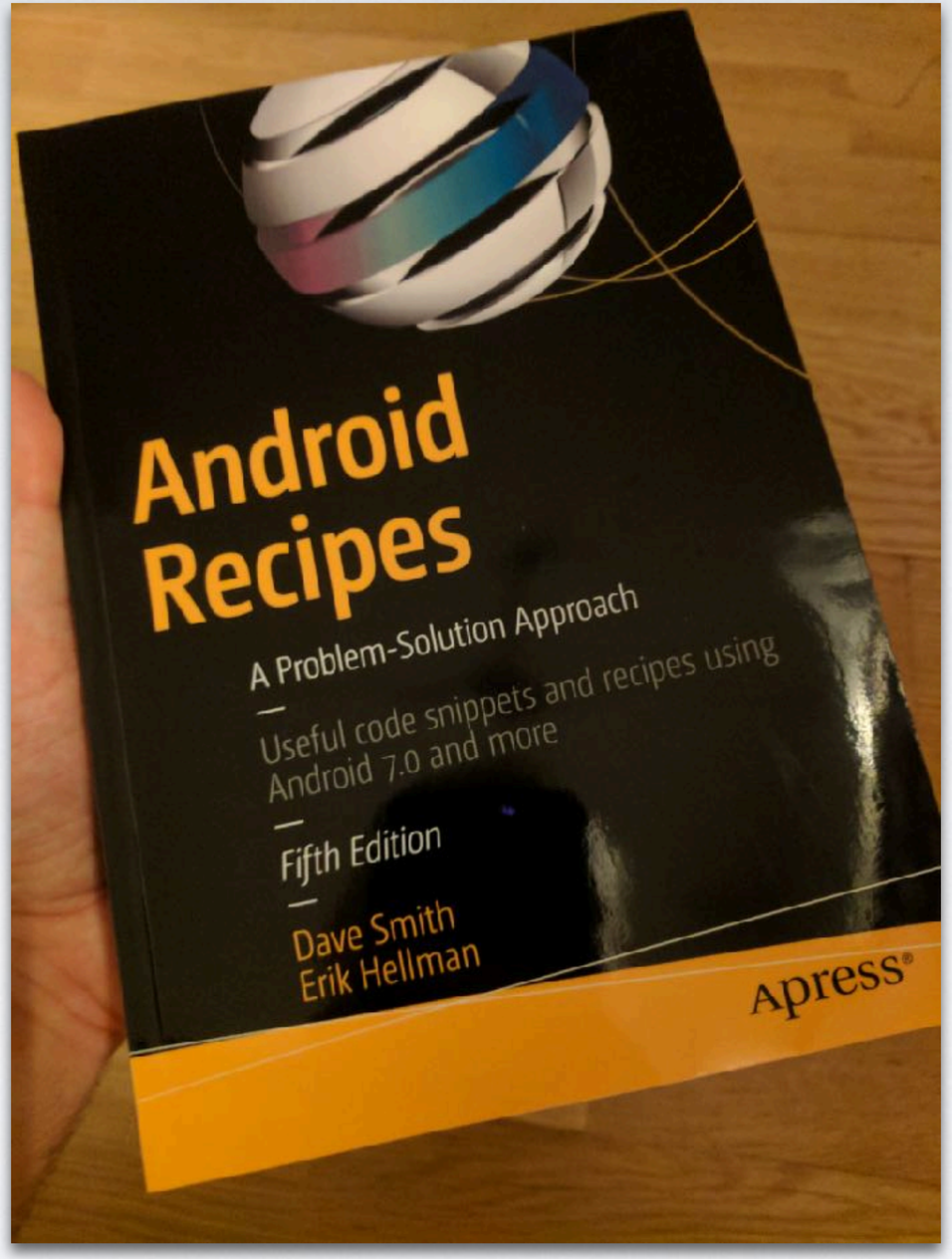
MVVM and RxJava

Living the Hype!

