Poor Programming Patterns and How to Avoid Them

Alice Yuan
Android Engineer at Pinterest
me@aliceyuan.ca
@Names_Alice

Pinterest-Lite Demo App

github.com/AliceYuan/DroidConDemo

github.com/AliceYuan/DroidConDemo

AliceYuan / DroidConDemo

Watch 1 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Insights

Overview | Active | Stale | All branches

Search branches...

Active branches

problem4_solution	Updated 11 hours ago by Alice Yuan	0 | 10	Compare
final_solution	Updated 11 hours ago by Alice Yuan	0 | 10	Compare
problem3_solution	Updated 7 days ago by Alice Yuan	0 | 8	Compare
problem2_solution	Updated 10 days ago by Alice Yuan	0 | 3	Compare
problem1_solution	Updated 10 days ago by Alice Yuan	0 | 2	Compare
bad_code	Updated 29 days ago by Alice Yuan	0 | 1	Compare

github.com/AliceYuan/DroidConDemo

AliceYuan / DroidConDemo

Watch 1 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Insights

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

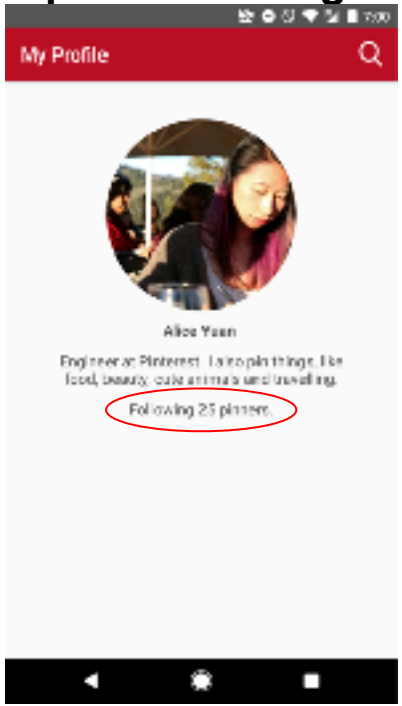
base: bad_code ... compare: problem1_solution ✓ Able to merge. These branches can be automatically merged.

1 commit | 9 files changed | 0 commit comments | 1 contributor

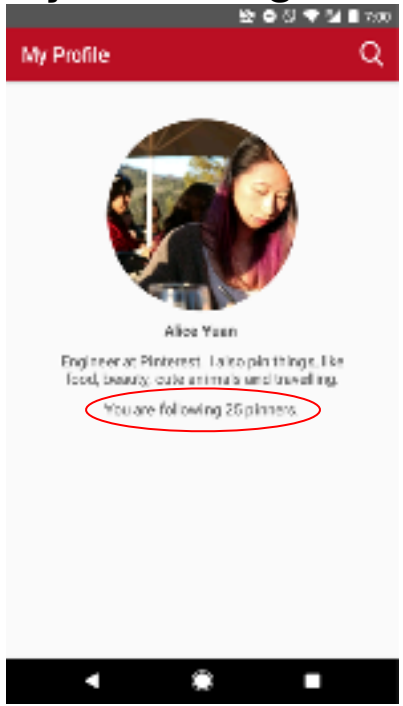
Problem #1

Simple UI, but a pain to add new features

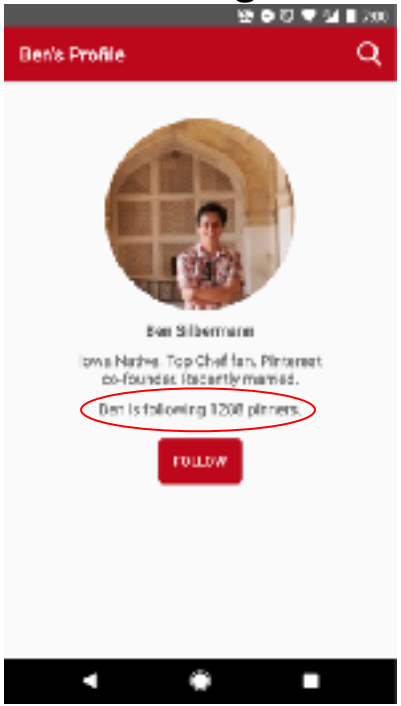
previous design



MyProfileFragment



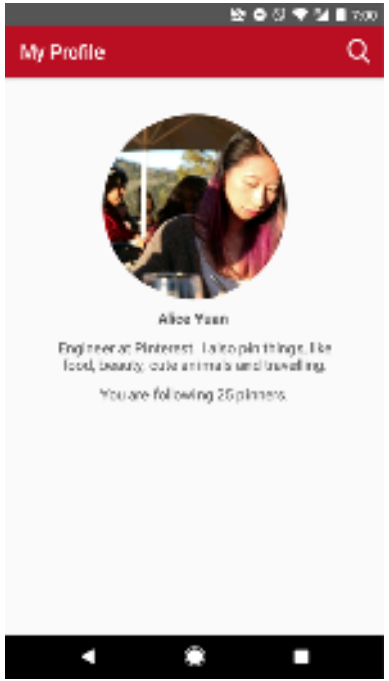
PinnerFragment



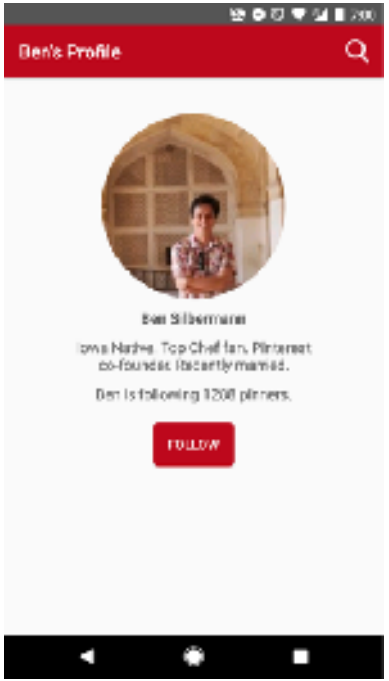
BaseProfileFragment - abstract class

with shared logic

MyProfileFragment



PinnerFragment



```
abstract class BaseProfileFragment extends Fragment {
    @Nullable protected RoundedImageView _profileIv;
    @Nullable protected TextView _nameTv;
    @Nullable protected TextView _bioTv;
    @Nullable protected TextView _followersTv;
    @Nullable protected LinearLayout _layout;

    protected void updateView(@NonNull final PDKUser user) {
        _nameTv.setText(user.getFirstName() + " " + user.getLastName());
        _bioTv.setText(user.getBio());
        _followersTv.setText(getResources().getString(R.string.following,
                                                      user.getFollowingCount());
```

```
abstract class BaseProfileFragment extends Fragment {
  @Nullable protected RoundedImageView _profileIv;
  @Nullable protected TextView _nameTv;
  @Nullable protected TextView _bioTv;
  @Nullable protected TextView _followersTv;
  @Nullable protected LinearLayout _layout;

  protected void updateView(@NonNull final PDKUser user) {
    _nameTv.setText(user.getFirstName() + " " + user.getLastName());
    _bioTv.setText(user.getBio());
    _followersTv.setText(getResources().getString(R.string.following,
                                 user.getFollowingCount());
```

Code Smell:
Protected member variables

```
abstract class BaseProfileFragment extends Fragment {
  @Nullable protected RoundedImageView _profileIv;
  @Nullable protected TextView _nameTv;
  @Nullable protected TextView _bioTv;
  @Nullable protected TextView _followersTv;
  @Nullable protected LinearLayout _layout;

  protected void updateView(@NonNull final PDKUser user) {
      _nameTv.setText(user.getFirstName() + " " + user.getLastName());
      _bioTv.setText(user.getBio());
      _followersTv.setText(getResources().getString(R.string.following,
                                  user.getFollowingCount());
```

Code Smell:
Logic in base class can change

```
public class MyProfileFragment extends
BaseProfileFragment {

    @Override
    protected void updateView(
                        @NonNull PDKUser user) {
        // update View and Avatar
```

```
public class PinnerFragment extends
BaseProfileFragment {

@Override
protected void updateView(
                    @NonNull PDKUser user) {
    // update View and Avatar
```

```
public class MyProfileFragment extends
BaseProfileFragment {

    @Override
    protected void updateView(
                    @NonNull PDKUser user) {
        // update View and Avatar
```

```
public class PinnerFragment extends
BaseProfileFragment {

@Override
protected void updateView(
                @NonNull PDKUser user) {
    // update View and Avatar
```

Code Smell:
Overriding and reimplementing logic

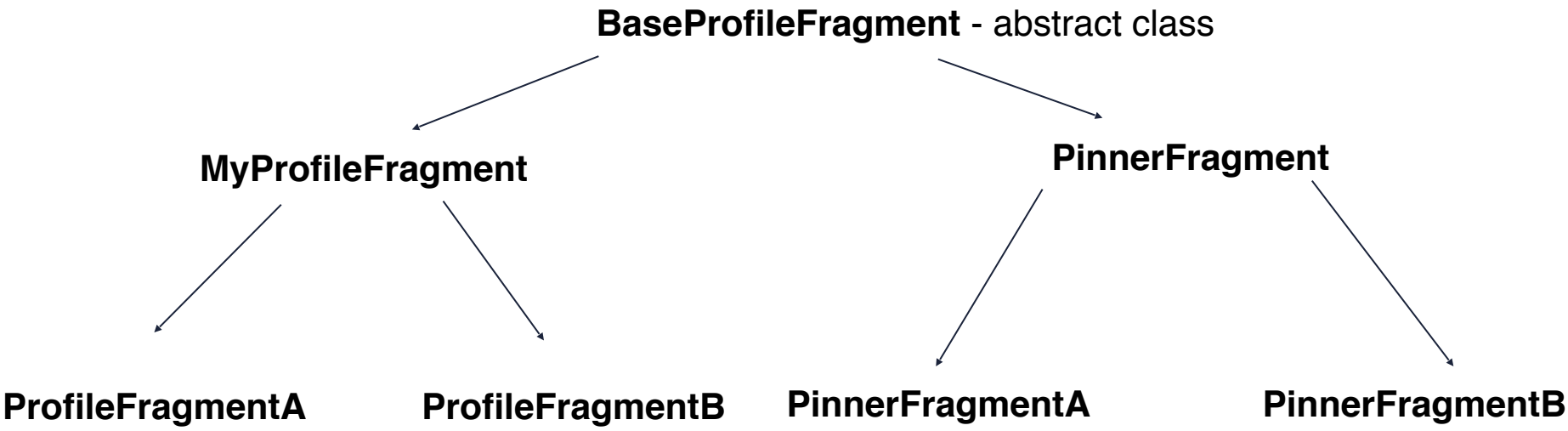
What if we multiplied the fragments?

