ANDROID DESIGN PATTERNS AND MERGING CODEBASES (ANDROID LEARNING, BTPOC, AND OLD ZEN)

INTRODUCTION

I've been building Android apps for some time now and I've gained a lot of insights and have developed and refined some really cool patterns that help simplify Android development in a big way. These patterns simply re-use existing platform capabilities, but reduce the friction in the developer experience (DevXP) and it reduces the "cost" of leveraging certain platform capabilities in Android.

FRAMEWORK PACKAGE ORGANIZATION

I don't like heavy frameworks which require me to make drastic changes to my project structure and even the way I think about my apps. I like frameworks that can be used partially, since their constituent elements deliver value; they need to be strongly related, but loosely coupled.

I've come up with a simple package structure for the sample app and framework that's shipped with this open source project.

- app.* - this package is where the app goes (your code)
- integration.* - this package is where binding between your code and the framework happens (since the framework is largely declarative)
    - Look at LocalEvents, ObservableProperty, Notifications, and DB_kvp_enum, and DB_blob_enum to declare the resources your app will need (to the framework)
- zen.* - this package actually contains the framework code
    - .base.* - this contains all the base classes that might use in your apps instead of Android platform equivalents (eg: SimpleFragmentActivity instead of FragmentActivity)
    - .core.* - this is where most of the framework classes exist. The most important ones are: LifecycleHelper, LocalEventsManager, DBManager, and NotificationsManager.

This structure keeps the framework code in code in the zen packages, while exposing only the declarative stuff in the integration package. And your code goes in the app package, or wherever you want.

It is difficult switching perspectives from a framework builder to an app builder (which is a framework consumer) and back and forth; and I think this is a good enough balance. It maintains a conceptual separation, but it's quite tightly connected in the codebase right now (it would be even

better if the declarative stuff in integration package could be generated using a domain specific language, or expressed in JSON or XML).

I've also come up with a simple way to save preferences. SavedDataKeys is an app specific (not framework) enum that makes it super easy to read/write strings into the Android SharedPreferences.

PASSING OBJECTS AROUND & COMPLEX LIFECYCLE STATES

One of the biggest problem with the Android DevXP is that it's loosely coupled, and it's difficult to pass object references around (which is also a strength of the platform and architecture). Even though an activity and service might be in the same VM/process, it's difficult to pass object references between them (activity to activity, or activity to service, or service to activity), due to the firewall-y nature of the OS and API. You can workaround this by making things static, but that introduces other issues (since processes can be retained between subsequent executions, that can introduce old state, and also leak stuff).

The basic problem is the Intent passing mechanism – it doesn't allow Java objects to be passed around. You have to turn everything into Parcelable in order to move it, either ways, you're not able to pass object references around, only serialized object values. So this poses a problem when shared state information must be passed around, and it's not possible to declare these variables, or pass them around due to the fact that the codebases aren't connected by scope. This happens all the time. An object does something, and a service or activity must know. This is incredibly difficult to do in Android, since object references can't be passed around via Intents! The are various solutions to this, most of which involve creating a static list of listeners for a bunch of events, and then managing these in the app itself. The only problem with static in Android is that static lives for the life of the process, which spans multiple lifetimes of service and activity! So you might have ghost objects in your static map or list, that were around from a previous run of the service or activity and aren't meant to be there. Now, there are ways around this – you can use ObjC like reference counting, and ensure that you dispose of all these listeners (that were allocated in onCreate), in onDestroy – and this works, but this adds tremendous friction to the DevXP and discourages developers from even going there.

I use SoftHashMap in the framework to be able to enable object passing via events; SoftHashMap cleans itself over time basically (non deterministically), and will discard any references to objects that have been passed some time ago. And internally in the framework I use reference counting type measures to allocate/deallocate correctly (so developers don't have to). So with rare exceptions (LocalEventsManager being one of them) the use of static has been avoided in the framework. Also, by using Application subclass it's possible to get around using static, and allocate everything in onCreate() that's going to be global, and then pass references to this around by making some base classes that subclass Activity, Service, Fragment, and their Map variants. All of this is in the zen.framework.base.* package. You just have to update your app's AndroidManifest.xml to use this AppData class as its Application.

HOW TO USE THE FRAMEWORK

Here are the solutions to pass objects around within the same application / process:

ObservableProperty

An observable property (OP) is globally accessible (but NOT static). An OP is a wrapper around an Object. The ObservablePropertyManager is created in the Application subclass (AppData) and is available to all the framework classes, and your classes if you use AppData. ObservablePropertyListeners (OP listeners) can be attached to any OP via LifecycleHelper (which is baked into the zen.base.* classes, or you can use it yourself). This provides an elegant solution of EASILY attaching an OP listener and not having to worry about how to wire it up exactly, or cleaning it up! LifecycleHelper does both the bind and release, and so this is just a wonderful pattern that I've created, which makes it easy, primarily, to act as a binder between the data model and the UI. The biggest uses I've made of this, so far, has been to get some meaningful state out of the underlying data model that directly impacts the UI, and then push these state changes directly into the UI. Eg: CurrentLocation (which is somehow generated by various sensors that I don't want to think about). Eg: Service can be stopped now, and service can be started or stopped in situations where there are complex state transitions that have to occur to be able to start/stop.

