On Modern User Interface Development and the Case of Android Applications

Antoine Bossard

Graduate School of Science, Kanagawa University, 3-27-1 Rokkakubashi, Kanagawa, Yokohama, Japan 221-8686.

* Corresponding author., Email: abossard@kanagawa-u.ac.jp

Manuscript submitted May18, 2023; revised June 20, 2023; accepted September 5, 2023.
doi: 10.17706/jsw.18.4.209-217

Abstract: The importance given to human-computer interactions (HCI) has been continuously increasing since the early days of computing. Advances in computer architecture and the related hardware components have notably enabled software developers to shift the focus from performance issues to usability issues, in other words towards the user. In this paper, we first give a brief presentation of the evolution of software development techniques for the implementation of user interfaces (UI). Then, we pedagogically document several *modern* user interface development best practices. The latter part of this investigation work has been conducted by considering the Android mobile application case and a very frequent application development scenario: view switching, responsive UI with asynchronous tasks and efficient data visualisation.

Key words: Software, computer, HMI, HCI, interaction, Java, mobile, smartphone, UI

1. Introduction

Software development has seen various evolutions from its inception, for instance the introduction of the object-oriented programming paradigm. User interface development has also seen significant changes, probably more than software development in general. This is partly due to the fact that there has been a priority shift from hardware to user: in the early days of computing, hardware resources were severely limited and that negatively impacted user interface development as fewer resources could then be allocated to UI issues. Nowadays, such hardware limitations being essentially gone, user-centred development is the cornerstone of software engineering [1] and advances related to the user interface have been numerous.

These advances with respect to UI development have been conducted in line with hardware evolution (e.g. from conventional displays to touch displays, batteries), software engineering evolution (e.g. programming paradigms) and, in general, our digital information consumption habits. As hinted by multiple previous research works (e.g. [2–4]), text-based user interfaces, such as text terminals (command prompts) and early window systems, must now indeed represent a tiny fraction of human-computer interactions.

In this paper, our objective is to briefly review the evolution of user interface development, that is from a programming point of view, and pedagogically document several modern UI development best practices. The latter is supported by considering the practical case of Android applications.

Whereas we focus herein on concrete, technical issues of user interface evolution, HCI challenges have been more broadly reviewed in [5], for instance considering cognitive, social and ethical matters. In addition, the importance of HCI issues and HCI trends over the 1994–2013 period have been analysed in [6].

The rest of this paper is organised as follows. We review in Section 2 earlier user interface development approaches. Then, we describe in Section 3 best practices with respect to modern UI programming in the case of Android applications. Obtained results are discussed in Section 4 before concluding the paper in Section 5.

2. Earlier User Interface Development Methodologies

We briefly review in this section earlier user interface development methodologies – it is not exhaustive. “Earlier” here means that they were introduced before the modern techniques described in the following sections, and not that they are not usable any more or deprecated.

Conventionally, the Microsoft Windows operating system and the corresponding development framework have provided user interface developers with direct access to native widgets, such as push buttons and list boxes, with the C function CreateWindow() (and its variants) of the Windows API. The prototype of this essential function is as follows: void CreateWindow(class, name, style, x, y, width, height, parent, menu, instance, parameter). The first argument of this function is the class name of the widget to create: it can be either user-defined in the case of a (child or parent) window, or predefined in the case of native UI widgets that are to be added to a (parent) window. Typical predefined window classes are BUTTON, COMBOBOX, EDIT, LISTBOX, SCROLLBAR and STATIC [7]. Note that the predefined window classes have been slightly changing over time with the evolution of the operating system features, for instance the class for the Microsoft Rich Edit control: RichEdit for version 1.0, RICHEDIT_CLASS for version 2.0, MSFTEDIT_CLASS for version 4.1.

In addition, the Microsoft Windows SDK has provided developers with another solution to more easily define user interfaces: dialog box templates defined in a resource file. It is key to note that while the whole widget creation process when relying on the CreateWindow() function happens inside the C source code file, a dialog box template is defined entirely in a resource file, which has its own format but whose syntax is more flexible than C code (it is declarative). Resource definition statements are basically of the form resource_name type parameters, with type being one of the predefined values such as PUSHBUTTON, MENU and DIALOG, the latter defining a dialog box template [8]. A dialog box template can then be “inflated” simply with one call to the DialogBox() or CreateDialog() functions (macros), depending on whether a modal or modeless dialog box is expected. The prototype of these two dialog functions is as follows: void DialogBox(instance, template, parent, msg_proc), void CreateDialog(instance, template, parent, msg_proc) [9, 10].