BaseProfileFragment - abstract class

MyProfileFragment

PinnerFragment

What if we multiplied the fragments?



What if we multiplied the fragments?

BaseProfileFragment - abstract class

MyProfileFragment

PinnerFragment

ProfileFragmentA

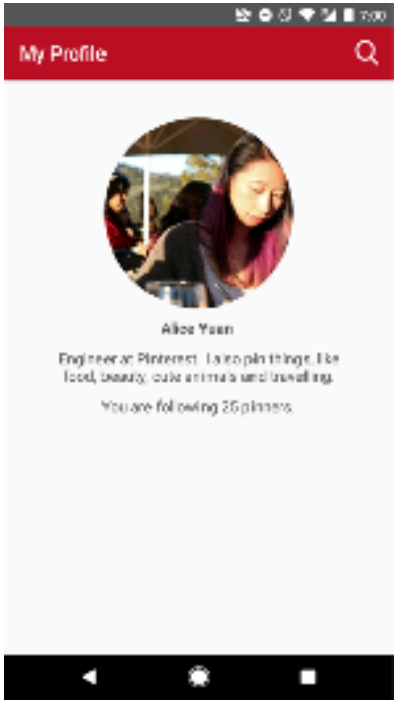
ProfileFragmentB

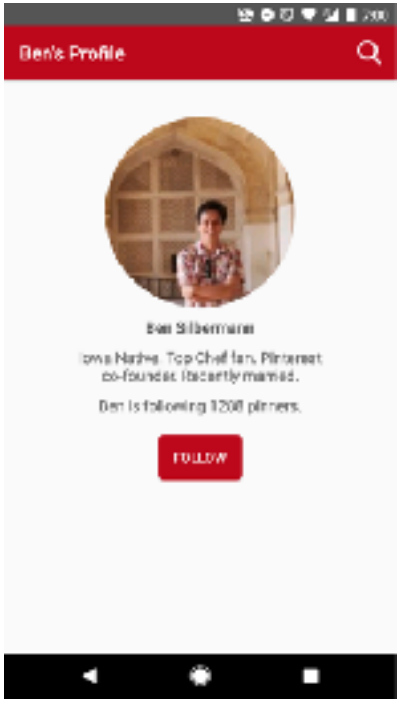
PinnerFragmentA

PinnerFragmentB

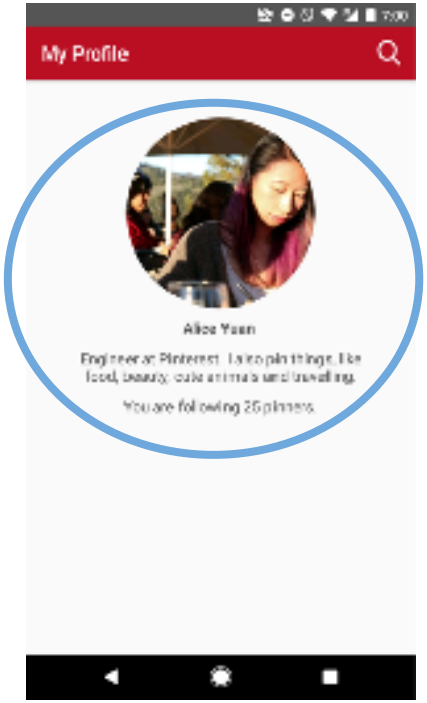
Inheritance Hell

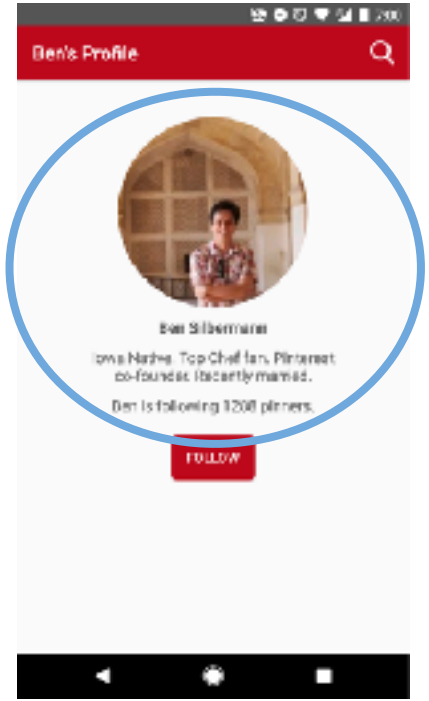
Is a? vs. Has a?
Inheritance vs Composition



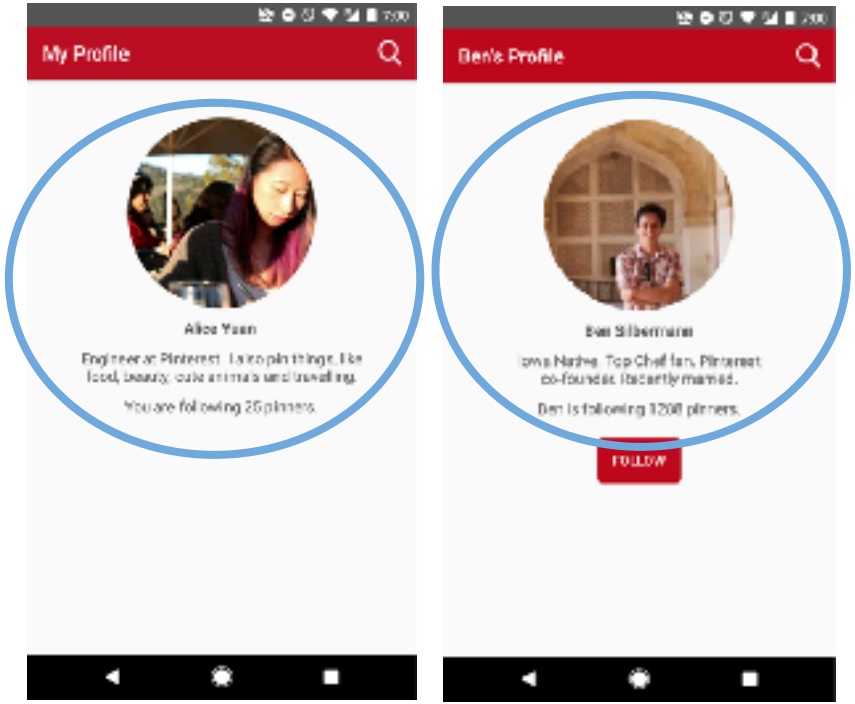


Is a? vs. Has a?
Inheritance vs Composition





Is a? vs. Has a?
Inheritance vs Composition



What is the relationship of the common UI?

MyProfileFragment **is** an avatar view
PinnerFragment **is** an avatar view

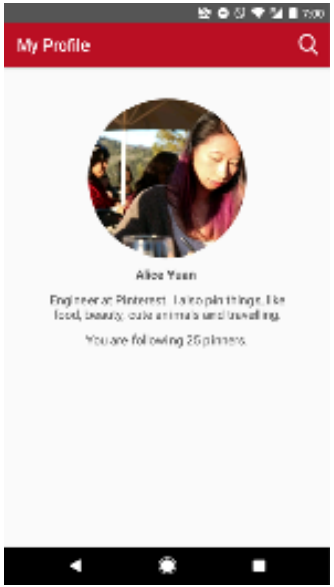
OR

MyProfileFragment **has** an avatar view
ProfileFragment **has** an avatar view

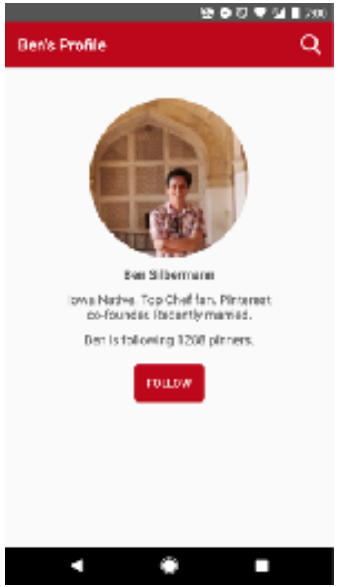
Previous architecture

BaseProfileFragment - abstract class

MyProfileFragment

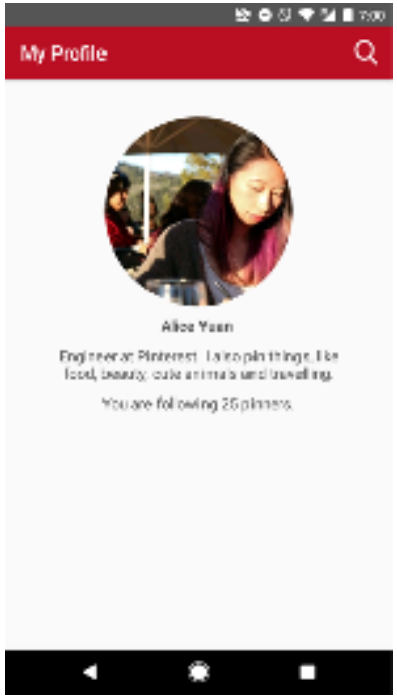


PinnerFragment



New architecture

MyProfileFragment

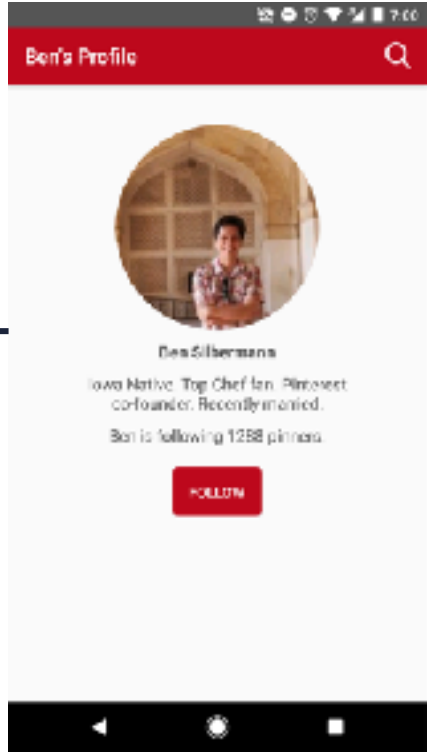


AvatarView

updateView(...)

