An Architectural Framework for Consistent UI in Android App Development

Abdul-Rahman Mawlood-Yunis
Department of Computer Science and Physics
Wilfrid Laurier University
Waterloo, Canada
amawloodyunis@wlu.ca

***Abstract*—The User Interface (UI) is an essential component in the development of interactive apps and web applications. In this study, we present an architectural framework designed to simplify the creation of a consistent UI for Android apps. The approach is both straightforward and innovative, utilizing well-established object-oriented programming concepts, such as abstraction and inheritance, to enable the development of flexible and scalable apps. Apps built using this framework are developed by extending abstract and generic concepts to ensure uniformity across the entire interface. We outline the key components of this architecture, provide sample code for implementation, and present an app developed using this framework to highlight its practical benefits. This paper offers two main contributions: accelerating app development and enabling the creation of consistent user interface designs that enhance both visual appeal and overall user experience. While the framework is initially designed for Android app development, its applicability extends to web development and can be used by tools focused on minimizing code complexity while supporting multiplatform compatibility, including web applications.**

Keywords-Software design and architecture; app development; design pattern; software engineering; software reuse; Android; Web development

I. Introduction

The user interface (UI) holds significant importance in interactive apps and application development, particularly in mobile apps. An attractive and user-friendly visual interface becomes increasingly crucial and plays an essential role in determining their success [1]–[3]. Extensive research in software engineering [4]–[8], design patterns [9]–[13], software architecture [14]–[16], human interaction [17]–[19], and related fields have been dedicated to the proper UI design for interactive apps and applications. Leading APIs [20] offer a range of components and classes to facilitate this goal [21]–[23]. For instance, the Android API provides the Fragment component [24], which enhances modular app development and UI flexibility. The Fragment is a reusable, self-contained portion of an activity's or UI screen that can be added, removed, or replaced dynamically, allowing for a more modular approach to app design. By utilizing the Android Fragment component, developers can design and develop each page or view of the app separately, promoting efficient reuse and customization. This is because Fragments allow developers to break down the UI into smaller, independent sections that can be reused across different parts of the app. For example, a Fragment developed for one screen (e.g., a login form) can be reused in multiple screens that require similar functionality, reducing redundancy and speeding up development. Additionally, Fragments can be customized to display different content or behavior based on the context, such as device orientation or user interaction, providing greater flexibility. Fragments achieve this modularity by serving as containers that can have UI elements and logic, which can be embedded into different parts of the app's layout. They allow developers to partition the app screen into multiple independent areas, each capable of hosting its own content or functionality. These Fragments can be attached to an activity's view and positioned to occupy one or more sections of the screen. By reusing these Fragments across multiple views, developers can maintain flexibility while reducing redundancy in the UI design, which leads to more efficient development and maintenance. This modular design also makes it easier to update or change individual parts of the UI without affecting the entire screen, further streamlining the development process.

While Fragment work focuses on designing a portion of the page or view to be reused, this work takes a broader approach by focusing on reusing the whole or important components of the page as the user navigates between different screens of an app. In other words, this work develops an architectural framework that enables persistent UI across app screens, ensuring a cohesive user experience as users move from one screen to another. This framework can be utilized throughout the app development process, allowing developers to create a consistent and dynamic interface that adapts to various contexts and navigation flows.

The suggested architecture does not serve as a replacement for the use of Fragment in Android app development, as Fragments offer more functionality beyond screen reuse. However, our proposed architecture surpasses Fragments when it comes to screen reuse. Here, the focus isn't solely on creating a sizeable component for reusing across app pages. Instead, we describe a framework for reusing as many components as needed across app pages and enable each component to function differently on various pages. This will be accomplished by developing a generic and abstract component containing the components to be reused and operating differently across app pages. The framework will accelerate the app development process and facilitate the creation of persistent UI creation, thereby improving the overall appearance and user experience of the apps. Moreover, the framework's applicability is not confined to Android-enabled devices. Although initially designed for Android app development, it can also be adapted for web development and tools aimed at reducing code complexity while ensuring compatibility across multiple platforms, including both mobile and web applications. Additionally, the knowledge and principles derived from this framework are transferable to a wide range of application development contexts, thereby contributing to the broader advancement of software engineering practices.

The paper is organized as follows. Section II provides an overview of the proposed architecture. Section III includes implementation examples for all components of the architecture. In Section IV, we present an app developed using this architecture and discuss the testing conducted to validate its effectiveness. Finally, Section V concludes the paper and highlights potential directions for future research.

II. Overview of The Proposed Architecture

In this section, we describe the building blocks of the proposed architecture and their main responsibilities.

