Beyond satisfying your clients and your employer, there's one more important individual to keep happy in your career as a developer: Future You! (The artist's conception of Future You to the right implies no guarantee of personal jetpack availabilty for developers in the near future.) :]

Future You will inherit the code you write at some point down the road, and will likely have a lot of questions about how and why you coded things the way you did. But instead of leaving tons of confusing comments in your code, a much better approach is to adopt common **design patterns**.

This article will introduce a few common design patterns for Android that you can use while developing apps. Design patterns are reusable solutions to common software problems. The design patterns covered here aren't an exhaustive list, nor an academically-citable paper. Rather, they serve as a workable references and starting points for further investigation.

Getting Started

"Is there anywhere in this project where I'll have to change the same thing in multiple places?" – Future You

Future You should minimize time spent doing "detective work" looking for intricate project dependencies, so they would prefer a project that's as reusable, readable, and recognizable as possible. These goals span a single object all the way up to the entire project and lead to patterns that fall into the following categories:

- **Creational patterns:** how you *create* objects.
- **Structural patterns:** how you *compose* objects.
- **Behavioral patterns:** how you *coordinate* object interactions.

You may already be using one or several of these patterns already without having A Capitalized Fancy Name for it, but Future You will appreciate you not leaving design decisions up to intuition alone.

In the sections that follow, you'll cover the following patterns from each category and see how they apply to Android:

Creational

- Builder
- Dependency Injection
- Singleton

Structural

- Adapter
- Facade

Behavioral

- Command
- Observer
- Model View Controller
- Model View ViewModel

Note: This article isn't like a traditional raywenderlich.com tutorial in that it doesn't have an accompanying sample project that you can follow along with. Treat it instead like an article to get you up to speed to the different patterns you'll see used in our other Android tutorials, and to discover ways to improve your own code.

Creational Patterns

"When I need a particular complex object, how do I get one?" – Future You

Future You hopes the answer is not "copy and paste the same code every time you need an instance of this object." Instead, creational patterns make object creation simple and easily repeatable.

Here are several examples:

Builder

At a sandwich spot down my block, I use a small pencil to check off the bread, ingredients, and condiments I'd like on my sandwich from a checklist on a slip of paper. Even though the checklist's title instructs me to "build my own" sandwich, I really only fill out the form and hand it over the counter. I'm not actually doing the sandwich-building, just the customizing…and the consuming. :]

Similarly, the Builder pattern separates the construction of a complex object (the slicing of bread, the stacking of pickles) from its representation (a yummy sandwich); in this way, the same construction process can create different representations.

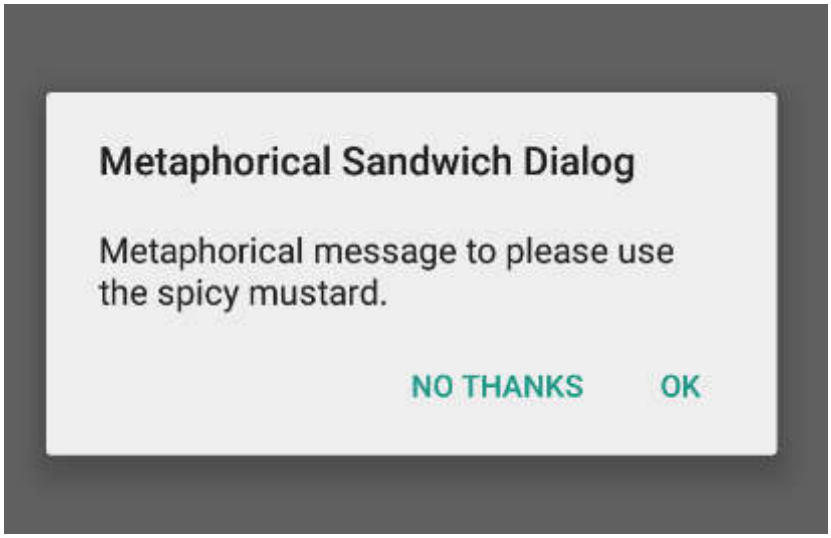
In Android, the Builder pattern appears when using objects like `AlertDialog.Builder`:

```
new AlertDialog.Builder(this)
    .setTitle("Metaphorical Sandwich Dialog")
    .setMessage("Metaphorical message to please use the spicy
mustard.")
    .setNegativeButton("No thanks", new
DialogInterface.OnClickListener() {
      @Override public void onClick(DialogInterface
dialogInterface, int i) {
        // "No thanks" button was clicked
      }
    })
    .setPositiveButton("OK", new
```

```
DialogInterface.OnClickListener() {
      @Override public void onClick(DialogInterface
dialogInterface, int i) {
        // "OK" button was clicked
      }
    })
    .show();
```

This builder proceeds step-by-step and lets you specify only the parts of your `AlertDialog` that matter to you. Take a look at the AlertDialog.Builder documentation; you'll see there are quite a few commands to choose from when building your alert.