New architecture

PinnerFragment



AvatarView

updateView(...)

```
class AvatarView extends LinearLayout {
    @Nullable private RoundedImageView _profileIv;
    @Nullable private TextView _nameTv;
    @Nullable private TextView _bioTv;
    @Nullable private TextView _followersTv;

    public void updateView(String name, String imageUrl, String bioText) {
        _nameTv.setText(name);
        _bioTv.setText(bioText);
        Glide.with(getContext())
            .load(imageUrl)
            .into(_profileIv);
    }
}
```

```
class AvatarView extends LinearLayout {
    @Nullable private RoundedImageView _profileIv;
    @Nullable private TextView _nameTv;
    @Nullable private TextView _bioTv;
    @Nullable private TextView _followersTv;

    public void updateView(String name, String imageUrl, String bioText) {
        _nameTv.setText(name);
        _bioTv.setText(bioText);
        Glide.with(getContext())
            .load(imageUrl)
            .into(_profileIv);
    }
}
```

Private member variables

```
class AvatarView extends LinearLayout {
  @Nullable private RoundedImageView _profileIv;
  @Nullable private TextView _nameTv;
  @Nullable private TextView _bioTv;
  @Nullable private TextView _followersTv;

  public void updateView(String name, String imageUrl, String bioText) {
    _nameTv.setText(name);
    _bioTv.setText(bioText);
    Glide.with(getContext())
        .load(imageUrl)
        .into(_profileIv);
  }
}
```

Pass through custom attributes

Key Takeaway: Be deliberate with inheritance - think composition first

Key Takeaway: Be deliberate with inheritance - think composition first

Inheritance is intentional - Declare your class as **final** initially

```
public final class MyProfileFragment
```

Key Takeaway: Be deliberate with inheritance - think composition first

Inheritance is intentional - Declare your class as **final** initially

```
public final class MyProfileFragment
```

Use inheritance when the **is** relationship makes sense

Example: a vehicle **has** tires, a truck **is** a type of vehicle

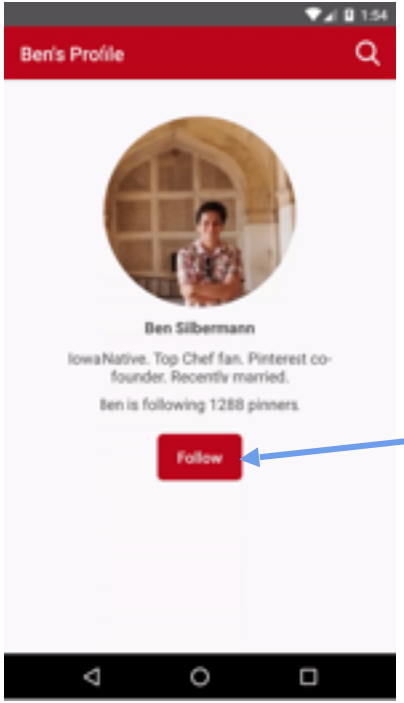
Example: the android Fragment: when we create a custom Fragment, the custom Fragment **is** an android fragment