A. Base View

The first step is to create a base class (also called a superclass) in Object-Oriented Programming (OOP) that acts as the foundational template for the app's screens. This base class provides a structure that defines the common components and functionality that all the screens in the app will share. However, the base class itself is abstract, meaning that it is not intended to be directly used or displayed. Instead, it serves as a container for shared elements, providing a common foundation upon which all individual screens will be built.

Each screen in the app extends this base class, meaning that the screen inherits the structure and components from the base class. However, each screen can customize or define the inherited components to suit the specific needs of that screen. This allows for consistency across screens (through shared components), while still allowing flexibility and customization in terms of design and functionality.

To define the generic view and shared behavior for all screens, the base class includes two key methods:

1) An abstract method: This method has no implementation in the base class itself, but must be implemented by each subclass (screen). It acts as a placeholder for screen-specific functionality.
2) A regular method with an empty body: This method is defined in the base class, but does not perform any actions initially. Subclasses can choose to override this method or leave it as is, depending on their needs.

Additionally, the base class defines a layout that includes UI components (such as buttons, text fields, or navigation elements) that need to appear across all the app's screens. By placing these components in the base class, every screen that extends it will automatically inherit these shared elements. This ensures that there is a consistent look and feel throughout the app while still allowing each screen to define its own unique content.

By using this approach, developers can create custom screens (subclasses) that fit their specific needs, while maintaining a persistent look (consistent UI elements) and shared functionality (common methods and components) across the app. This promotes reusability, reduces redundancy, and simplifies the overall app development process.

B. Derived Views

Derived views are concrete implementations or materialized views that extend the abstract base view. In other words, these views represent actual screens or pages in the app, whereas the base view acts as a template or blueprint. To create a new screen (view) for the app, a new page needs to be designed. This new page inherits components, methods, and basic layout properties from the base view, but it also adds custom components and functionality to create a unique screen tailored to its specific needs. The process of implementing a derived view involves the following steps:

1) Reusing the base layout container: The first step is to reuse the layout container defined in the base view. This container holds only the components that are common and need to exist on every page. These are shared elements, such as headers, footers, or navigation bars, that appear across multiple screens. By reusing this container, you ensure that these consistent elements are present on every screen, helping to maintain a uniform look and feel across the app.
2) Creating a new layout for the derived view: Next, you create a custom layout for the derived view. This layout will contain the specific components and content for this screen. You then place this new layout into the space inherited from the base layout container. The inherited space ensures that your derived layout fits into the structure and design already established by the base view, preserving the overall app's consistency while allowing for unique content on each screen.
3) Implementing abstract methods: The base view has abstract methods—these are methods that have no implementation in the base view itself but must be implemented in the derived view. These methods act as placeholders, requiring the derived view to define specific behavior. For example, an abstract method in the base class returns the content view, and the derived view provides the logic for how that should be done for its specific screen.
4) Completing empty methods inherited from the base view: In addition to abstract methods, the base view has regular methods with an empty body. These methods are already given a structure but do not perform any actions in the base view. The derived view must complete the implementation of these empty methods by adding the necessary logic. For example, the base class might define a method to initialize UI elements, and the derived view would add specific code to populate those elements with data or behavior relevant to the screen it represents.

C. Customizing Views Behaviour

While every page of an app serves distinct purposes and boasts unique features, reusing components from the base view ensures a consistent appearance and functionality. A decision must be made regarding reusable components, as each page's requirements must be developed accordingly. For instance, a toolbar component can be placed in the base view and inherited across all app pages, yet the action functionalities of each toolbar item may vary from one page to another. In other words, when the toolbar items are clicked, the actions executed on page one will differ from those performed on pages two, three, and beyond.

Figure 1 illustrates the proposed architecture, including an example of a component intended for reuse. The figure displays the following components:

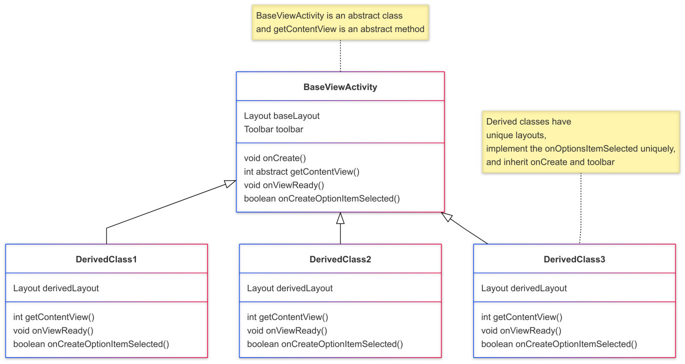

Figure 1. Class diagram for the proposed framework.

1) An abstract superclass featuring two properties: baseLayout, of type Layout, and toolbar, of type Toolbar. This class also encompasses four methods: onCreate(), onViewReady(), getView(), and onOptionMenuSelected().
2) Derived classes override the onViewReady(), getView(), and onOptionMenuSelection() methods, each providing its own implementation. Additionally, each derived class boasts a distinctive layout property, using getView() to access and initialize the local layout properties.