Source code

1. Declaring an OP:

```
public enum ObservableProperty implements ConstantsIF {
  ServiceIsStarted,
}//end enum ObservableProperty
```

2. Setting a value on the OP (in a subclass of zen.base.* superclasses):

```
getAppData().observablePropertyManager
  .setValue(ObservableProperty.ServiceIsStarted,
            Boolean.TRUE);
```

3. Responding to value changes in OP (OP listener):

```
getAppData().observablePropertyManager.addPropertyChangeObserver(
    new ObservablePropertyListener() {
      public ObservableProperty getProperty() {
        return ObservableProperty.ServiceIsStarted;
      }

      public String getName() {return "MyActivity service start/stop buttons";}

      public void onChange(String propertyName, Object value) {
        if (value == Boolean.TRUE) {
          myViews.btn_startservice.setEnabled(false);
          myViews.btn_stopservice.setEnabled(true);
        }
        else {
          myViews.btn_startservice.setEnabled(true);
          myViews.btn_stopservice.setEnabled(false);
        }
      }
    },
    true
);
```

LocalEvents

The idea here is that any object anywhere in the app's process might want to send a message, to any other object, and pass data (Java objects) as parameters. And this is the problem that's solved by LocalEventsManager, by using a very clever way of passing objects around (in IntentHelper using UUIDs and SoftHashMaps) it makes this easy to do in Android.

In order to implement LocalEvents, I actually use the Android 4 compatibility library's LocalBroadcastManager (which is a system service). There's no state information that's stored in the LocalEventsManager (with the exception of localHistory, which is currently experimental), and thusly it's static.

Without thinking you can use LifecycleHelper and add a LocalEventsListener to it (as a resource), which will respond to a local event firing, and do something. You declare all your LocalEvents in the enumeration (declaratively) and this class is in the integration.* package.

LifecycleHelper does the bind/release for you, and you don't think about it. Also, you can fire LocalEvents from anywhere, and just pass it 1 string payload, and 1 object parameter. These 2 params are kept simple by design (there's actually nothing in the underlying implementation to prevent it from passing any number of params using varargs; I've just restricted this behavior in the interface that's exposed).

The key use cases for this are notifying disparate parts of the app (eg: activity to service, or service to activity interactions) that were just very difficult to do before (without putting in a lot of effort). Eg: it's not possible for a service to directly communicate with an Activity. The Activity has to use a ServiceConnection to get to the service, and all kinds of complicated stuff. Instead, you can use LocalEvents. Just register listeners on the service and activity, and either can fire the LocalEvents of interest and pass objects back and forth; no need to think about it at all! Without having a simple,

stateless, static, accessible from anywhere mechanism like LocalEvent, this integration would be a very difficult and cycle consuming thing to do. Instead, you have a simple, cheap, and frictionless way to pass objects around that are all interested in a specific LocalEvents enumeration, and it works even for complex scenarios.

Source code

1. Declaring a LocalEvents enum:

```
public enum LocalEvents {
  ShutdownMyService,
}//end enum LocalEvents
```

2. Firing an event:

```
LocalEventsManager.fireEvent(getActivity(),
                             LocalEvents.ShutdownMyService,
                             “string”, new Object());
```

3. Responding to an event (in a subclass of any zen.base.* superclass):

```
getLifecycleHelper().addResource(new LocalEventsListener() {
  public LocalEvents getLocalEvent() {
    return LocalEvents.ShutdownMyService;
  }

  public String getName() {
    return "responds to shutdown service event";
  }

  public void onReceive(String stringPayload, Object objectPayload, Bundle extras) {
    // stringPayload is “string” that was fired
    // objectPayload is new Object(); that was fired
    stopServiceNow();
  }
});
```

Persistence

The framework also has support for easy database management. It defines to types of collections – a key value pair store, and a blob store. You can declaratively tell Zen how many of these to create and manage for you. You can do this by changing the following enums in the integration.* package:

1. DB_kvp_enum – can store text key and value pairs
2. DB_blob_enum – can store text “blobs”; so it’s really “text” and not “blob”

Declare however many of these collections you want the framework create for you at startup (when the Application, AppData, is created). You will have to use the Application class, and the zen.base.* classes for this. You can easily access data in these databases by using getAppData().dbManager from any subclass of the base-classes in zen.base.* package.