Problem #2

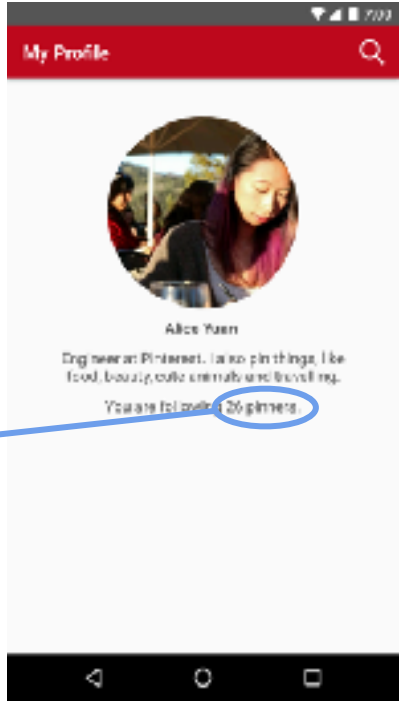
So many bugs related to follow button

So many bugs related to follow button

PinnerFragment

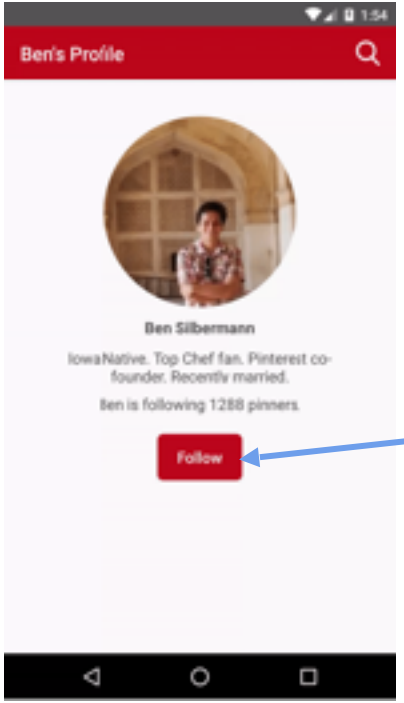


MyProfileFragment

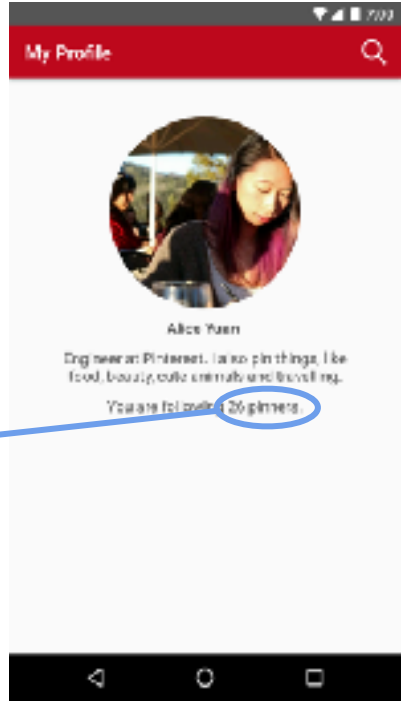


So many bugs related to follow button

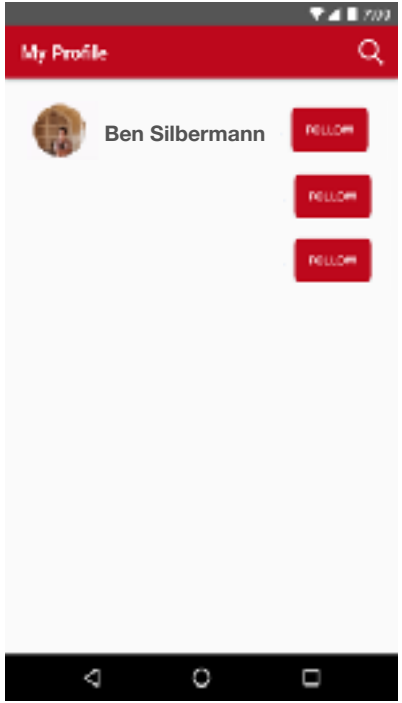
PinnerFragment



MyProfileFragment

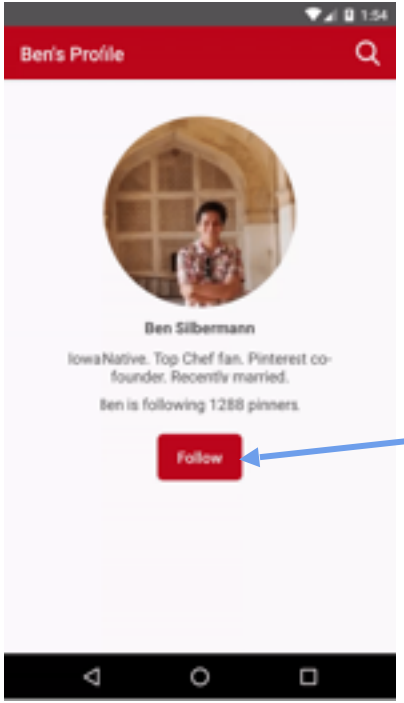


Fragment2

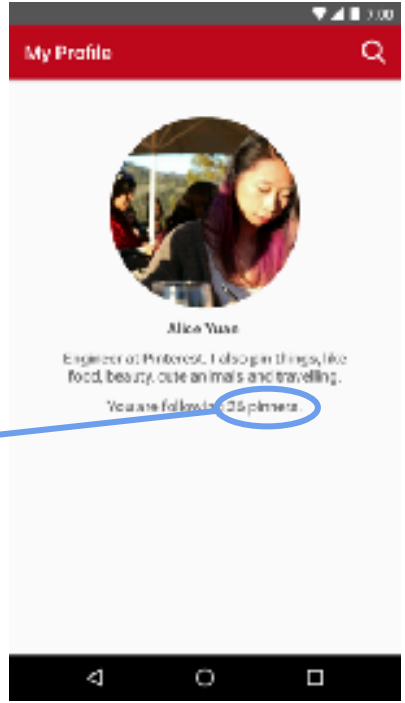


So many bugs related to follow button

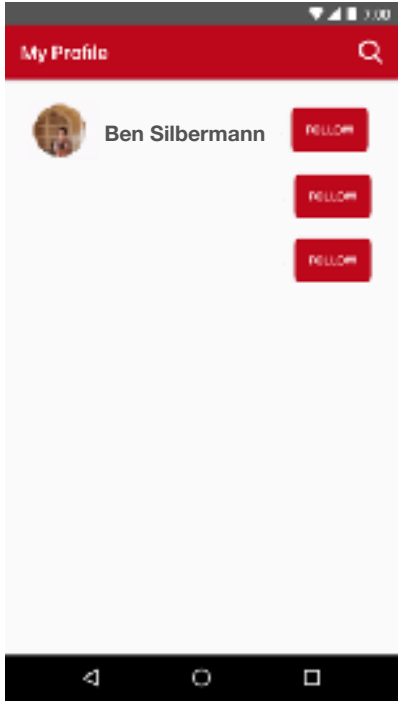
PinnerFragment



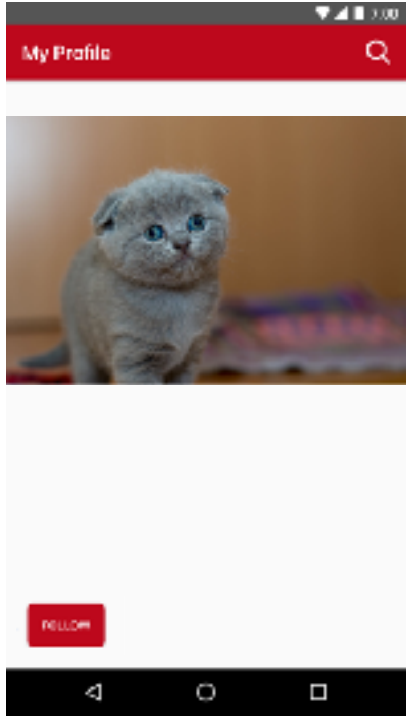
MyProfileFragment



Fragment2

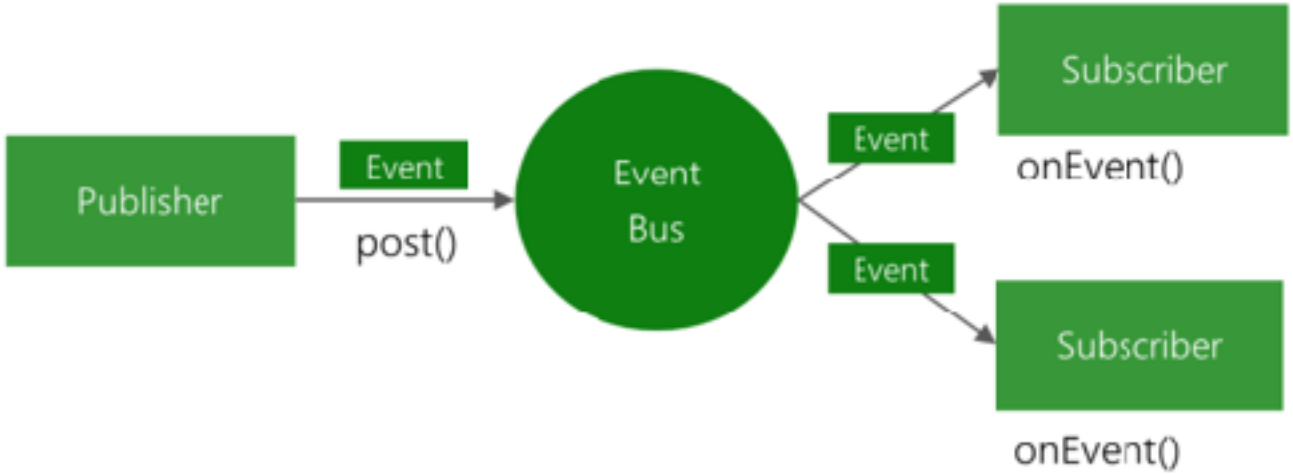


Fragment3



EventBus Libraries such as Eventbus, Otto, Tinybus

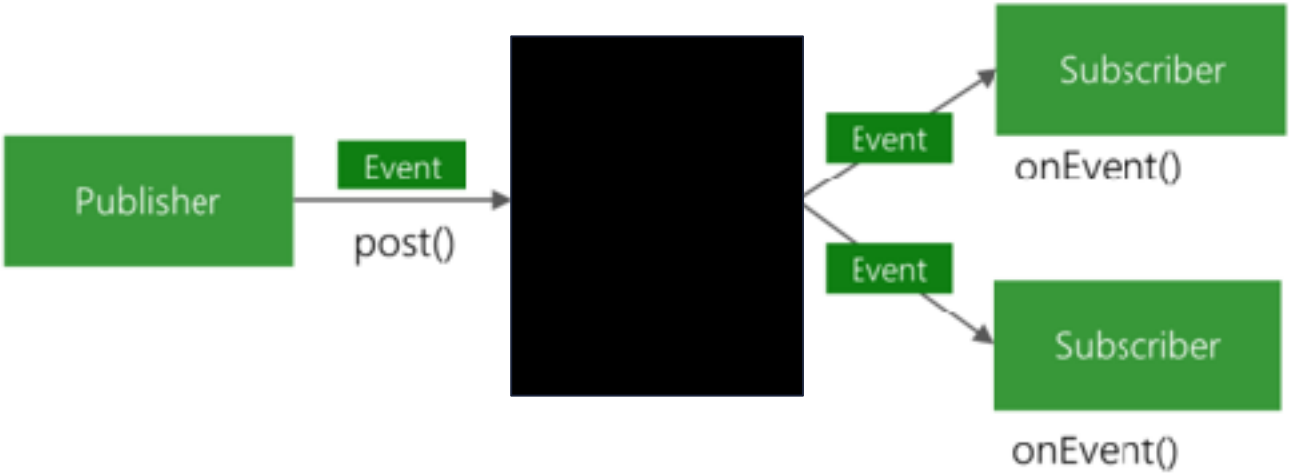
EventBus is a publish/subscribe event bus optimized for Android.



At the implementation level, it is a global event queue.

EventBus Libraries such as Eventbus, Otto, Tinybus

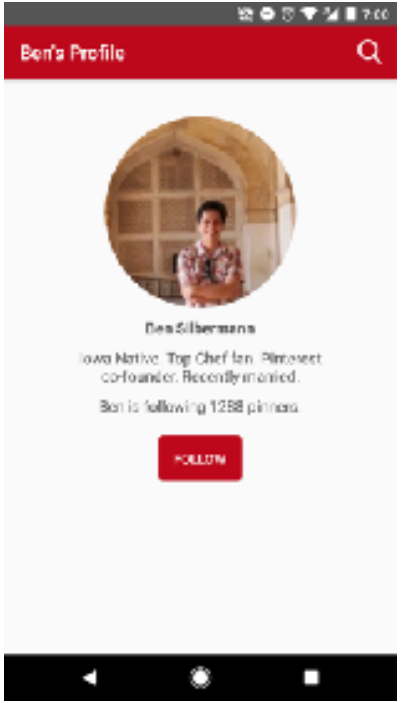
EventBus is a publish/subscribe event bus optimized for Android.



At the implementation level, it is a global event queue.

Notifying updates - why is it breaking?

PinnerFragment

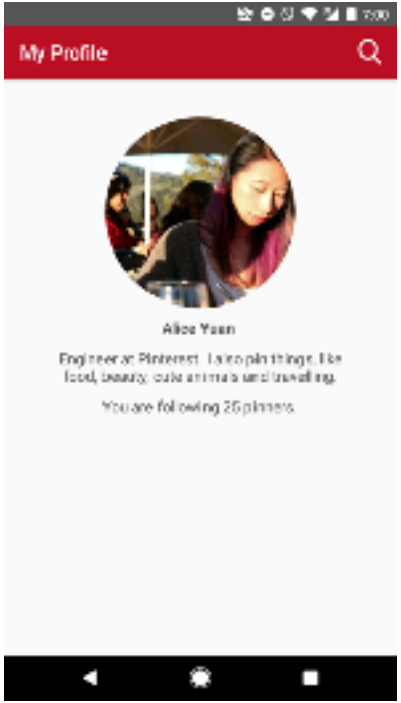


Publisher

```
void onFollowButtonClicked() {
    EventBus.getDefault()
    .post(new FollowEvent(newFollowingCount));
}
```

Notifying updates - why is it breaking?

MyProfileFragment

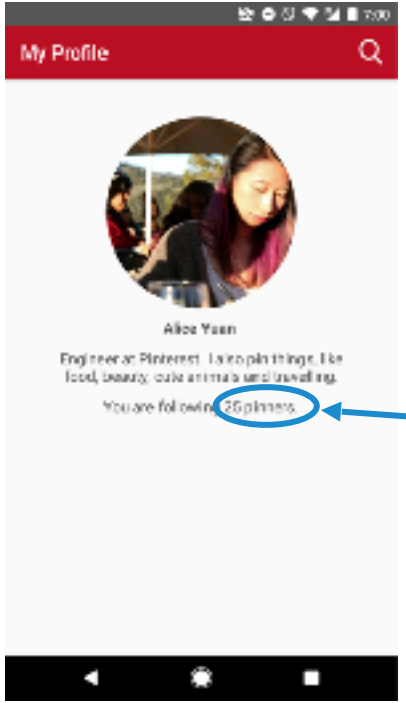


Subscriber

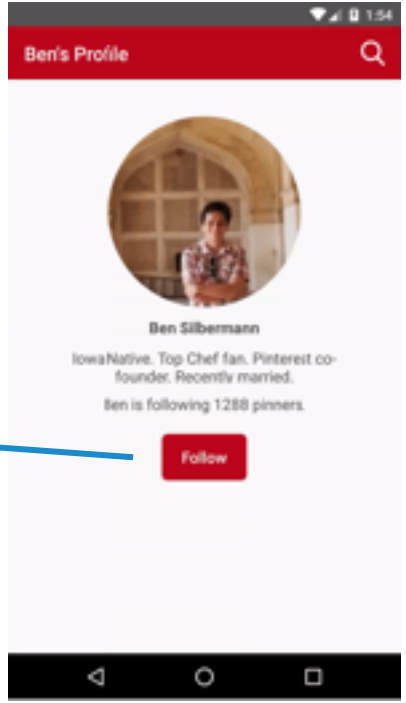
```
onMessageEvent(FollowEvent event) {
    setFollowingCount(event.getNumFollowing();
}
```

Notifying updates - why is it breaking?