III. The Implementation of The Proposed Architecture

In this section, we will list the architecture components, explain their roles and usage in the overall design in more detail, and provide sample code to demonstrate how they can be implemented.

A. *The Base View Implementation*

As previously mentioned, a key element of this architecture is the base view. In Android development, this base view can be represented by an abstract class like BaseViewActivity, which extends AppCompatActivity. AppCompatActivity serves as a foundational class in the Android API, providing various built-in features utilized by screens. Consequently, it is extended and reused when new app screens are created.

The **BaseViewActivity** class is abstract and defines essential properties that are common to all apps, such as the layout and necessary components. The list 1 provides a template for defining the BaseViewActivity class. Additional details about other crucial methods and the app's layout are discussed in the following subsections. The ellipses (...) in Listing 1 represent the parts of the code that need to be implemented, which will depend on the specific requirements of the app.

```
public abstract class BaseViewActivity extends
      AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_base);
    ...
    onViewReady(savedInstanceState, getIntent());
  }
  // To be used by child activities.
  protected void onViewReady(Bundle savedInstanceState,
      Intent intent) {}
  protected abstract int getContentView ();
  @Override
  public boolean onOptionsItemSelected(MenuItem menuItem) {
   ...
 }
}
```

Listing 1. Base class template

More details and an overview of the base implementation are provided in the following.

1) OnViewReady Method: Within the BaseViewActivity class, we've established a method named onViewReady(), which is invoked within the onCreate() method subsequent to calling setContentView(). The method currently contains no code, leaving its implementation to be completed by the derived class. This enables derived classes or screens to append new widgets or modify the layout of the base class. The complete implementation of this method will be demonstrated when we define the derived classes in part B, item 2 of this section.

2) getContentView Method: We have defined another method called getContentView(). This method is an abstract method that needs to be implemented by the Derived classes. This method is a helper method and it will be used by the onViewReady() method to update or make changes to the layout of the base class.

3) Base Layout or Base Content: A layout for the BaseViewActvity class or its content is created and is shown in Listing 2. The layout is an XML file that will be transformed into the programming code and integrated with the rest of the app code at run time. The code statement below inside the onCreate method of the BaseViewActvity class will take care of this step, i.e., setting the layout for the base view by calling this method. setContentView(R.layout.activity_base);

The layout has two mandatory parts. A component(s) that will be reused across all the screens of the app, in this case, is a toolbar, and an inner layout. The inner layout is empty and, in this case of type linear layout. In other words, no widgets or components are attached to the inner layout and when items are attached to this layout, they will be arranged sequentially from left to right top down and take the whole screen of the base view. These properties allow the derived screen, the classes that inherit this layout, to update its content and attach widgets to it. The layout is shown in Listing 2 where the toolbar and inner layouts are shown.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
tools:context=".BaseViewActvity ">
    <androidx.appcompat.widget.Toolbar
android:id="@+id/in_base_my_toolbar"
android:layout_width="match_parent"
android:layout_height="?attr/actionBarSize" />
    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/
             android"
        xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
android:id="@+id/toolbarIDInbase">
    </LinearLayout>
</LinearLayout>
```

Listing 2. The layout for the Base class

4) Reused Components: An important part of this architecture which aims to enable persistent looks and feel is component reuse. One or more components can be included in the base view and be reused in various screens of the app. To demonstrate this concept one component, a toolbar, is added to the base view layout and the statement below will add or set its corresponding object code to the definition of the BaseViewActvity class.

```
  setSupportActionBar(toolbar);
```

In addition to reusing components, a crucial concept in this architecture is the ability to redefine the behavior of objects defined in the base view. An example of object behaviour definition for the toolbar is shown in Listing 3. This implementation serves as a default behaivour and it will be overridden in the derived class to behave differently and to become more relevant to the new context while using the same toolbar buttons and menu items defined in the base view.

```
public boolean onOptionsItemSelected(MenuItem menuItem) {
  int id = menuItem.getItemId();
  switch (id) {
    case R.id.action_one:
      Intent intent = new Intent(BaseViewActvity.this,
           ChildActivity.class);
      startActivity(intent);
      break;
    case R.id.action_two:
      snackbar.setText("You are at home").show();
      break;
    case R.id.action_three:
      Uri webpage = Uri.parse("https://www.canada.ca/");
      intent = new Intent(Intent.ACTION_VIEW, webpage);
      if (intent.resolveActivity(getPackageManager()) !=
           null) {
        startActivity(intent);
      }
      break;
    case R.id.action_about:
      Toast
          .makeText(this,
              "Version 1.0,"
                  + "developer_information",
              Toast.LENGTH_LONG).show();
  }
  Return true;
}
```

Listing 3. A behaviour implementation of a component in the base view that will be overridden in the derived classes

5) A Complete code for Base View: Combining all the code snippets provided above creates a comprehensive template for the base view class, featuring a single component, the toolbar, intended for reuse across the app's screens. Listing 4 illustrates this template.