Erik Hellman <erik.hellman@hellsoft.se>

SHAMELESS PLUG!



I HOPE YOU KNOW SOME…

- **Basic RxJava**
- **Dependency Injection (e.g., Dagger 2)**
- **Android Data Binding**

“We often see questions from developers that are asking from the Android platform engineers about the kinds of design patterns and architectures they use in their apps. But the answer, maybe surprisingly, is we often don't have a strong opinion or really an opinion at all.”

–Dianne Hackborn, Google

MVVM?

Model-View-ViewModel

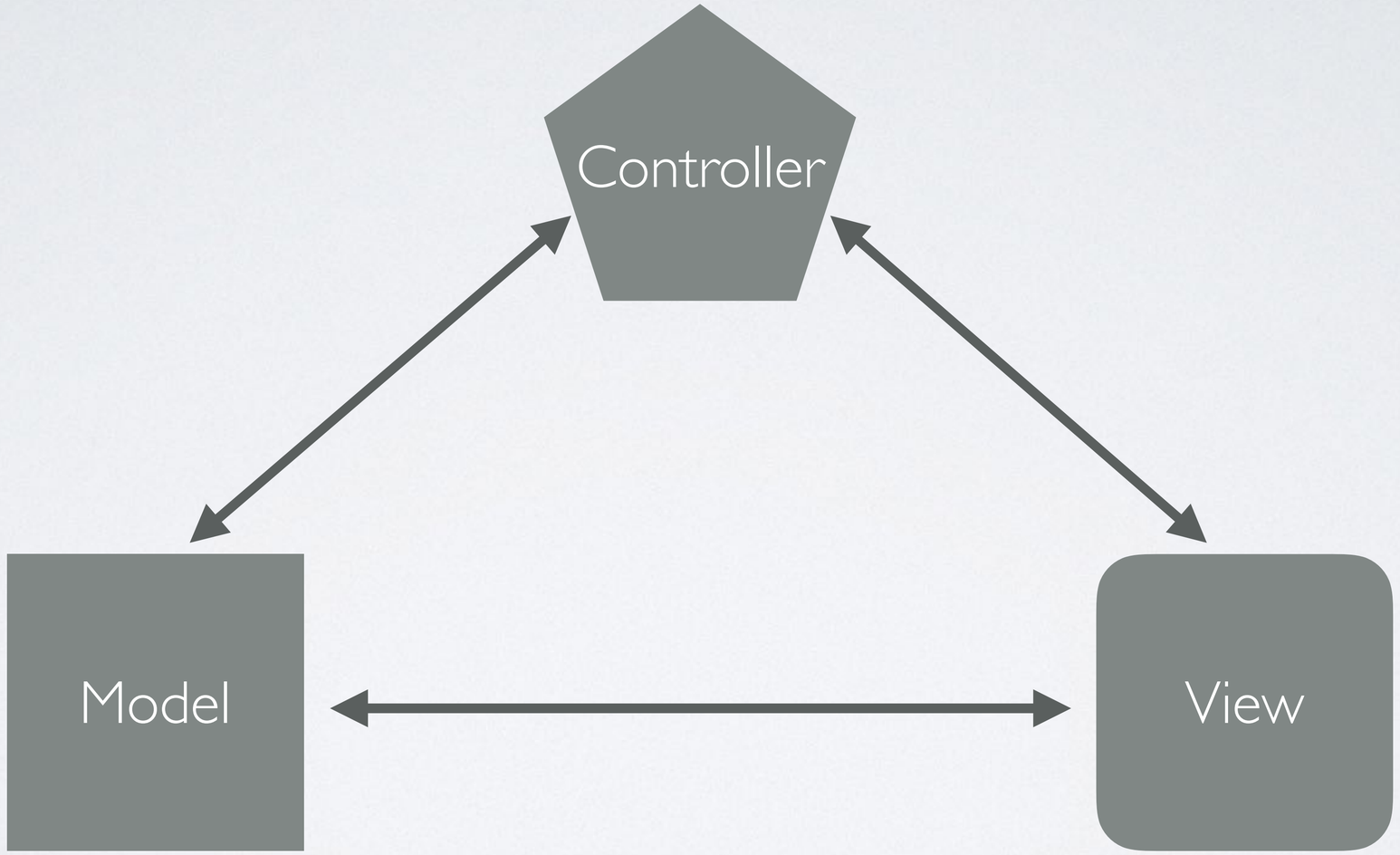
“MVVM was developed by Microsoft architects Ken Cooper and Ted Peters specifically to simplify event-driven programming of user interfaces”