Subscriber: MyProfileFragment

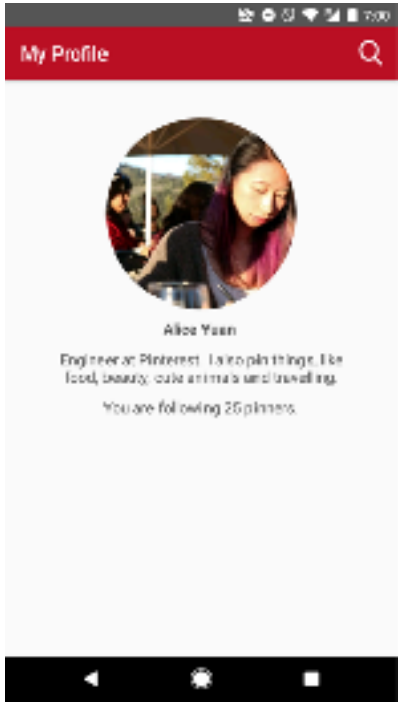


Publisher: PinnerFragment

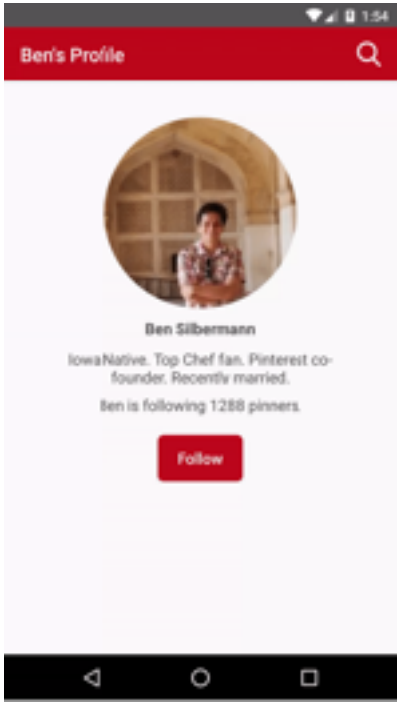


Notifying updates - why is it breaking?

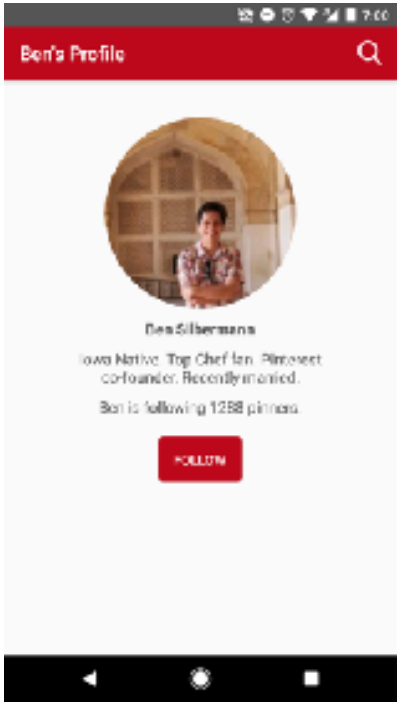
Subscriber 1



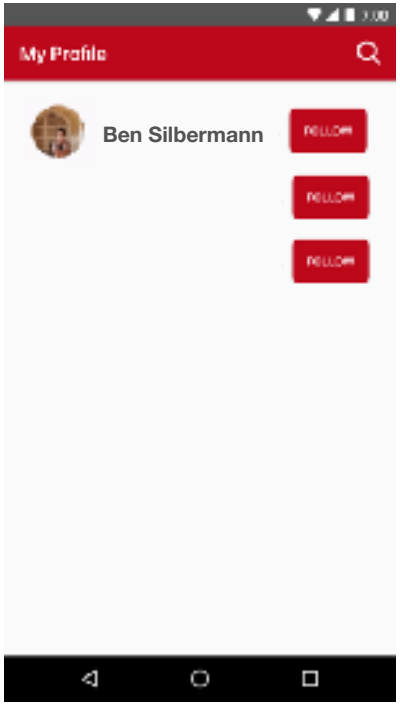
Subscriber 2



Publisher 1



Publisher 2



Because it’s decoupled, Eventbus libraries have many pitfalls

- There’s **no enforced responsibility** of ensuring something is listening
- As we add more events it **decreases reliability and maintainability** of the code
- A pain to write **tests** for
- Thus, only use eventbus when the client **does not care** if the event is **consumed** or not eg. Logging events are consumed by the server

Solution: Observer/ Listener Pattern

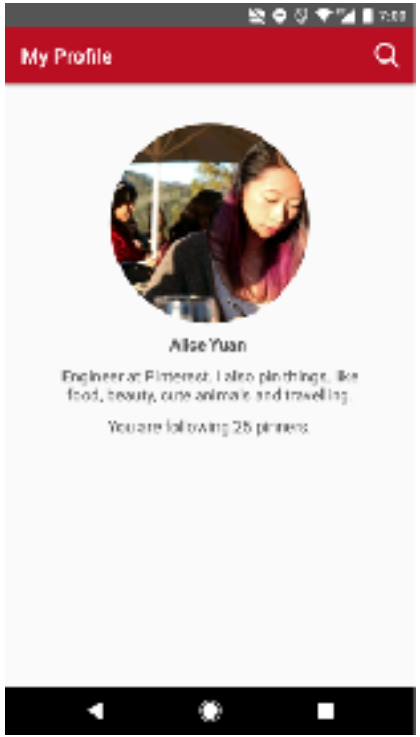
Simple interface to enforce tight coupling with an observer and subscriber pattern

FollowListener.java

```
public interface FollowListener {

   void onFollowCountChanged(int count);

}
```

Solution: Observer/ Listener Pattern

MyProfileFragment

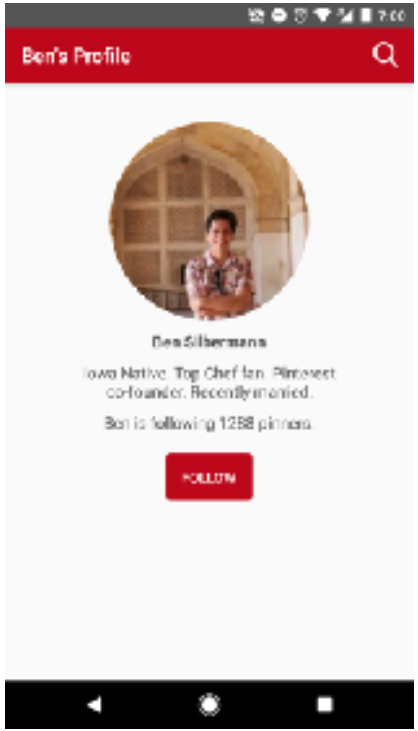


Subscriber

```
public class MyProfileFragment implements FollowListener {
    //… registerListener in navigation

    @Override
    public void onFollowCountChanged(int count) {
        setFollowingCount(count);
    }
}
```

Solution: Observer/ Listener Pattern

PinnerFragment

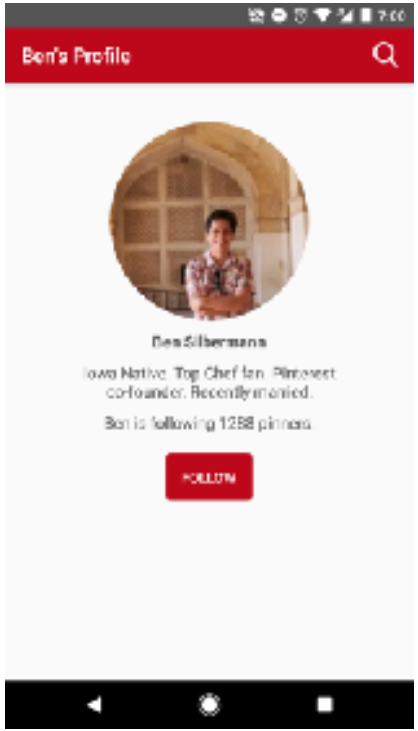


Publisher

```
public class PinnerFragment {
    private FollowListener _followListener;

    public void registerListener(FollowListener followListener) {
        _followListener = followListener;
    }
    //...
```

Solution: Observer/ Listener Pattern

PinnerFragment



Publisher

```
// network request to get following count…
new UserCountApiCallback() {
    @Override
    public void onSuccess(int count) {
        _following = newFollowing;
        if (_followListener != null) {
            _followListener.onFollowCountChanged(count);
        }
    }
}
```

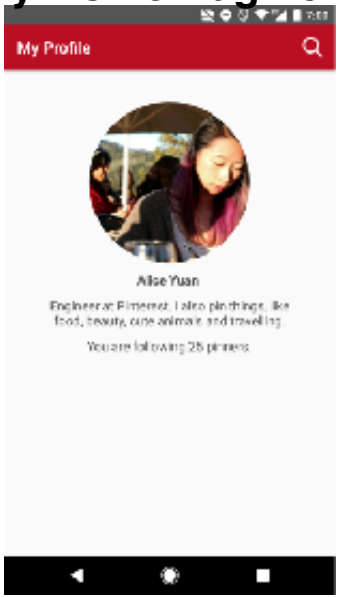
Key Takeaway: EventBus Libraries are often abused due to its simplicity

- UI updates is not a use case that benefits from loose coupling
- Use event bus for places where loose coupling makes sense
- Use an Observable/ Listener pattern otherwise

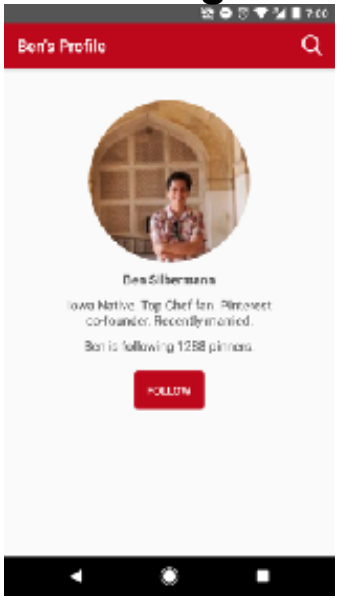
Problem #3

Do we need to send events to maintain data consistency?

