Code Smell and Refactoring

Duplicate Code: FilledButton

Grace uses the `FilledButton.tonalIcon` factory constructor to build primary buttons across the platform. In the current state, there are only two instances of this button, but as buttons are a common widget that are likely to become redundant as I rebuild the same button, leading to **duplicate code**.

Refactoring

To manage this complexity and resolve the code smell, I wrote a widget `PrimaryIconButton` that takes in arguments to build a button that looks consistent across the platform.

This class is available at `lib/widgets/buttons/primary_button.dart`.

Duplicate Code: SnackBar

Grace uses the `SnackBar` widget to build toast-style widgets invoked in multiple spaces across the platform. In the current state, there are only two instances of this widget, but as `SnackBar` is a common widget, it is possible it could become redundant as I rebuild it across multiple classes, leading to **duplicate code**.

Refactoring

To manage this complexity and reoslve the code smell, I wrote a class `GraceNotification` with two static methods that accept arguments to build and return `SnackBar` widgets that look consistent across the platform.

This class is available at `lib/widgets/notifications/notification.dart`.

Large Class: Scaffold

Grace is built with two primary structural layouts - one for desktop, one for mobile. Building a single `Scaffold` widget to house both the desktop and mobile layout would be sensible except that it could lead to a **large class** and be difficult to read and maintain.

Refactoring

To manage this complexity and resolve the code smell, I wrote a class `GraceBaseScaffold` which unifies the desktop and mobile layouts by conditionally rendering `GraceDesktopScaffold`

or `GraceMobileScaffold` and passing down any available arguments.

These classes are available at `lib/layout/scaffold/`.

Large Class: HomeScreen

The Grace home screen widget tree is nested deeply - `HomeScreen` contains `CollectionSnapshot`, and `CollectionSnapshot` contains `Resource`, and those are all nested within other widgets. This is a necessary structure but led to a **large class**.

Refactoring

To manage this complexity and resolve the code smell, I wrote `HomeScreen`, `CollectionSnapshot`, and `Resource`, each in their own class, and each accepting arguments to build their respective widgets in a more sustainable class size.

These classes are available at `lib/screens/home.dart` and `lib/widgets/collections/`.

Long Method, Redundant Code: ParseUtilities

The internal `GraceApi` interface enables a facade to fetch `Book` data from multiple sources in a series of sequential calls, but these sequential calls rely on one another to gather alphanumeric identifiers to make each subsequent call. These alphanumeric identifiers are structured in a way that has to be altered before the next query can be made, and thus a **long method** with **redudant code** was written.

Refactoring

To manage this complexity and resolve the code smell, I wrote `ParseUtilities` which contains a method `formatResourceId` which strips the received identifier of unhelpful syntax and leaves only the data relevant to the subsequent call.

This class is available at `lib/utilities/parse_utilities.dart`.

Middle Man: Scaffold

As mentioned above, Grace is built with two primary structural layouts that are unified and conditionally rendered by `GraceBaseScaffold`, but while this is readable and maintainable, it may not be best practice. As `GraceBaseScaffold` does not provide meaningful change to the application - only delegates based on screen width - it may be an unnecessary **middle man**.

Incomplete Library Class: Firestore

Grace uses Firestore to store user collection data, and while the Firestore Dart library is robust, it does lack in some common, easy-to-use features - namely, checking that a new document was successfully inserted. The Firestore `add` method returns `Future<DocumentReference>` but does not provide a simple check to ensure the document was stored correctly. There are other methods to storing data in Firestore which can be looked into, but using this recommended approach, there is a seemingly **incomplete library class**.

Shotgun Surgery: Colors

Grace does not use a custom `Theme` because I intended to use Google's in-built theme, but after working with it for some time, decided that it would be easier to declare color values in each class. While this was the easier decision, I have declared constant color values across nearly every class containing a widget, so updating the platform theme to make it consistent would require **shotgot surgery**.

Shotgun Surgery: TextStyle

As mentioned above, Grace does not use a custom `Theme`, which resulted in declared constant `TextStyle` values across nearly every class containing a widget. As with colors, the result of this design is that updating the platform theme to make it consistent would require **shotgot surgery**.

Shotgun Surgery: Spacing

Grace is responsive to both desktop and mobile layouts and is spaced accordingly (e.g., padding and margin) but these space values are declared as constants in nearly every class containing a widget. To provide parity across all widgets would now require **shotgot surgery** and extracting the values as project-level constants.