The approach followed by Motif can be said to be between the conventional Windows API widget creation method with the CreateWindow() function and the Windows API dialog box template (resource file) approach (e.g. DialogBox() function). Indeed, unlike dialog box templates of the Windows API, Motif widgets of the X Window System (X11) such as push buttons still need to be instantiated by a function call in the source file, e.g. XmCreatePushButton(). Such a call enables to assign a name to the newly created widget, name which is then reused in the resource file to set additional properties for that GUI element: background and foreground colours, font, button label, and so on [11]. A sample function prototype is given: Widget XmCreatePushButton(parent, name, parameters, count).

For reference, two sample resource template files, one for the Windows API and the other for X11’s Motif, are given in Listings 1 and 2, respectively.

Listing 1. Sample resource template file of the Windows API: a simple dialog (i.e. window, a.k.a. form) with a status bar is declared.

```
DLG_MAIN DIALOGEX 0, 0, 300, 200 CAPTION L"Dialog title" STYLE DS_CENTER | WS_OVERLAPPEDWINDOW {
  CONTROL L"Status bar text", IDC_STATUS, STATUSCLASSNAME, SBARS_SIZEGRIP, 0, 0, 0, 0
}
```

Listing 2. Sample resource template file for X11’s Motif: the label and style of the push button pushButton1 of the form myForm is defined.

```
Application*XmPushButton.foreground: white
```

```
Application*XmPushButton.background: blue
Application.myForm.pushButton1.labelString: Proceed
```

Object-oriented programming (OOP) has strongly impacted user interface development, notably with inheritance. Java, .NET and probably others have thus relied on OOP classes to define UI widgets and their relationships. For the programmer, user interfaces can thus be defined by instantiating widgets (i.e. objects) directly into the Java (e.g. with the AWT and Swing libraries), .NET, etc. source code. For instance, in C# (.NET framework) within the System.Windows.Forms namespace, a dialog, called a form, can be instantiated simply with Form f = new Form(), populated with Button b = new Button() and f.Controls.Add(b), and made visible to the user with f.ShowDialog() [12].

Finally, user interfaces can be declared with an XML-based resource file, then inflated and managed within the source code files. Microsoft has introduced the XAML language to this end, which can be combined, for example, with C#. The Windows Presentation Foundation (WPF) UI system and the (short-lived) SilverLight application framework take advantage of XAML [13]. Mozilla XUL is another such UI system based on XML resource files. The UI techniques and best practices described hereinafter are also based on this declarative approach.

3. Modern User Interface Best Practices for Android Applications

Without loss of generality, we consider in this section the Java programming language. The choice of language is not important since it is only an interface towards the underlying software architecture. Hence, the Kotlin language could be similarly considered. A very frequently encountered user interface scenario of a mobile application is considered: view switching, asynchronous data retrieval (e.g., from the Internet) and efficient data visualisation.

3.1. On Activity, Fragment and ViewModel

There are three main components in a modern Android application: the class derived from the Activity class, the classes derived from the Fragment class and the classes derived from the ViewModel class.

The Activity class acts as entry point into the application program: it is used to setup the user interface of the parent view, for example inflating resources from XML resource template files and registering event handlers. It is also in charge of setting up the window chrome: the application main menu, its action bar and so on. Importantly, it is also the place where navigation between child views is configured, typically based once again on an XML resource template file (called a navigation graph). Finally, as exemplified in Section 3.3, global scope operations such as thread setup are conducted therein too.

A Fragment class represents one child view of the application: switching between child views is an important aspect of such mobile applications and it is configured with a navigation graph in the Activity class as just mentioned. So, a Fragment is responsible for setting up the UI of the corresponding child view, notably inflating resources from the corresponding resource template file, registering event handlers and instantiating adapters (see Section 3.2). In addition, a special type of Fragment (PreferenceFragmentCompat) can also be used to easily handle global application preferences (i.e. persistent settings). The skeleton of a typical XML resource template file corresponding to Activity, with window chrome (action bar), and to the hosted child view (Fragment) to which is set a navigation controller is shown in Listing 3.