–Wikipedia

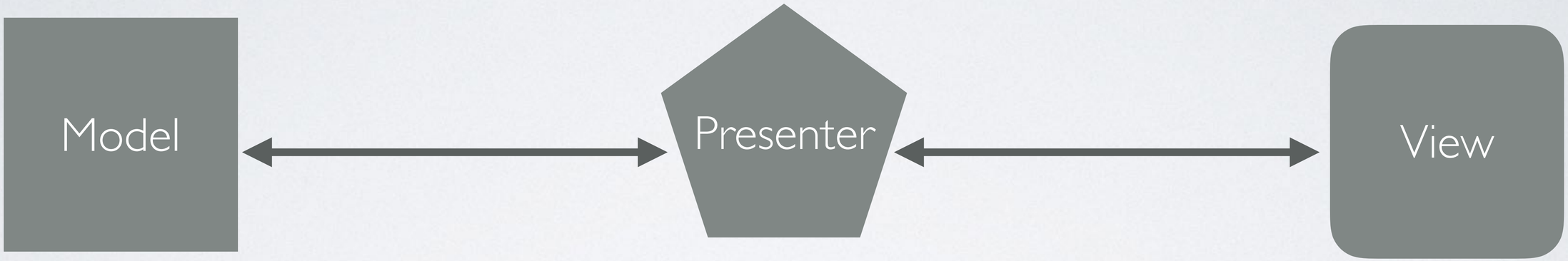
"MVVM was developed by Microsoft architects Ken Cooper and Ted Peters specifically to simplify **event-driven programming** of user interfaces"