MyProfileFragment

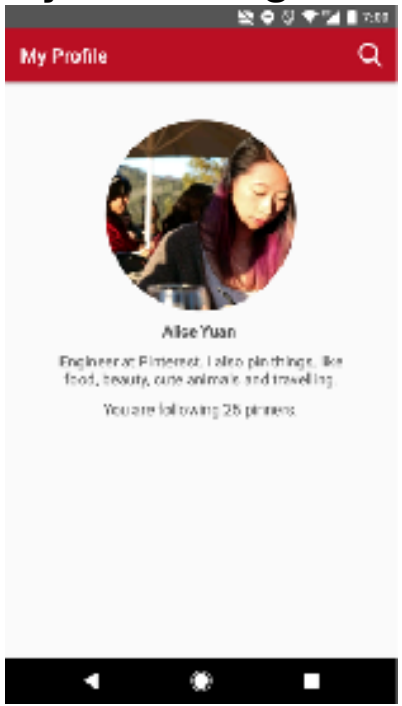


PinnerFragment



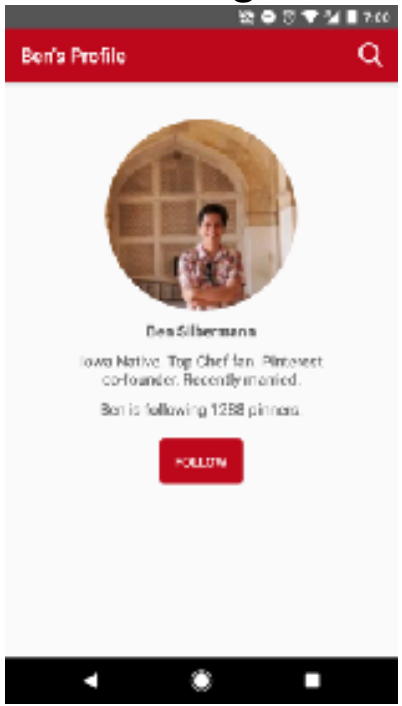
Why do I even need to send events?

MyProfileFragment

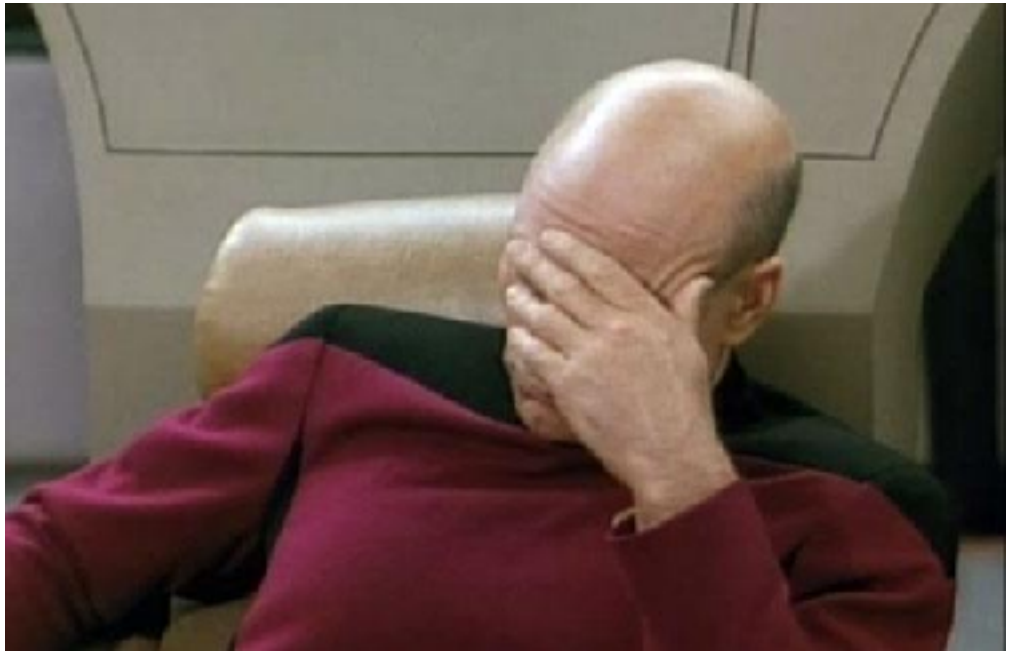



```
PDKUser _myUser;

int _followingCount
```

PinnerFragment




```
PDKUser _myUser;

PDKUser _curUser;
```



```
public class MyProfileFragment extends MVPFragment implements MyProfileView {

    //cache values to avoid having to make network calls in the future
    private PDKUser _myUser;
    private int _followingCount;

    private void loadUser() {
        if (_myUser == null) {
            loadMyUserAPI();
        } else {
            _avatarView.updateView(_myUser.getFirstName() + " " + _myUser.getLastName(),
                    MyUserUtils.get().getLargeImageUrl(_myUser), _myUser.getBio());
            updateFollowingCount(_followingCount);
    }
}
```

```
public class MyProfileFragment extends MVPFragment implements MyProfileView {

  //cache values to avoid having to make network calls in the future
  private PDKUser _myUser;
  private int _followingCount;

  private void loadUser() {
      if (_myUser == null) {
          loadMyUserAPI();
      } else {
          _avatarView.updateView(_myUser.getFirstName() + " " + _myUser.getLastName(),
                  MyUserUtils.get().getLargeImageUrl(_myUser), _myUser.getBio());
          updateFollowingCount(_followingCount);
  }
}
```

Code Smell:
Caching models on fragment basis

```
public class MyProfileFragment extends MVPFragment implements MyProfileView {

  //cache values to avoid having to make network calls in the future
  private PDKUser _myUser;
  private int _followingCount;

  private void loadUser() {
      if (_myUser == null) {
          loadMyUserAPI();
      } else {
          _avatarView.updateView(_myUser.getFirstName() + " " + _myUser.getLastName(),
                  MyUserUtils.get().getLargeImageUrl(_myUser), _myUser.getBio());
          updateFollowingCount(_followingCount);
  }
}
```

Code Smell:
Model dependent logic on the view layer

Other code smells which indicate poor data consistency

- Our example: UI instances tracking model state
- More examples:

Other code smells which indicate poor data consistency

- Our example: UI instances tracking model state
- More examples:
  - Global static variables

```
public static PDKUser myUser = null;
```

Other code smells which indicate poor data consistency

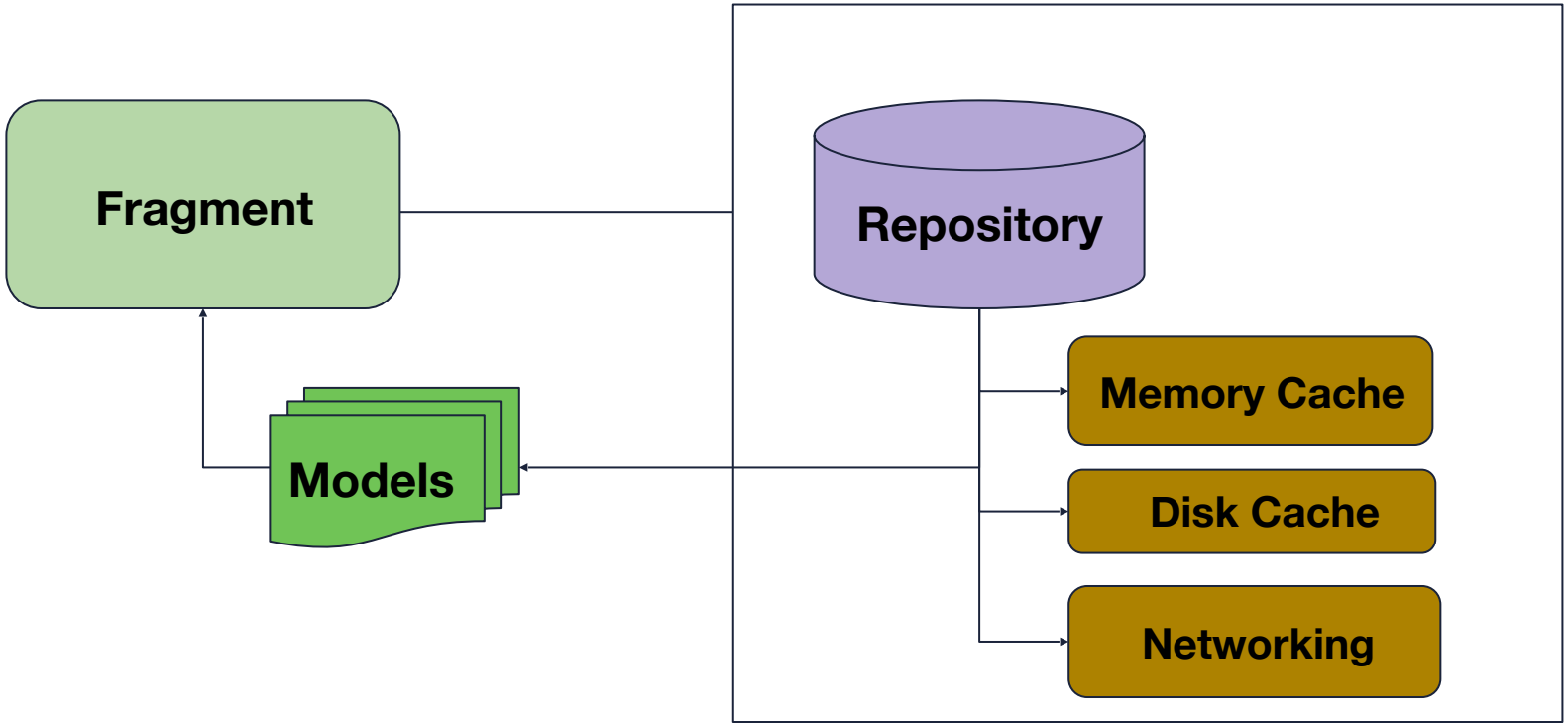
- Our example: UI instances tracking model state
- More examples:
  - Global static variables

```
public static PDKUser myUser = null;
```

  - Variables hidden through a singletons. Often a utils class pattern

```
class MyUserHelper {
    public static String updateUserNameAndStuff(String userName) {
        myUser.setUserName(userName);
        // ... stuff happens
    }
```

Solution: have a central area to handle all storing and retrieval of models



Solution Code:
Repository of User models

```
interface RepositoryListener<M extends Object> {

    void onSuccess(M model);

    void onError(Exception e);

}
```

```
public class UserRepository {
  private PDKUser _myUser;
  //...
  public void loadMyUser(@NonNull final RepositoryListener<PDKUser> listener) {
    if (_myUser != null) {
        listener.onSuccess(_myUser);
        return;
    }

    PDKClient.getInstance().getMe(USER_FIELDS, new PDKCallback() {
        @Override
        public void onSuccess(PDKResponse response) {
            _myUser = response.getUser();
            listener.onSuccess(_myUser);
        }
```

```
public class UserRepository {
  private PDKUser _myUser;
  //...
  public void loadMyUser(@NonNull final RepositoryListener<PDKUser> listener) {
    if (_myUser != null) {
        listener.onSuccess(_myUser);
        return;
    }

    PDKClient.getInstance().getMe(USER_FIELDS, new PDKCallback() {
        @Override
        public void onSuccess(PDKResponse response) {
            _myUser = response.getUser();
            listener.onSuccess(_myUser);
        }
```