```
public abstract class BaseViewActvity extends
     AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_base);
    Toolbar myToolbar = findViewById(R.id.
         in_base_my_toolbar);
    setSupportActionBar(myToolbar);
    onViewReady(savedInstanceState, getIntent());
  }
  protected void onViewReady(Bundle savedInstanceState,
       Intent intent) {
    // To be used by child activities.
  }
  protected abstract int getContentView();
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    // invoked automatically by activity
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main_activity_actions, menu);
    return true;
  }
  public boolean onOptionsItemSelected(MenuItem menuItem) {
    int id = enuItem.getItemId();
    switch (id) {
      case R.id.action_one:
        Intent intent = new Intent(BaseViewActvity.this,
             ChildActivity.class);
        startActivity(intent);
        break;
      case R.id.action_two:
        snackbar.setText("You are at home").show();
        break;
      case R.id.action_three:
        Uri webpage = Uri.parse("a_url");
        intent = new Intent(Intent.ACTION_VIEW, webpage);
        if (intent.resolveActivity(getPackageManager()) !=
             null) {
          startActivity(intent);
        }
        break;
      case R.id.action_about:
        Toast
            .makeText(this,
                "Version 1.0,"
                    + "developer_name",
                Toast.LENGTH_LONG)
            .show();
    }
    return true;
  }\\ end of the method
}\\ end of the class
```

Listing 4. A template for the Base View

B. Implementations of The Derived Views

Every useful app is made of more than one screen. Following the proposed architecture, as many screens as needed can be created with similar feels and looks by extending the base view and thus reusing the inherited components. To create new screens using the existing base view the following need to be done.

- Defining a new class(s) by extending the base class.
- Define layout(s) for the derived classes or screens to replace the empty inner layout from the base view.
- Complete the definition of the inherited methods.
- Define new behaviors for the inherited component.

These steps are described in the following subsection along with sample codes on how to implement them

1) Defining New Classes: To develop a complete app, you'll need to define one or more classes that utilize the base view, BaseViewActivity, as their superclass. The number of classes depends on the required screens. An example of such a derived class, DerivedClass1, is provided in listing 5.

DerivedClass1 inherits the onCreate() method from the base class instead of implementing it directly. This inheritance allows DerivedClass1 to replace the base view layout with its own. This substitution occurs because the onViewReady() method is invoked within the onCreate() method, which is triggered when the screen is being loaded. Refer to the BaseViewActivity class for the definition of the onCreate() method and the location of where onViewReady() is called.

It's worth emphasizing that loading the screen entails calling the inherited and hidden onCreate() method within the derived class, followed by invoking the onViewReady() method. This process ultimately leads to displaying a personalized layout for the newly created screen.

```
public class DerivedClass1 extends BaseViewActvity  {
  LinearLayout linearLayout;
     ...
}
```

Listing 5. A derived view class definition header

2) Define new layouts for derived classes: The code snippet presented in Listing 6 depicts the layout for a derived class. This layout constitutes the content of the newly created screen and differs from the layout defined in the base view. It's important to note that this serves as merely an example of a newly created screen, and the actual screen will vary depending on the specific requirements of the app. The provision of a placeholder for the layout in the main or base view allows each newly created screen to utilize this space and construct its desired screen while also benefiting from the reusable components at the same time.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/
        android"
    xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
android:layout_marginTop="40dp">
    <ImageView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/ic_favorite_border_black_24dp"/>
</LinearLayout>
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
```

Listing 6. A layout example for a derived class

3) Overriding The onViewReady Method: The onCreate method isn't explicitly defined in derived classes; however, it's inherited from the base class and is automatically invoked upon opening the derived screens, i.e., executing the derived class code. As the onViewReady method is invoked within the inherited onCreate method, this provides an opportunity to override it in a manner that allows for setting a distinct layout for the derived screens. In other words, the onViewReady method takes on the responsibility of the onCreate method inside the derived classes. This explains why the onViewReady method definition is empty in the base class, and code which typically placed inside the onCreate method is now placed inside the onViewReady method.