–Wikipedia

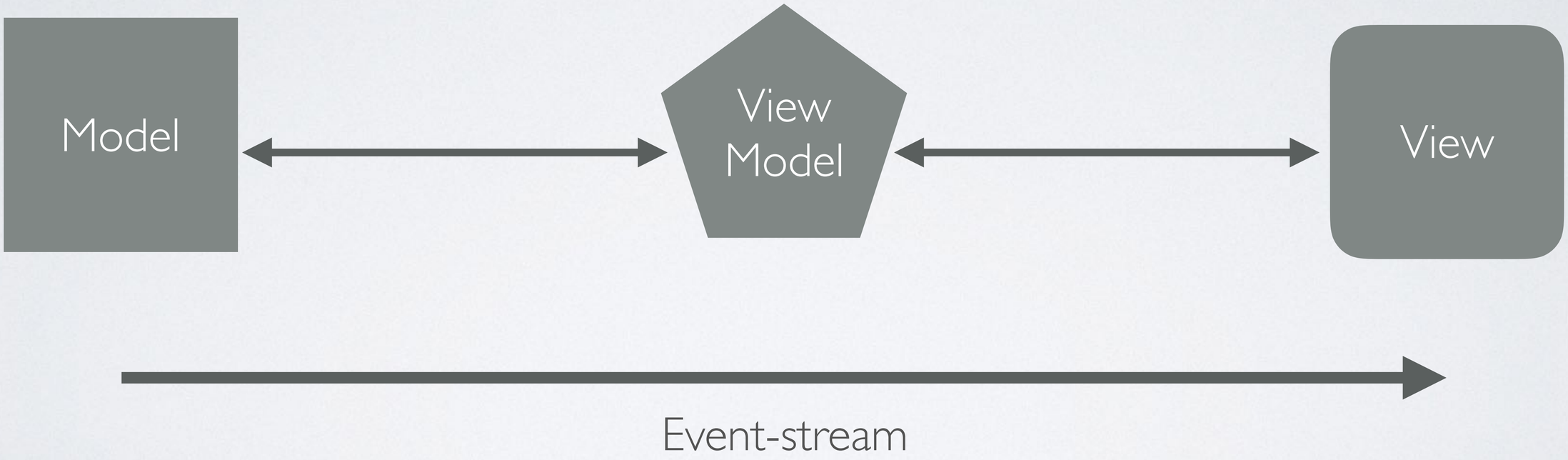
MODEL-VIEW-CONTROLLER



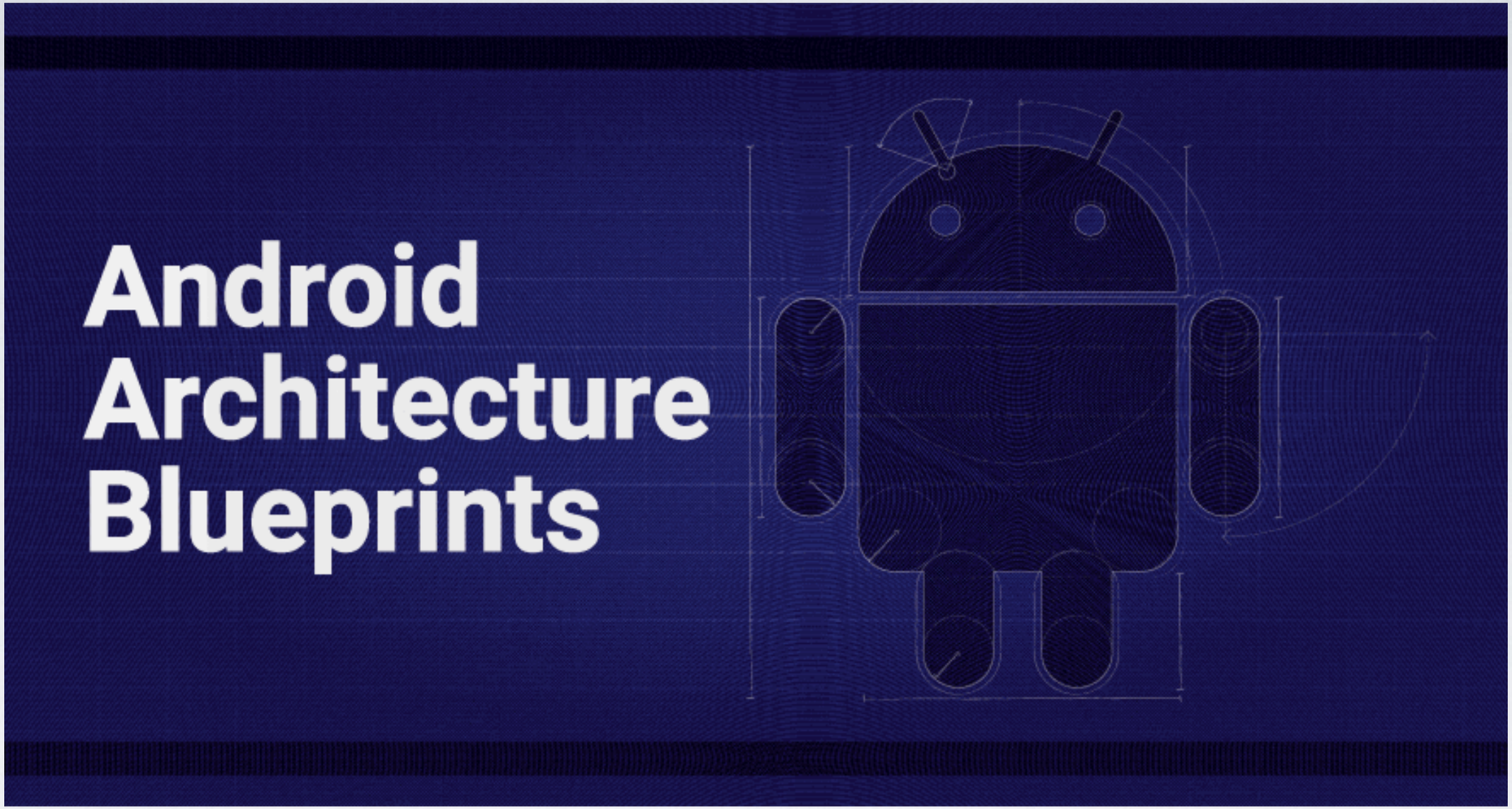
MODEL-VIEW-PRESENTER



MODEL-VIEW-VIEWMODEL



PRESENTER? VIEWMODEL?



https://github.com/googlesamples/android-architecture

PRESENTER

```
public interface TasksContract {

    interface View extends BaseView<Presenter> {

        void setLoadingIndicator(boolean active);

        void showTasks(List<Task> tasks);

        void showAddTask();

        void showTaskDetailsUi(String taskId);

        void showTaskMarkedComplete();

        void showTaskMarkedActive();

        void showCompletedTasksCleared();

        void showLoadingTasksError();
```

PRESENTER

- **Imperative**
- **Single-purposed**
- **Stateful**

VIEW MODEL TO THE RESCUE!

- **Event-Driven**
- **Reactive**
- **Reusable**

USE CASE:THE KITTEN APP

KITTEN DTO

```
public class Kitten {
    public long id;
    public String name;
    public int age;
    public String photoUri;
}
```

FIRST, SOME RX!

```
public interface KittenApi {
    @GET("api/kittens")
    Call<List<Kitten>> fetchKittens();
}
```

FIRST, SOME RX!

```
public interface KittenApi {
    @GET("api/kittens")
    Observable<List<Kitten>> fetchKittens();
}
```

FIRST, SOME RX!

```
public void testRx(KittenApi kittenApi) {
    kittenApi.reactiveFetchKittens()
            .flatMapIterable(kittens -> kittens)
            .doOnNext(this::writeToDatabase)
            .toList()
            .subscribe(listAdapter::setData);
}
```

FIRST, SOME RX!

```
public void testRx(KittenApi kittenApi) {
    kittenApi.reactiveFetchKittens()