The code block above produces the following alert:



A different set of choices would result in a completely different sandwich – er, alert. :]

Dependency Injection

Dependency injection is like moving into a furnished apartment. Everything you need is already there; you don't have to wait for furniture delivery or follow pages of IKEA instructions to put together a Borgsjö bookshelf.

In strictly software terms, dependency injection has you provide any objects required when you instantiate a new object; the new object doesn't need to construct or customize the objects itself.

In Android, you might find you need to access the same complex objects from various points in your app, such as a network client, an image loader, or `SharedPreferences` for local storage. You can **inject** these objects into your activities and fragments and access them right away.

Dagger 2 is the most popular open-source dependency injection framework for Android and was developed in collaboration between Google and Square. You simply annotate a class with `@Module`, and populate it with `@Provides` methods such as the following:

```
@Module
public class AppModule {
  @Provides SharedPreferences
```

```
provideSharedPreferences(Application app) {
    return app.getSharedPreferences("prefs",
Context.MODE_PRIVATE);
  }
}
```

The module above creates and configures all required objects. As an additional best-practice in larger apps, you could even create multiple modules separated by function.

Then, you make a **Component** interface to list your modules and the classes you'll inject:

```
@Component(modules = AppModule.class)
interface AppComponent {
  ...
}
```

The component ties together where the dependencies are *coming from* (the modules), and where they are *going to* (the injection points).

Finally, you use the `@Inject` annotation to request the dependency wherever you need it:

```
@Inject SharedPreferences sharedPreferences;
```

As an example, you could use this approach in an Activity and then use local storage, *without* any need for the Activity to know how the `SharedPreferences` object came to be.

Admittedly this is a simplified overview, but you can read up on the Dagger documentation for more detailed implementation details. This pattern may seem complicated and "magical" at first, but its use can help simplify your activities and fragments.

Singleton

The Singleton Pattern specifies that only a single instance of a class should exist with a global point of access. This works well when modeling real-world objects only having one instance:

```
public class ExampleSingleton {
  private static ExampleSingleton instance = null;
  private ExampleSingleton() {
    // customize if needed
  }
  public static ExampleSingleton getInstance() {
    if (instance == null) {
      instance = new ExampleSingleton();
    }
    return instance;
  }
}
```

The above class "hides" the creation of the single instance behind the static method `getInstance()` to ensure you only initialize the class once. When you need to access the singleton object, you simply make a call as follows:

```
ExampleSingleton.getInstance();
```

You'll then know you're using the same instance of that class throughout your app.

The Singleton is probably the easiest pattern to initially understand, but can be dangerously easy to overuse – and abuse. Since it's accessible from multiple objects, the singleton can undergo unexpected side effects that are difficult to track down – exactly what Future You doesn't want to deal with. It's important to understand the pattern, but other design patterns may be safer and easier to maintain.

Structural Patterns

"So when I open up this class, how will I remember what's it's doing and how it's put together?" – Future You

Future You will undoubtedly appreciate the Structural Patterns you used to help organize the guts of your classes and objects into familiar arrangements that perform typical tasks. Two commonly-seen patterns in Android are **Adapter** and **Facade**.

Adapter

A famous scene in the movie Apollo 13 features a team of engineers tasked with fitting a square peg into a round hole. This, metaphorically, is the role of an adapter. In software terms, this pattern lets two incompatible classes work together by converting the interface of a class into another interface the client expects.

Consider the business logic of your app; it might be a Product, or a User, or a Tribble. It's the square peg. Meanwhile, a `RecyclerView` is the same basic object across all Android apps. It's the round hole.

In this situation, you can use a subclass of `RecyclerView.Adapter` and implement the required methods to make everything work:

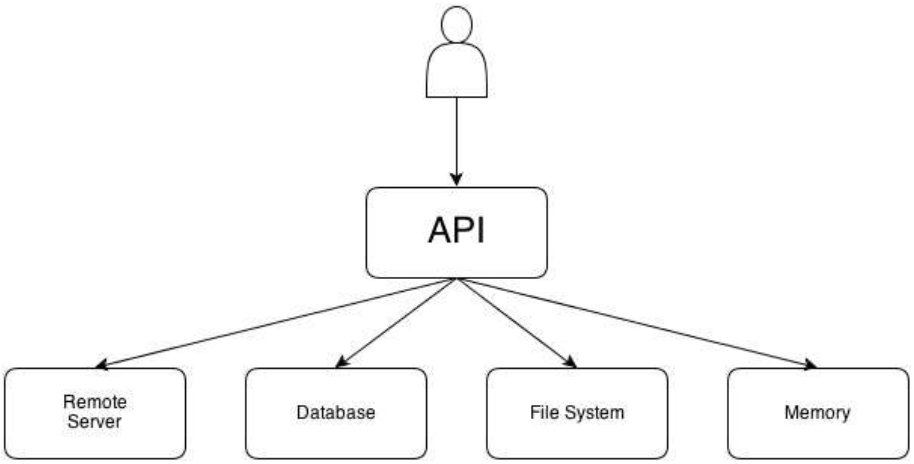
```
public class TribbleAdapter extends RecyclerView.Adapter {

  private List mTribbles;

  public TribbleAdapter(List tribbles) {
    this.mTribbles = tribbles;
  }

  @Override
  public TribbleViewHolder onCreateViewHolder(ViewGroup
viewGroup, int i) {
    LayoutInflater inflater =
LayoutInflater.from(viewGroup.getContext());
    View view = inflater.inflate(R.layout.row_tribble,
viewGroup, false);
    return new TribbleViewHolder(view);
  }

  @Override
  public void onBindViewHolder(TribbleViewHolder viewHolder,
int i) {
    viewHolder.configure(mTribbles.get(i));
  }

  @Override
```

```
  public int getItemCount() {
    return mTribbles.size();
  }
}
```

`RecyclerView` doesn't know what a Tribble is, as it's never seen a single episode of Star Trek – not even the new movies. :] Instead, it's the adapter's job to handle the data and send the configuration command to the correct `ViewHolder`.

Facade

The Facade pattern provides a higher-level interface that makes a set of other interfaces easier to use. The following diagram illustrates this idea in more detail:



If your Activity needs a list of books, it should be able to ask a single object for that list *without* understanding the inner workings of your local storage, cache, and API client. Beyond keeping your Activities and Fragments code clean and concise, this lets Future You make any required changes to the API implementation without any impact on the Activity.

Retrofit from SquareOne is an open-source Android library that helps you implement the Facade pattern; you create an interface to provide API data to client classes like so:

```
public interface BooksApi {
  @GET("/books")
  void listBooks(Callback&lt;List&gt; callback);
}
```

The client simply needs to call `listBooks()` to receive a list of `Book` objects in the callback. It's nice and clean; for all it knows, you could have an army of Tribbles assembling the list and sending it back via transporter beam. :]

This lets you make all types of customizations underneath without affecting the client. For example, you can specify a customized JSON deserializer about which the Activity has no clue:

```
RestAdapter restAdapter =
  new RestAdapter.Builder()
      .setConverter(new MyCustomGsonConverter(new Gson()))
      .setEndpoint("http://www.myexampleurl.com")
```

```
        .build();
return restAdapter.create(BooksApi.class);
```

Notice the use of `MyCustomGsonConverter`, working behind the scenes as a JSON deserializer. With Retrofit, you can further customize operations with `RequestInterceptor` and `OkClient` to control caching behavior without the client knowing what's going on.

The less each object knows about what's going on behind the scenes, the easier it will be for Future You to manage changes in the app. This pattern can be used in a lot of other contexts; Retrofit is only one mechanism among many.

Behavioral Patterns

"So… how do I tell which class is responsible for what?" – Future You

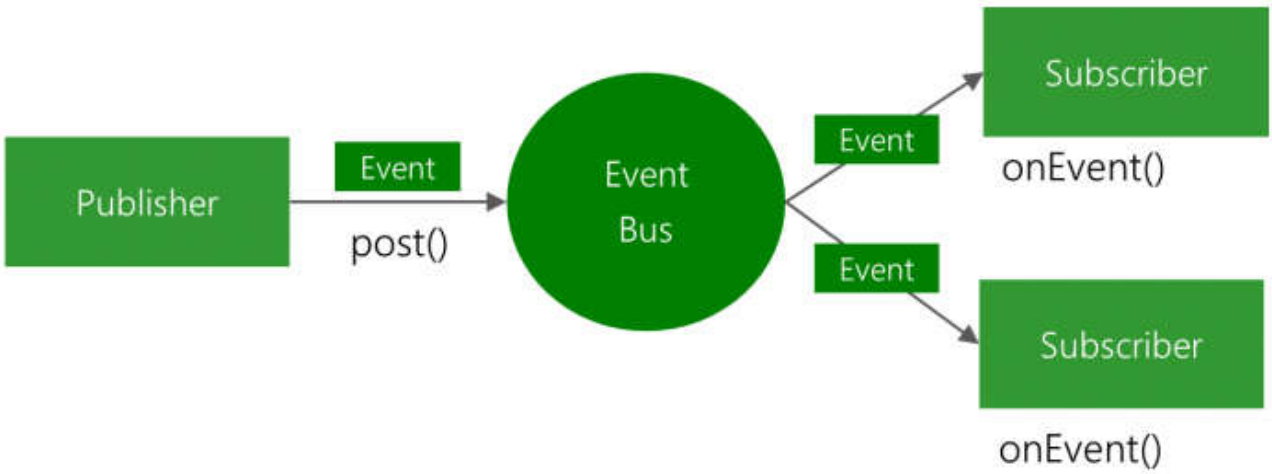
Behavioral patterns let you assign responsibility for different app functions; Future You can use them to navigate the structure and architecture of the project. These patterns can vary in scope, from the relationship between two objects to the entire architecture of your app. In many cases, the various behaviorial patterns are used together in the same app.