An implementation of onViewReady is illustrated in Listing 7, where a new layout is loaded by onViewReady to create a custom view for a derived screen.

```
@Override
protected void onViewReady(Bundle savedInstanceState,
     Intent intent) {
  setTitle("First Screen");
  super.onViewReady(savedInstanceState, intent);
  linearLayout = findViewById(R.id.baseLayout);
  LayoutInflater layoutInflater = LayoutInflater.from(
       DerivedClass1.this);
  layoutInflater.inflate(getContentView(), linearLayout,
       true);
}
```

Listing 7. onViewReady implementation by a derived class

The code in the Listing 7. Does the following:

a. Get the container, the linear layout container, that has been defined in the layout for the base class. It is an empty container, and we can add views to it. This is achieved using this code statement:
findViewById(R.id.baseLayout);
b. The retrieved base linear layout is used to initialize a local liner layout instance variable of the derived classes. This is achieved using this code statement:
linearLayout = findViewById(R.id.baseLayout);
c. The retrieved layout from the previous step is put inside the empty linear layout container from the base class. This is achieved using the following lines of code:

```
LayoutInflater layoutInflater =
 LayoutInflater.from(DerivedClass1.this);
layoutInflater.inflate(getContentView(),
   linearLayout, true);
```

4) Overriding the getView Method: The getContentVeiw is an accessor method used by the onViewReady method to retrieve the locally defined layout for a derived class. The method definition is presented in Listing 8 and its usage is shown in Listing 7.

```
@Override
protected int getContentView() {
 // layout for a derived class
 return R.layout.activity_main;
}
```

Listing 8. getContentView implementation

5) Override the behavior of shared components: In the example under consideration here, we opted to utilize the toolbar consistently across the derived views while altering its functionality by overriding the onOptionItemSelected() method. Essentially, this means that when the toolbar items are clicked, the actions performed on each derived screen can be different from the default behavior and each other. The code snippet in Listing 9 illustrates such an implementation, where the code contained within each switch case is different from what has been included within the switch statement for the base classes and each derived class. It's important to mention that the onOptionItemSelected method can be implemented within the base view. This implementation will act as a default behavior inherited by derived classes. Alternatively, it can be left empty, like the layout in the base view. In either scenario, the method can be overridden in the derived classes to offer customized functionality. In this example, we've chosen to provide a default implementation in the base view class.

```
public boolean onOptionsItemSelected
  (MenuItem menuItem) {
  int id =menuItem.getItemId();
  String phoneNumber = "613 000 0000";
  switch (id) {
  case R.id.action_one:
      Intent intent = new Intent(Intent.ACTION_DIAL);
      intent.setData(Uri.parse("tel:" + phoneNumber));
     if (intent.resolveActivity(getPackageManager()) !=
         null) {
           startActivity(intent);
       }
   break;

 case R.id.action_two:
 intent = new Intent(ChildActivity.this,
                  NewMainActivity.class);
   startActivity(intent);
  break;

 case R.id.action_three:
  Uri webpage = Uri.parse( "a_url");
  Intent i = new Intent(Intent.ACTION_VIEW);
  i.setData(webpage);
   startActivity(i);
  break;

case R.id.action_about:
     Toast.makeText(this,
      "Version 1.0," + " Developeer_information",
       Toast.LENGTH_LONG).show();
  }
   return true;
}
```

Listing 9. Toolbar behaviour implementation for a derived class

6) An Example of a Derived Class Implementation: The code provided in Listing 10 serves as an instance of a fully developed derived class, encompassing the steps described from 1 to 5.

```
public class ChildActivity extends BaseViewActvity {
    LinearLayout linearLayout;

    @Override
    protected int getContentView() {
        return R.layout.activity_child;
    }

    @Override
    protected void onViewReady(Bundle savedInstanceState,
         Intent intent) {
        super.onViewReady(savedInstanceState, intent);
        linearLayout = findViewById(R.id.baseLayout);
        LayoutInflater layoutInflater = LayoutInflater.from
             (ChildActivity.this);
        layoutInflater.inflate(getContentView(),
             linearLayout, true);
    }

    public boolean onOptionsItemSelected(MenuItem menuItem)
          {
        int id = menuItem.getItemId();
        String phoneNumber = "613 000 0000";
        switch (id) {
        case R.id.action_one:
            Uri webpage = Uri.parse("a_url");
            Intent intent = new Intent(Intent.ACTION_VIEW,
                 webpage);
            if (intent.resolveActivity(getPackageManager())
                 != null) {
                startActivity(intent);
            }
            break;
        case R.id.action_two:
            intent = new Intent(ChildActivity.this,
                 NewMainActivity.class);
            startActivity(intent);
            break;
        case R.id.action_three:
            intent = new Intent(Intent.ACTION_DIAL);
            intent.setData(Uri.parse("tel:" + phoneNumber))
                 ;
            if (intent.resolveActivity(getPackageManager())
                 != null) {
                startActivity(intent);
            }
            break;
        case R.id.action_about:
            Toast.makeText(this, "Version 1.0," + "
                 Developer_information", Toast.LENGTH_LONG).
                 show();
        }
        return true;
    }
}
```

Listing 10. An instance of a fully developed derived class

IV. Running and Testing The Proposed Architecture

To evaluate the proposed architecture, we developed an app that can be downloaded at [25]. The app was created in Java using Android Studio. Using this app, two sets of tests were conducted.