            .flatMapIterable(kittens -> kittens)
            .doOnNext(this::writeToDatabase)
            .toList()
            .subscribe(listAdapter::setData,
                throwable -> {
                    listAdapter.showError(R.string.kittenError);
                });
}
```

FIRST, SOME RX!

```
public void testRx(KittenApi kittenApi) {
    kittenApi.reactiveFetchKittens()
            .flatMapIterable(kittens -> kittens)
            .doOnNext(this::writeToDatabase)
            .toList()
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(listAdapter::setData,
                throwable -> {
                    listAdapter.showError(R.string.kittenError);
                });
}
```

FIRST, SOME RX!

```
new Observer<List<Kitten>>() {

    @Override
    public void onSubscribe(Disposable d) {
        // When subscribe happens
    }

    @Override
    public void onNext(List<Kitten> kittens) {
        // For each event
    }

    @Override
    public void onError(Throwable e) {
        // Something went wrong. Terminal event.
    }

    @Override
    public void onComplete() {
        // No more events. Terminal event.
    }
}
```

NEXT, A REPOSITORY!

```
interface KittenRepository {
    Observable<Kitten> fetchKitten(Long id);

    Observable<List<Kitten>> fetchKittens();

    Completable saveKitten(Kitten kitten);

    Completable deleteKitten(Kitten kitten);
}
```

REPOSITORY

```
interface Repository<T> {
    Observable<T> fetchItem(Long id);

    Observable<List<T>> fetchItems();

    Completable saveItem(T item);

    Completable deleteItem(T item);
}
```