Command

When you order some excellent Saag Paneer at an Indian restaurant, you don't necessarily know which cook will prepare your dish; you only give your order to the waiter, who posts the order in the kitchen for the next available cook.

Similarly, the Command pattern lets you issue requests without knowing the receiver. You encapsulate a request as an object and send it off; deciding how to complete the request is an entirely separate mechanism.

Greenrobot's EventBus is a popular Android framework that supports this pattern in the following manner:



An `Event` is a command-style object that can be triggered by user input, server data, or pretty much anything else in your app. You can create specific subclasses which carry

data as well:

```
public class MySpecificEvent { /* Additional fields if needed */ }
```

After defining your event, you obtain an instance of `EventBus`and register an object as a subscriber:

```
eventBus.register(this);
```

Now that the object is a subscriber, tell it what type of event to subscribe to and what it should do when it receives one:

```
public void onEvent(MySpecificEvent event) {/* Do something */};
```

Finally, create and post one of those events based on your criteria:

```
eventBus.post(event);
```

Since so much of this pattern works its magic at run-time, Future You might have a little trouble tracing this pattern unless you have good test coverage. Still, a well-designed flow of commands balances out the readability and should be easy enough to follow at a later date.

Observer

The Observer pattern defines a one-to-many dependency between objects. When one object changes state, all of its dependents are notified and updated automatically.

This is a versatile pattern; you can use it for operations of indeterminate time, such as API calls. You can also use it to respond to user input.

The RxAndroid framework (aka Reactive Android) will let you implement this pattern throughout your app:

```
apiService.getData(someData)
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe (/* an Observer */);
```

In short, you define `Observable` objects that will *emit* values. These values can emit all at once, as a continuous stream, or at any rate and duration.

`Subscriber` objects will *listen* for these values and react to them as they arrive. For example, you can open a subscription when you make an API call, listen for the response from the server, and react accordingly.

Model View Controller

Model View Controller refers to the current reigning architectural pattern across several platforms; it's particularly easy to set your project up in this way on Android. It refers to the three divisions of classes used in this pattern:

- **Model:** your data classes. If you have `User` or `Product`objects, these "model" the real world.
- **View:** your visual classes. Everything the user sees falls under this category.
- **Controller:** the glue between the two. It updates the view, takes user input, and makes changes to the model.

Dividing your code between these three categories will go a long way toward making your code decoupled and reusable.

Future You will eventually get a request from the client to add a new screen to the app, but simply using the existing data in the app; following the MVC paradigm means Future You can easily re-use the same models and only change the views. Or perhaps the client will ask Future You to move that fancy widget from the home screen to the detail screen. Separating your view logic makes this an easy task.

Additionally, moving as much layout and resource logic as possible into Android XML keeps your View layer clean and tidy. Nice!

You may have to do some drawing in Java from time to time, in which case it will help to separate the drawing operations from your activity and fragment classes.

Over time, you'll find making architectural decisions becomes easier under MVC and Future You can more easily solve issues as they arise.

Model View ViewModel

This unfortunately-quite-confusingly-named architectural pattern is similar to the MVC pattern; the Model and View components are the same. The ViewModel object is the "glue" between the model and view layers, but operates differently than the Controller component. Instead, it exposes commands for the view and binds the view to the model. When the model updates, the corresponding views update as well via the data binding. Similarly, as the user interacts with the view, the bindings work in the opposite direction to automatically update the model. This reactive pattern removes a lot of glue code.

The MVVM pattern is trending upwards in popularity but is still a fairly recent addition to the pattern library. Future You would love it if you kept your eye on this one! :]

Where to Go From Here?

While it feels great to keep abreast of the latest flashy APIs, keeping your apps updated can quickly lead to redesign-fatigue. Investing in software design patterns early on will