Central cache check

```
public class UserRepository {
  private PDKUser _myUser;
  //...
  public void loadMyUser(@NonNull final RepositoryListener<PDKUser> listener) {
    if (_myUser != null) {
      listener.onSuccess(_myUser);
      return;
    }

    PDKClient.getInstance().getMe(USER_FIELDS, new PDKCallback() {
      @Override
      public void onSuccess(PDKResponse response) {
        _myUser = response.getUser();
        listener.onSuccess(_myUser);
      }
```

Central network call & update model

```
public class MyProfileFragment extends Fragment {

    private AvatarView _avatarView;
    //...

    private void loadMyUser() {
        UserRepository.get().loadMyUser(new RepositoryListener<PDKUser>() {
            @Override
            public void onSuccess(PDKUser user) {
                _avatarView.updateView(user.getFirstName() + " " + user.getLastName(),
                        user.getImageUrl(), user.getBio());
            }
        });
}
```

```
public class MyProfileFragment extends Fragment {

  private AvatarView _avatarView;
  //...

  private void loadMyUser() {
    UserRepository.get().loadMyUser(new RepositoryListener<PDKUser>() {
      @Override
      public void onSuccess(PDKUser user) {
        _avatarView.updateView(user.getFirstName() + " " + user.getLastName(),
            user.getImageUrl(), user.getBio());
      }
    });
}
```

Simple call to update view

Repository with RxJava

- Lots of asynchronous communication problems cannot be easily solved with a listener pattern
- More complex example: chaining data calls, returning more than one type data response
- rxJava can solve this through Observable stream
- There exist libraries that adapt network callbacks into rxJava Observables for you

Key Takeaway: Build a central way to fetch and retrieve models

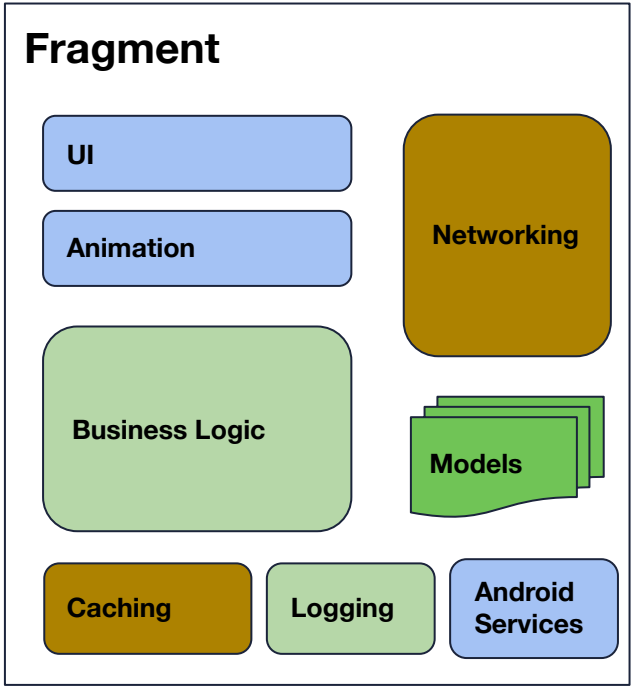
- **Stop** storing instances of models in your fragments!
- Ensure **data consistency** regardless of where we're retrieving or storing our models
- Store and fetch models in a central area

Final issue, Problem #4

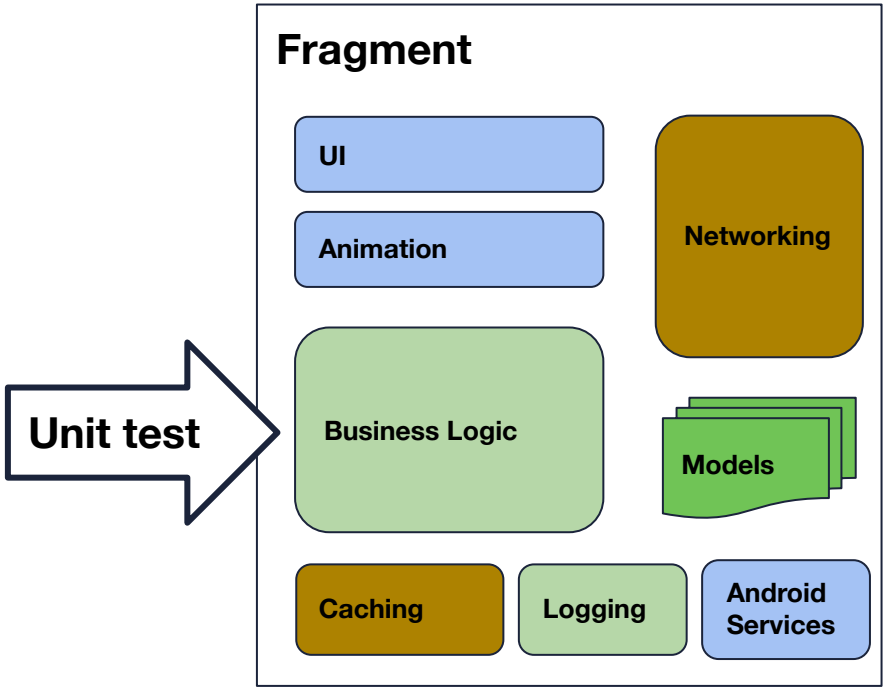
No unit tests :(
Why is writing unit tests so difficult?

- We want to ensure that we are correctly setting the user profile display data
- What makes writing this unit test so complex?

What a typical fragment looks like



What a typical fragment looks like



class MyProfileFragment

```
private void loadMyUser() {
  PDKClient.getInstance().getMe(USER_FIELDS, new PDKCallback() {
    @Override
    public void onSuccess(PDKResponse response) {
      ...
      PDKUser user = response.getUser();
      _myAvatarView.setUser(user);
    }
```

class MyProfileFragment

Mock network callback

```
private void loadMyUser() {
  PDKClient.getInstance().getMe(USER_FIELDS, new PDKCallback() {
    @Override
    public void onSuccess(PDKResponse response) {
      ...
      PDKUser user = response.getUser();
      _myAvatarView.setUser(user);
    }
```

class MyProfileFragment

```
private void loadMyUser() {
  PDKClient.getInstance().getMe(USER_FIELDS, new PDKCallback() {
    @Override
    public void onSuccess(PDKResponse response) {
      ...
      PDKUser user = response.getUser();
      _myAvatarView.setUser(user);
    }
```

Mock Translation of Response to Model

class MyProfileFragment

```
private void loadMyUser() {
  PDKClient.getInstance().getMe(USER_FIELDS, new PDKCallback() {
    @Override
    public void onSuccess(PDKResponse response) {
      ...
      PDKUser user = response.getUser();
      _myAvatarView.setUser(user);
    }
```

Mock Android Framework UI using Roboelectric

Let’s make this simpler,
How should a unit test look like?

```
@Test
public void testLoadMyUserSuccess() throws Exception {
   verify(_myProfileView).updateAvatarView(FIRST_NAME + " " + LAST_NAME,
          IMAGE_URL, BIO);
  verify(_myProfileView).updateFollowingText();
}
```

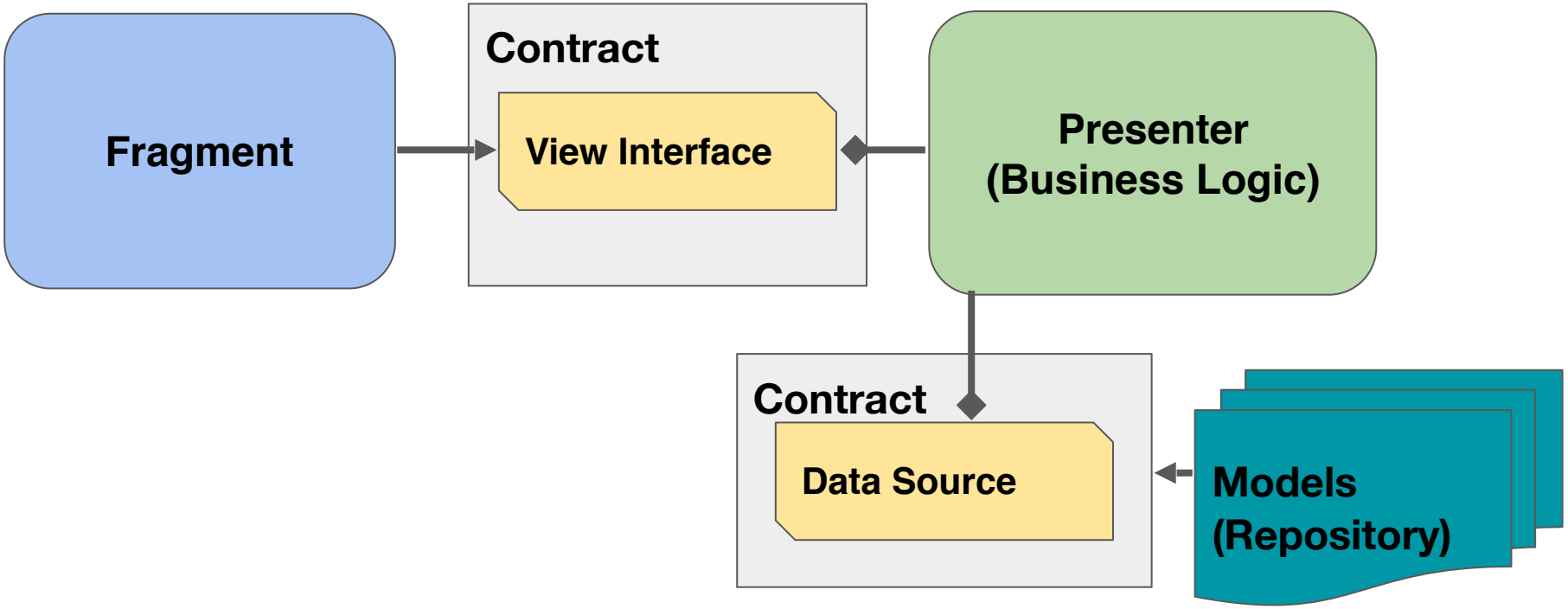
Solution: Separate concerns through an interface