REPOSITORY FACADE

```
public class KittenRepository implements Repository<Kitten> {
    @Inject
    @Named("localKittens")
    private Repository<Kitten> localRepository;

    @Inject
    @Named("remoteKittens")
    private Repository<Kitten> remoteRepository;

    @Override
    public Observable<Kitten> fetchItem(Long id) {
        return localRepository.fetchItem(id);
    }

    @Override
    public Observable<List<Kitten>> fetchItems() {
        return localRepository.fetchItems();
    }
```

REPOSITORY FACADE

```
    @Override
    public Completable saveItem(Kitten kitten) {
        return remoteRepository
                .saveItem(kitten)
                .concatWith(localRepository.saveItem(kitten));
    }

    @Override
    public Completable deleteItem(Kitten kitten) {
        return remoteRepository
                .deleteItem(kitten)
                .concatWith(localRepository.deleteItem(kitten));
    }
}
```

LOCAL REPOSITORY

```
public class LocalKittenRepository implements Repository<Kitten> {
    @Inject
    @Named("kittenDatabase")
    BriteDatabase kittenDatabase;

    @Override
    public Observable<Kitten> fetchItem(Long id) {
        rx.Observable<Kitten> rxObservable = kittenDatabase
                .createQuery("kitten",
                        "SELECT * FROM kitten WHERE id = %s", id.toString())
                .mapToOne(this::cursorToKitten);

        return RxJavaInterop.toV2Observable(rxObservable);
    }
```

LOCAL REPOSITORY

```
@Override
public Observable<List<Kitten>> fetchItems() {
    rx.Observable<List<Kitten>> rxObservable = kittenDatabase
            .createQuery("kitten", "SELECT * FROM kitten")
            .mapToList(this::cursorToKitten);

    return RxJavaInterop.toV2Observable(rxObservable);
}
```

LOCAL REPOSITORY

```
    @Override
    public Completable saveItem(Kitten kitten) {
        return Completable.fromCallable(() -> {
            return kittenDatabase.insert("kitten",
                    kittenToValues(kitten));
        });
    }

    @Override
    public Completable deleteItem(Kitten item) {
        return Completable.fromCallable(() -> {
            return kittenDatabase.delete("kitten", "id = %s",
                    String.valueOf(item.id));
        });
    }
}
```

POPULATING LOCAL REPO?

DATA SYNC

```
public interface KittenDataSync {
    Completable syncKittens();
}
```

DATA SYNC

```
@Override
public Completable syncKittens() {
    return remoteRepository.fetchItems()
            .flatMapIterable(kittens -> kittens)
            .flatMap(kitten -> localRepository
                    .saveItem(kitten).toObservable())
            .toList()
            .toCompletable();
}
```

DATA SYNC

```
public class KittenSyncService extends JobService {
    @Inject
    KittenDataSync kittenDataSync;

    @Override
    public boolean onStartJob(JobParameters params) {
        kittenDataSync
                .syncKittens()
                .subscribeOn(Schedulers.io())
                .subscribe(() -> jobFinished(params, false));

        return true;
    }

    @Override
    public boolean onStopJob(JobParameters params) {
        return true;
    }
}
```

DATA SYNC

```
@Override
public boolean onStartJob(JobParameters params) {
    kittenDataSync
            .syncKittens()
            .subscribeOn(Schedulers.io())
            .subscribe(() -> jobFinished(params, false));

    return true;
}
```

THE VIEWMODEL

VIEWMODEL

```
public class KittenViewModel  {
    @Inject
    KittenRepository repository;
    private Observable<Kitten> kittenObservable;

    public KittenViewModel(Long kittenId) {
        kittenObservable = repository.fetchItem(kittenId);
    }

    public Observable<Kitten> kitten() {
        return kittenObservable;
    }
}
```