In the first set, multiple screens were created. Consistent with the proposed architecture, none of these screens directly utilized the 'onCreate()' method, which serves a role similar to the constructor method in object-oriented programming and is essential for object instantiation. Instead, all screens inherited the 'onCreate()' method from an abstract base view and implemented the 'onViewReady()' method locally to instantiate three distinct screens. These screens are depicted in Figs. 2-4, and the concept of attaching a customized page to the base page is illustrated in Figure 5.

Second, following the approach used for screen creation, the proposed architecture was employed to develop a reusable component. This component can be inherited and integrated across all screens, ensuring a consistent and persistent user interface throughout the application. As demonstrated in the app screens, each screen features a uniform toolbar, depicted in Figure 6.

Although the toolbars appear identical across screens, their functionality varies significantly, adapting to the specific requirements of each page. For example, on the first screen, selecting toolbar items 1, 2, 3, and 4 (as shown in Figure 6) triggers the following actions:

1. Opens a second page, illustrated in Figure 4.
2. Displays a message indicating that you are on the home page.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

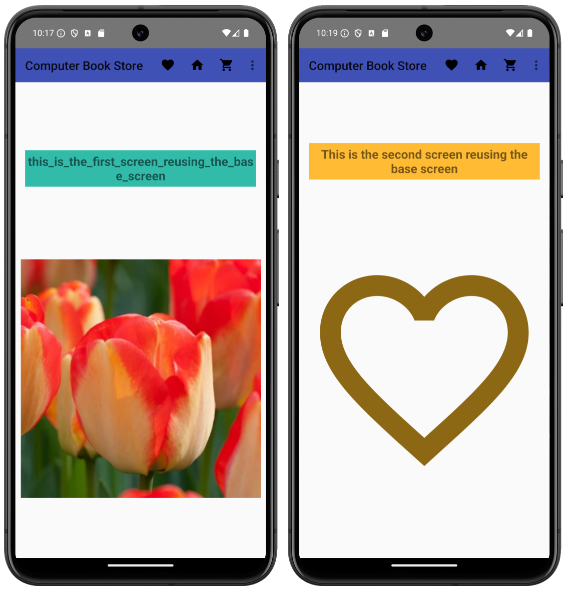
Computer Book Store
this_is_the_first_screen_reusing_the_base_screen
Computer Book Store
This is the second screen reusing the base screen

Figure 2. and 3, examples of two pages created using base page

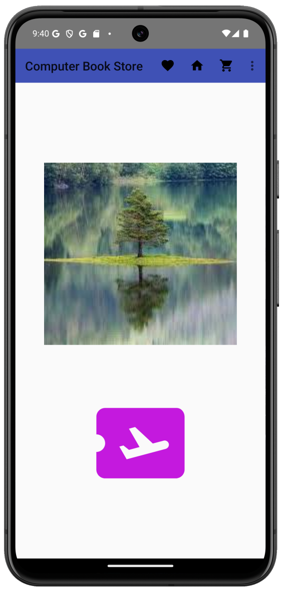

Figure 4. Another example of custom page using the base page

3. Launches a web page on the internet; an example is shown in Figure 7.
4. Opens an "About" window, providing information such as details about the app's author.

On the second and third screens, the functionality of these toolbar items differs. For instance, on the second page, clicking:

1. Opens a new app page. This demonstrates the consistent use of the first toolbar item to open a new app page.
2. Navigates back to the home page or the previous page.
3. Launches a phone-dialing interface, as shown in Figure 8, displaying how the same toolbar item can trigger different functionalities depending on the page.

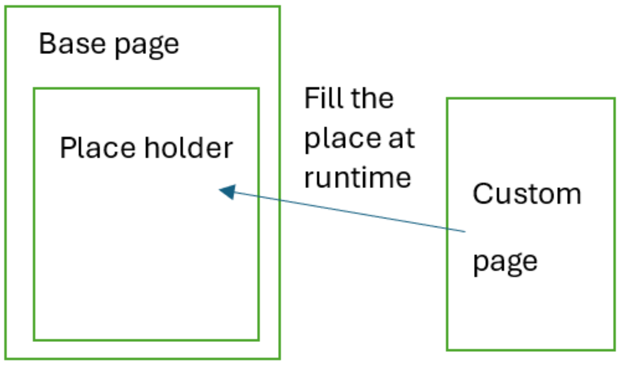

Figure 5. Attaching a customized page into the base page

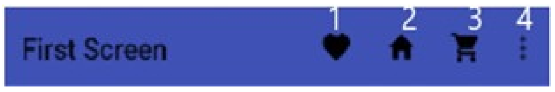

Figure 6. Page toolbar and its items

4. Opens an "About" window, presenting the same content presented on the first screen. This illustrates the option to assign a common functionality to a toolbar item across all screens.