Source code

1. Declaring a “blob” data store (not a KVP or key-value-pair store):

```
public enum DB_blob_enum {
  Test_BLOB_DB,
}
```

2. Saving to persistence:

```
getAppData().dbManager.getDB(DB_blob_enum.Test_BLOB_DB).add(“some string”);
```

3. Responding to persistence write (using LocalEvents that are fired by DBManager):

```
getLifecycleHelper().addResource(new LocalEventsListener() {
  public LocalEvents getLocalEvent() {
    return LocalEvents.DB_blob_Change;
  }

  public String getName() {
    return "respond to changes in db, and update list";
  }

  public void onReceive(String stringPayload, Object objectPayload, Bundle extras) {
    DB_blob db = getAppData().dbManager.getDB(DB_blob_enum.Test_BLOB_DB);
    String dbName = db.getDbName());
    if (stringPayload.equals(dbName) {
      // do something with the cursor from the db ...
      listAdapter.changeCursor(db.getAllCursor());
    }
  }
});
```

Automatic binding and releasing

That’s another thing that might not be apparent with this architecture – and that is, the ease with which things can be bound/released. You don’t have to worry about specific lifecycle events to hook into, or complex interactions at different lifecycle events (eg: the Android ones for Activity, Fragment, and Service). You simply attach OP listeners, or LocalEvents listeners, and you’re good to go. If you’re a zen.base.* subclass, then you can use OP manager to fire OP changes. And anyone can use LocalEventsManager to fire LocalEvents and pass objects around (since LocalEventsManager is totally static & stateless)!

So as you can see the OP stuff is more tied to Android platform specific stuff (eg: Activity, Fragment, Service) and the LocalEvents stuff is literally across the entire app, any part of it! In summary:

1. OP – stateful, tied to zen.base.* classes.
2. LocalEvents – stateless, and can be called from anywhere to pass any object.

BEST PRACTICES

I've used the Zen framework to build many apps, and I've come up with some best practices for when to use different framework capabilities. What are the appropriate uses of these paradigms, and what use cases neatly map to one or the other? The basic mechanisms for sharing data are:

1. OP – this actually keeps a copy of the property in memory at all times (tied to AppData/Application life). And it also "pokes" interested parties (notifies OP listeners) only when the value of the property changes (you can set the same value repeatedly on the OP, and the OP listener will only be notified the first time the value changed).
2. LocalEvents – these are stateless; and can be used to poke listeners when things change, making interested parties aware of them. You can always search for usages of a specific LocalEvents enum, and it will instantly tell 2 types of users of this event, which is incredibly useful for understanding what is going on with local events:
    a. Things that fire events
    b. Things that respond to events fired.
3. DBManager – these persist key-value pairs or blobs in the DB, and they do fire events when the DB changes using LocalEvents (.DB_blob_Change and .DB_kvp_Change).

There are many different uses for these 3 facilities. In fact, some use cases require them to be used together. Here are some use cases that outline the use of some of all of these.

- UI control enable/disable – observable property is great for this. There are many situations where you want to enable/disable a button based on the state of something, and this maps perfectly to the OP pattern.
- Wiring between components – local event is great for this. The wiring does not have to have any previous state information that it has to maintain making the LocalEvents pattern perfect for only transmitting changes, as they occur to interested parties (when they are interested).

Those were the clean/simple use cases. Here are some more complex ones:

- Large data storage and change notification – a combination of OP and DB is good for this. Let's say a weather forecast report is downloaded by a recurring service (based on a user's location), and stored using DBManager. However, the UI needs this when it's inflated and needs to bind to the weather data, but this object is so heavy that it would take substantial effort just to get it out of the DB and into memory so that the UI can display it. There are different scenarios where a DB Cursor would be the right approach, but not in this case, since there's just 1 large object that needs to be managed. So the correct pattern is to NOT use LE, but to use OP & DB. There needs to be an intent service that runs when AppData launches that loads all this heavy stuff into an OP (at startup) from the DB. Subsequently when a UI needs this data, it's just in OP. Also if the data wasn't even in the DB when the this intent service ran, it's ok, since the UI will just understand there's no data to display. And when the data does come in (from a different recurring service) it will be pushed into the UI via the OP anyway.

- Non persistent data storage and notification – a combination of OP and static classes are good for this. For eg, let's say we built an object (location-store) that stores the current location (that can be retrieved using a variety of different methods, by using internal sensors or even external location sensors via BT). So the location-store object manages the current location. Let's say there are 2 sources and sinks for location – internal location sensor (phone GPS) and external location sensor (BT GPS device). So when any services require the location, where do they go? They can just go to the static class and get the current location from there; but what if no location has been set yet (by any source)? Well, then the location will be null. Again, in this case, it would be best for a 1 time intent service to run and pre-populate the OP for location with some value. And when the UI runs, interested parties can register to get updates when this OP changes. No need for LocalEvents in this situation – since state information is required (even though this state is not stored in a DB).