VIEWMODEL

```
protected void onStart() {
    super.onStart();
    kittenViewModel.kitten()
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(kitten -> {
                TextView kittenName = (TextView)
findViewById(R.id.kitten_name);
                kittenName.setText(kitten.name);
                // Please stop!
            });
}
```

VIEWMODEL

```
public class KittenViewModel extends BaseObservable {
    @Inject
    KittenRepository repository;

    private Kitten kitten;

    public KittenViewModel(Long kittenId) {
        repository.fetchItem(kittenId)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(k -> {
                    kitten = k;
                    notifyPropertyChanged(BR.kitten);
                });
    }

    @Bindable
    public Kitten getKitten() {
        return kitten;
    }
}
```

VIEWMODEL

```
<TextView
    android:id="@+id/kitten_name"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{viewModel.kitten.name}"/>
```

THAT VIEWMODEL SEEMS UNNECESSARY…

RECYCLERVIEW AND DIFFUTILS

```
public static class KittenDiffCallback extends DiffUtil.Callback {
    private final List<Kitten> oldList;
    private final List<Kitten> newList;

    public KittenDiffCallback(List<Kitten> oldList, List<Kitten> newList) {
        this.oldList = oldList;
        this.newList = newList;
    }

    public int getOldListSize() { return oldList.size(); }

    public int getNewListSize() { return newList.size(); }

    public boolean areItemsTheSame(int oldItemPosition, int newItemPosition) {
        Kitten oldKitten = oldList.get(oldItemPosition);
        Kitten newKitten = newList.get(newItemPosition);
        return oldKitten.id == newKitten.id;
    }

    public boolean areContentsTheSame(int oldItemPosition, int newItemPosition) {
        Kitten oldKitten = oldList.get(oldItemPosition);
        Kitten newKitten = newList.get(newItemPosition);
        return oldKitten.equals(newKitten);
    }
}
```

RECYCLERVIEW AND DIFFUTILS

```
public void subscribe() {
    Observable<List<Kitten>> source = repository.fetchItems();
    Observable<List<Kitten>> startWithEmpty
            = source.startWith(new ArrayList<Kitten>());

    disposable = Observable.zip(startWithEmpty, source,
            (oldList, newList) -> {
                KittenDiffCallback callback
                        = new KittenDiffCallback(oldList, newList);
                DiffUtil.DiffResult diffResult = DiffUtil.calculateDiff(callback);
                return Pair.create(newList, diffResult);
            })
            .subscribeOn(io())
            .observeOn(mainThread())
            .subscribe(listDiffResultPair -> {
                List<Kitten> kittens = listDiffResultPair.first;
                DiffUtil.DiffResult diffResult = listDiffResultPair.second;
                kittensAdapter.setKittens(kittens);
                diffResult.dispatchUpdatesTo(kittensAdapter);
            });
}
```

RECYCLERVIEW AND DIFFUTILS

```
Observable<List<Kitten>> source = repository.fetchItems();
// Same Observable, but start with empty list
Observable<List<Kitten>> startWithEmpty
        = source.startWith(new ArrayList<Kitten>());

// Take latest item from each Observable and apply the function
disposable = Observable.zip(startWithEmpty, source,
        (oldList, newList) -> {
            KittenDiffCallback callback
                    = new KittenDiffCallback(oldList, newList);
            DiffUtil.DiffResult diffResult = DiffUtil.calculateDiff(callback);
            // Combine the new list with the relevant DiffResult
            return Pair.create(newList, diffResult);
        })
```

WHAT ABOUT LIFECYCLES?