The navigation flow described above is summarized in Figure 9. The ability to maintain a consistent user interface across all screens while enabling contextual behavior through the extension of a generic screen template highlights the feasibility and flexibility of the proposed architecture for app and application development.

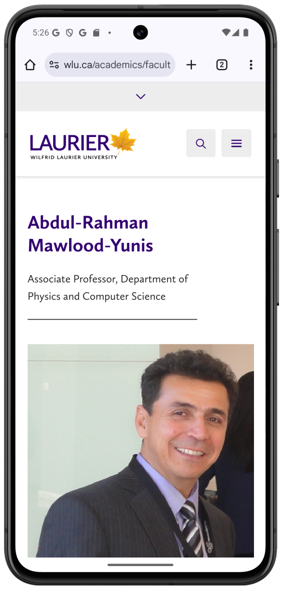
wlu.ca/academics/facult
LAURIER
WILFRID LAURIER UNIVERSITY
Abdul-Rahman Mawlood-Yunis
Associate Professor, Department of Physics and Computer Science

Figure 7. An example of web page lunched by clicking third item on the first page

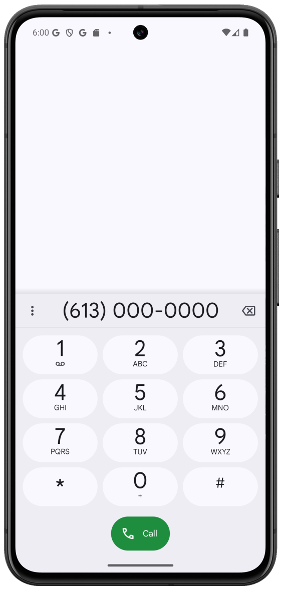

Figure 8. Example of phone-dialing interface

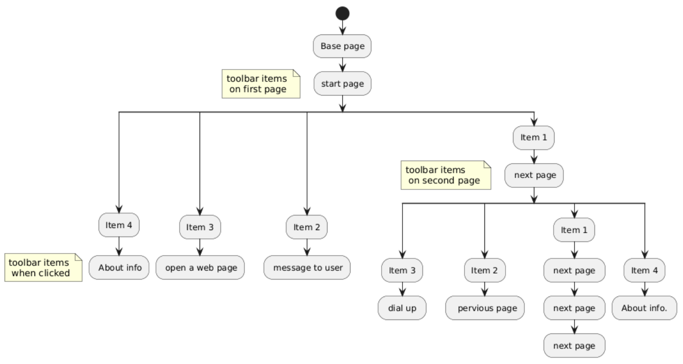

Figure 9. The app structure and component functionalities

V. Conclusion and Future Works

Drawing upon techniques from object-oriented programming, we introduced an app architecture aimed at facilitating persistent UI development in Android apps, thus enhancing the overall appearance and user experience of apps, an essential factor for their success. Detailed descriptions of the architecture components, along with an example of how to implement each component, have been provided. Furthermore, we tested the effectiveness of the architecture by building an app, highlighting its advantages and feasibility.

It is important to note that the applicability of the framework is not limited to devices enabled with Android. Although initially designed for Android app development, it can also be adapted for web development and used by tools and frameworks that aim to reduce code base and support multiple platform development, including mobile and web applications. Additionally, the knowledge and principles derived from this framework are transferable to a wide range of application development contexts, thereby contributing to the broader advancement of software engineering practices.

This work can be expanded in various ways. Much like the way we extended the base view using a single hierarchy, each derived class can undergo further extension, using multilevel class inheritance. This approach enables the development of complex and sophisticated applications with fewer codes and a persistent user interface.

We chose a toolbar to demonstrate the ability to define a component once and reuse it across different pages, each page featuring unique functionalities for the component. In future iterations, one of the toolbar items could help navigate between different app pages. Clicking on a toolbar item might activate a pop-up menu that displays all navigation options. We consider this to be viable future work, as smooth navigation is crucial for any app's success.

We believe that this work lays the groundwork for further utilization of object-oriented programming features in Android app development, enabling efficient app design and implementation. By fostering consistency, scalability, and reusability, this architecture not only accelerates development, but also paves the way for innovative advancements in both mobile and web application development.

References

[1] C. Zarmer and J. Johnson, "User interface tools: Past, present, and future trends," Hewlett-Packard Laboratories, 1990.

[2] I. Qasim, F. Azam, M. W. Anwar, H. Tufail, and T. Qasim, "Mobile User Interface Development Techniques: A Systematic Literature Review," in 2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), 2018, pp. 1029–1034, doi: 10.1109/IEMCON.2018.8614764.