You've likely heard of the paradigms MVVM and MVP (Model-View-ViewModel, Model-View-Presenter)

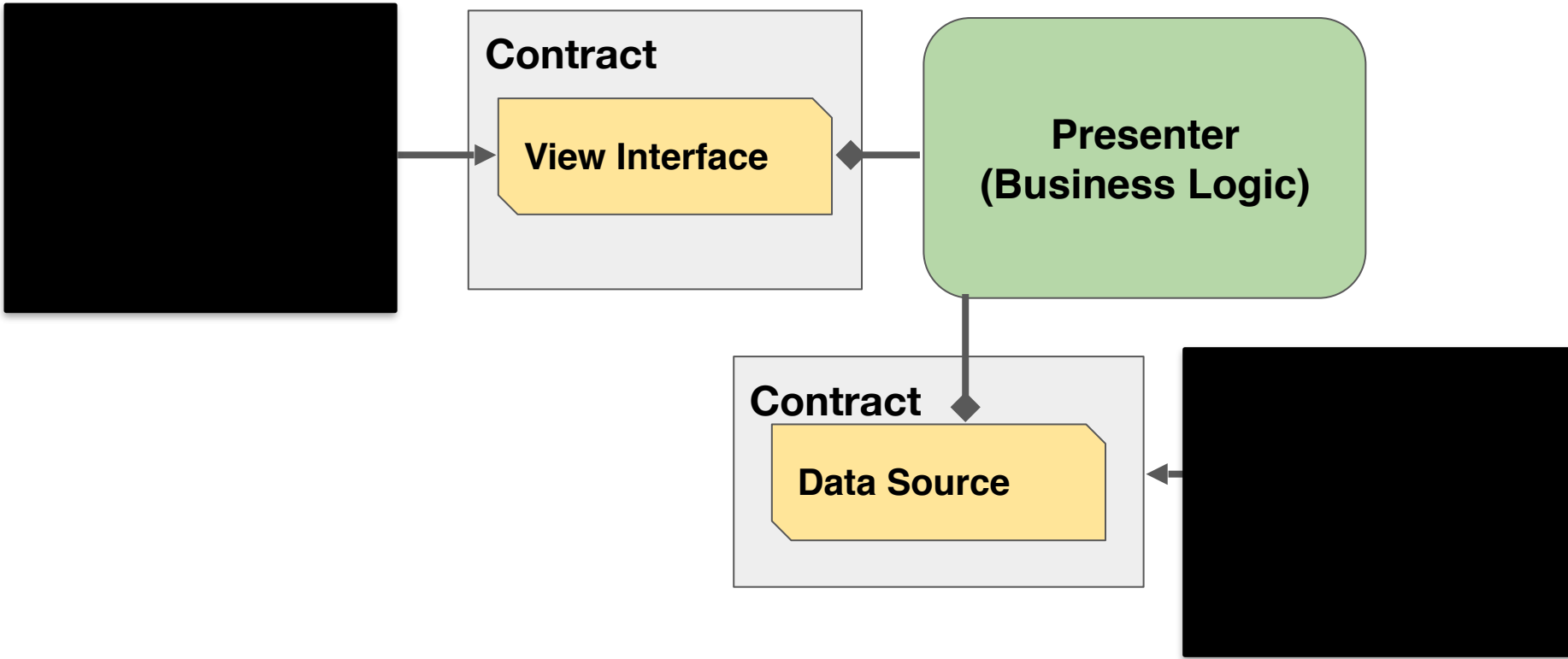
Key value: they separate concerns between areas that do not need to know about each other.

You can now communicate between classes without knowing the internals

Separate concerns - MVP example



Separate concerns - MVP example



Define a Contract for the View

```
interface MyProfileView extends MVPView {

    void updateAvatarView(String name, String imageUrl, String bioText);

    void updateFollowingCount(int count);

}
```

```
public class MyProfileFragment extends MVPFragment implements MyProfileView {

    @Override
    public void updateAvatarView(String name, String imageUrl, String bioText) {
        _avatarView.updateView(name, imageUrl, bioText);
    }

    @Override
    public void updateFollowingCount(int count) {
        _avatarView.updateFollowingText(getResources().getString(
            R.string.my_user_following, count));
    }
}
```

Define a Contract for the Repository

```
public interface UserDataSource {

  void loadMyUser(@NonNull final RepositoryListener<PDKUser> listener);

  void loadMyUserNumFollowing(@NonNull final RepositoryListener<Integer> listener);

}
```

```
public interface UserDataSource {

  void loadMyUser(@NonNull final RepositoryListener<PDKUser> listener);

  void loadMyUserNumFollowing(@NonNull final RepositoryListener<Integer> listener);

}


public class UserRepository implements UserDataSource {
```

```
public class MyProfilePresenter implements Presenter<MyProfileView> {
    public MyProfilePresenter(@NonNull UserDataSource dataSource) {
      _dataSource = dataSource;
    }

    @Override
    public void attachView(@NonNull final MyProfileView view) {
        _view = view;
        loadUser(view);
   }

   void loadUser(final MyProfileView view) {
      _dataSource.loadMyUser(new RepositoryListener<PDKUser>() {
       @Override
       public void onSuccess(PDKUser user) {
```

```
public class MyProfilePresenter implements Presenter<MyProfileView> {
    public MyProfilePresenter(@NonNull UserDataSource dataSource) {
      _dataSource = dataSource;
    }

    @Override
    public void attachView(@NonNull final MyProfileView view) {
        _view = view;
        loadUser(view);
    }

    void loadUser(final MyProfileView view) {
      _dataSource.loadMyUser(new RepositoryListener<PDKUser>() {
        @Override
        public void onSuccess(PDKUser user) {
```

No longer referencing repository

```
public class MyProfilePresenter implements Presenter<MyProfileView> {
    public MyProfilePresenter(@NonNull UserDataSource dataSource) {
      _dataSource = dataSource;
    }

    @Override
    public void attachView(@NonNull final MyProfileView view) {
        _view = view;
        loadUser(view);
    }

    void loadUser(final MyProfileView view) {
      _dataSource.loadMyUser(new RepositoryListener<PDKUser>() {
        @Override
        public void onSuccess(PDKUser user) {
```

No longer need to mock Android Framework view

```
    @Override
    public void attachView(@NonNull final MyProfileView view) {
        _view = view;
        loadUser(view);
    }

    void loadUser(final MyProfileView view) {
        _dataSource.loadMyUser(new RepositoryListener<PDKUser>() {
            @Override
            public void onSuccess(PDKUser user) {
                view.updateAvatarView(user.getFirstName() + " " + user.getLastName(),
                        user.getImageUrl(), user.getBio());
            }
        }
    });
```

```
@Override
public void attachView(@NonNull final MyProfileView view) {
    _view = view;
    loadUser(view);
}

void loadUser(final MyProfileView view) {
    _dataSource.loadMyUser(new RepositoryListener<PDKUser>() {
      @Override
      public void onSuccess(PDKUser user) {
        view.updateAvatarView(user.getFirstName() + " " + user.getLastName(),
              user.getImageUrl(), user.getBio());
      }
    }
});
```

No longer mocking network callback

*Requires a MVP framework to function

```
interface Presenter<V extends MVPView> {

  void attachView(@NonNull final V view);
  void detachView();

}

interface MVPView {
  Presenter createPresenter();
  Presenter getPresenter();
}
```

What does writing unit tests look like now?

```
@Test
public void testLoadMyUserSuccess() throws Exception {
  verify(_myProfileView).updateAvatarView(FIRST_NAME + " " + LAST_NAME,
        IMAGE_URL, BIO);
  verify(_viewResources).getString(anyInt(), eq(_followingUsersList.size()));
  verify(_myProfileView).updateFollowingText();
}
```

```
public static abstract class BaseMyProfileTest {
        @Mock MyProfileView _myProfileView;
        UserDataSource _mockDataSource;
        @Mock ViewResources _viewResources;

        @Before
        public void setUp() throws Exception {
            _mockDataSource = getUserDataSource();
            MyProfilePresenter myProfilePresenter = new MyProfilePresenter(_viewResources,
_mockDataSource);
            myProfilePresenter.attachView(_myProfileView);
        }

       abstract UserDataSource getUserDataSource();
    }
```

*MVP Testing Infra not shown

```
public static abstract class BaseMyProfileTest {
        @Mock MyProfileView _myProfileView;
        UserDataSource _mockDataSource;
        @Mock ViewResources _viewResources;

        @Before
        public void setUp() throws Exception {
            _mockDataSource = getUserDataSource();
            MyProfilePresenter myProfilePresenter = new MyProfilePresenter(_viewResources,
_mockDataSource);
            myProfilePresenter.attachView(_myProfileView);
        }

       abstract UserDataSource getUserDataSource();
    }
```

Mock interfaces using Mockito

*MVP Testing Infra not shown

Separation of Concerns makes the code cleaner!

- Improves understandability of codebase
  - view updates can be quite long and that detracts from understanding logic of the codebase
- Increases reusability of the codebase, views can be reused
- Can also be used in building libraries and modularizing the codebase

Key takeaway: Unit tests are easy to write when you separate the business logic

- Choose a paradigm (MVP, MVVM) to follow which separates concerns
- Use interfaces to abstract internals away and use a mocking library eg. Mockito to mock functionality
- Improves testability and also understandability of the code

We made it, that’s all!

Recap

1. **Be deliberate** about inheritance, think about **composition**
2. Eventbus is a loosely coupled library - **use tight coupling** patterns such as RxJava or Observable callbacks instead
3. Create a **central** location to store and retrieve models to ensure **data consistency**
4. **Separate the areas of concern** to increase testability and maintenance

The solution to avoiding poor patterns is awareness

Thanks!

Find my slides at:
https://bit.ly/2pJrHQF

You ask me questions at:
me@aliceyuan.ca
@Names_Alice