SUBSCRIBE/UNSUBSCRIBE

```
private void subscribeToKitten() {
    subscription = kittenObservable
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(k -> {
                kitten = k;
                notifyChange();
            });
}

private void unsubscribeFromKitten() {
    subscription.dispose();
    subscription = null;
}
```

SUBSCRIBE/UNSUBSCRIBE

```
@Override
public void addOnPropertyChangedCallback(OnPropertyChangedCallback c) {
    super.addOnPropertyChangedCallback(c);
    if (subscribersCount.incrementAndGet() == 1) {
        subscribeToKitten();
    }
}

@Override
public void removeOnPropertyChangedCallback(OnPropertyChangedCallback c) {
    super.removeOnPropertyChangedCallback(c);
    if (subscribersCount.decrementAndGet() == 0) {
        unsubscribeFromKitten();
    }
}
```

RXNAVI

```
public class KittenActivity extends NaviAppCompatActivity {

    @Inject
    KittenViewModel viewModel;

    @Override
    protected void onStart() {
        super.onStart();
        viewModel.subscribe(RxNavi.observe(this, Event.STOP));
    }
}
```

RXNAVI

```
public void subscribe(Observable<Object> onStopObservable) {
    kittenObservable
            .takeUntil(onStopObservable)
            .subscribeOn(io())
            .observeOn(mainThread())
            .subscribe(k -> {
                kitten = k;
                notifyChange();
            });
}
```

CONTROL UI STATE WITH RX

```
<Button
    android:layout_width="wrap_content"
    android:layout_height=“wrap_content"
    android:text="@string/label_save"
    android:enabled="@{viewModel.enableSaveButton}"
    android:onClick="@{viewModel.saveKitten()}"/>
```

CONTROL UI STATE WITH RX

```
public ObservableBoolean enableSaveButton = new ObservableBoolean(true);

public void saveKitten() {
    repository.saveItem(kitten)
            .subscribeOn(io())
            .observeOn(mainThread())
            .doOnSubscribe(d -> enableSaveButton.set(false))
            .subscribe(() -> enableSaveButton.set(true));
}
```

TESTING RXJAVA

```
testScheduler = new TestScheduler();
RxAndroidPlugins.setInitMainThreadSchedulerHandler(c -> testScheduler);
RxJavaPlugins.setInitIoSchedulerHandler(c -> testScheduler);
```

TESTING THE VIEWMODEL

```
@Before
public void setUp() {
    kittenRepository = new KittenRepository();
    fakeKittenRepository = new FakeKittenRepository();
    kittenRepository.localRepository = fakeKittenRepository;

    testScheduler = new TestScheduler();
    RxAndroidPlugins.setInitMainThreadSchedulerHandler(c -> testScheduler);
    RxJavaPlugins.setInitIoSchedulerHandler(c -> testScheduler);
}

@After
public void tearDown() {
    RxAndroidPlugins.reset();
    RxJavaPlugins.reset();
}
```

TESTING THE VIEWMODEL

```
static class FakeKittenRepository implements Repository<Kitten> {
    Kitten kitten;

    FakeKittenRepository() {
        kitten = new Kitten();
        kitten.name = "Cookie";
        kitten.color = Color.BLACK;
        kitten.age = 5;
        kitten.photoUri = "http://fake.uri/with/path";
    }

    public Observable<Kitten> fetchItem(Long id) {
        return Observable.just(kitten);
    }
}
```

TESTING THE VIEWMODEL

```
@Test
public void subscribeTriggersDataBindingUpdate() {
    KittenViewModel kittenViewModel = new KittenViewModel(100l);
    kittenViewModel.repository = kittenRepository;
    AtomicBoolean wasCalled = new AtomicBoolean(false);
    android.databinding.Observable.OnPropertyChangedCallback callback
            = new android.databinding.Observable.OnPropertyChangedCallback() {
        @Override
        public void onPropertyChanged(android.databinding.Observable o, int i) {
            wasCalled.set(true);
            Kitten kitten = kittenViewModel.getKitten();
            assertEquals(fakeKittenRepository.kitten, kitten);
        }
    };

    kittenViewModel.addOnPropertyChangedCallback(callback);


    kittenViewModel.subscribe(Observable.never());
    testScheduler.triggerActions();
    assertTrue(wasCalled.get());
}
```

https://github.com/googlesamples/android-architecture

THANK YOU FOR LISTENING!