[3] H. Hu, et al., "Automated Mapping of Adaptive App GUIs from Phones to TVs," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 2, pp. 1–31, 2024, doi: 10.1145/3459610.

[4] H.-W. Six and J. Voss, "A software engineering perspective to the design of a user interface framework," in [1992] Proceedings. The Sixteenth Annual International Computer Software and Applications Conference, Chicago, IL, USA, 1992, pp. 128–134, doi: 10.1109/CMPSAC.1992.217591.

[5] C. E. Wills, "User interface design for the engineer," in Proceedings of ELECTRO '94, Boston, MA, USA, 1994, pp. 415–419, doi: 10.1109/ELECTR.1994.472682.

[6] V. Chernikov, "Approach to Rapid Software Design of Mobile Applications' User Interface," in 2018 23rd Conference of Open Innovations Association (FRUCT), 2018, pp. 1–7, doi: 10.23919/FRUCT.2018.8588030.

[7] M. J. Tsai and D. J. Chen, "Generating user interface for mobile phone devices using template-based approach and generic software framework," Journal of Information Science and Engineering, vol. 23, no. 4, pp. 1189–1211, 2007.

[8] A. Marcus, "User interface design and culture," in Usability and Internationalization of Information Technology, vol. 3, pp. 51–78, 2005.

[9] J. Arifin, E. M. Fasha, and M. A. Ayu, "User Interface Design Patterns for Marketplace Mobile Application," in IEEE 7th International Conference on Computing, Engineering and Design (ICCED), 2021, pp. 1–6, doi: 10.1109/ICCED53389.2021.9664843.

[10] E. G. Nilsson, "Design patterns for user interface for mobile applications," Advances in Engineering Software, vol. 40, no. 12, pp. 1318–1328, 2009.

[11] S. Hoober and E. Berkman, "Designing mobile interfaces: Patterns for interaction design," O'Reilly Media, Inc., 2011.

[12] T. Neil, "Mobile design pattern gallery: UI patterns for smartphone apps," O'Reilly Media, Inc., 2014.

[13] I. C. Morgado and A. C. Paiva, "The impact tool: Testing UI patterns on mobile applications," in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 876–881.

[14] P.H.J. Chong, P.L. So, P. Shum, X.J. Li and D. Goyal, “Design and implementation of user interface for mobile devices,” IEEE Transactions on Consumer Electronics, vol. 50, no. 4, pp. 1156–1161, 2004, doi: 10.1109/TCE.2004.1362513.

[15] M. J. Tsai and D. J. Chen, “Generating user interface for mobile phone devices using template-based approach and generic software framework,” Journal of Information Science and Engineering, vol. 23, no. 4, pp. 1189–1211, 2007.

[16] S. Wendler and D. Streitferdt, “The Impact of User Interface Patterns on Software Architecture Quality,” in The Ninth International Conference on Software Engineering Advances (ICSEA 14) IARIA, 2014, pp. 134–143.

[17] B. Shneiderman and C. Plaisant, “Designing the user interface: Strategies for effective human-computer interaction,” Pearson Education India, 2010.

[18] T. Mandel, “User/System Interface Design,” Encyclopedia of Information Systems, vol. 1, pp. 1–4, 2002.

[19] G. Briones-Villafuerte, A. Naula-Bone, M. Vaca-Cardenas, and L. Vaca-Cardenas, “User Interfaces Promoting Appropriate HCI: Systematic Literature Review,” Revista Ibérica de Sistemas e Tecnologias de Informação, no. E47, pp. 61–76, 2022.

[20] I. O. Suzanti, N. Fitriani, A. Jauhari, and A. Khozaimi, “REST API implementation on Android-based monitoring application,” in Journal of Physics: Conference Series, vol. 1569, no. 2, p. 022088, July 2020.

[21] S. Gao, L. Liu, Y. Liu, H. Liu, and Y. Wang, “API recommendation for the development of Android App features based on the knowledge mined from App stores,” Science of Computer Programming, vol. 202, p. 102556, 2021.

[22] Y. Wang, H. Liu, S. Gao, and X. Tang, “Animation2API: API recommendation for the implementation of Android UI animations,” IEEE Transactions on Software Engineering, vol. 49, no. 9, pp. 4411-4428, 2023.

[23] Google, “User Interface - Android Developers,” retrieved: [February, 2025]. Available: https://developer.android.com/develop/ui.

[24] Google, ”Fragment - Android Developers,” retrieved: [February, 2025]. Available: https://developer.android.com/reference/android/app/Fragment.

[25] ”Architectural Framework Based App” retrieved: [February, 2025]. Available: http://bohr.wlu.ca/amawloodyunis/framework/ArchitecturalFramework.zip.