Listing 3. The skeleton of a typical XML resource template file corresponding to Activity, with window chrome (action bar), and to the hosted child view (Fragment).

```
<!-- activity.xml -->
<?xml version="1.0" encoding="utf-8"?>
```

```
<...CoordinatorLayout ... >
  <...AppBarLayout ... >
```

```
        <...Toolbar ... />
    </...AppBarLayout>
    <include layout="@layout/content" />
</...CoordinatorLayout>

<!-- content.xml -->
<?xml version="1.0" encoding="utf-8"?>
<...ConstraintLayout ... >
    <fragment ...
        android:id="@+id/nav_host_fragment_content"
        android:name="...NavHostFragment"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />
</...ConstraintLayout>
```

A ViewModel class is used to reliably exchange data between two Fragments or between a Fragment and the Activity class. The ViewModel is thus in charge of the actual storage of the data to be communicated (shared), which precisely involves the instantiation and initialisation of the corresponding data container (the data can be a single variable or reference though), and of possible operations on the data, for example element insertion into the data container.

An illustration of the relationship between the Activity, Fragment and ViewModel components and of their respective main duties is given in Fig. 1. Note that the Activity class in this figure actually represents a class that extends Activity (which is itself often replaced by the AppCompatActivity class). The same remark holds for the Fragment and ViewModel classes.

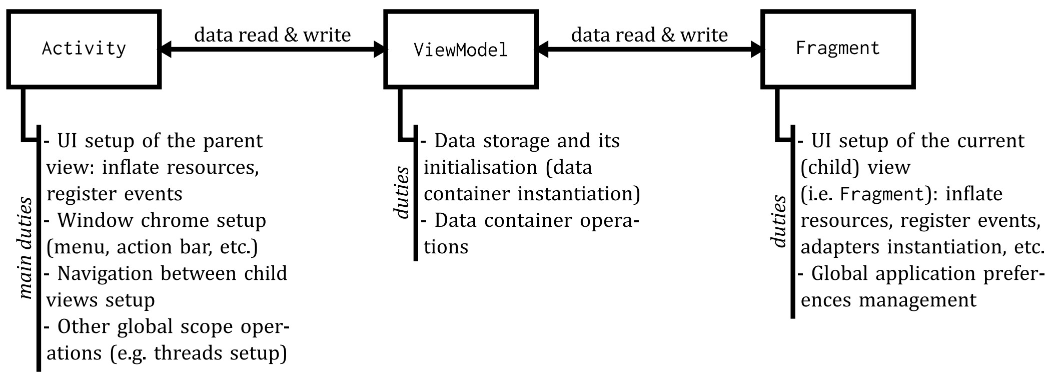
Activity
data read & write
ViewModel
data read & write
Fragment
main duties
- UI setup of the parent view: inflate resources, register events
- Window chrome setup (menu, action bar, etc.)
- Navigation between child views setup
- Other global scope operations (e.g. threads setup)
duties
- Data storage and its initialisation (data container instantiation)
- Data container operations
duties
- UI setup of the current (child) view (i.e. Fragment): inflate resources, register events, adapters instantiation, etc.
- Global application preferences management

Fig. 1. An illustration of the relationship between the Activity, Fragment and ViewModel components and of their respective main duties.

3.2. Efficient Data Visualisation with RecyclerView

The RecyclerView component is recommended to display data as it efficiently “recycles” items to keep the memory footprint of the application as low as possible. Layout settings of the RecyclerView component (e.g. 1-dimensional layout or 2-dimensional, grid-like layout) is typically done when initialising the Fragment hosting it.

Essential to a RecyclerView is the data set to which it is bound. This is the purpose of the Adapter class (precisely, it is typically a class derived from RecyclerView.Adapter<CustomAdapter.ViewHolder>). Once instantiated, a reference to the corresponding data container is stored inside the Adapter and used when the RecyclerView needs to display one item (i.e. *what*) or retrieve the total number of elements in the data set. The Adapter is also in charge of setting *how* an item is to be displayed. These three components could be said to implement the model-view-adapter (MVA) software architectural pattern [14].

Finally, it may be useful to store a reference of the Adapter into a ViewModel to allow other Fragments, or the Activity, to interact with the data set of the RecyclerView component, for example in response to user interaction with the window chrome.

An illustration of the relationship between these components and of their respective sample main duties is given in Fig. 2.

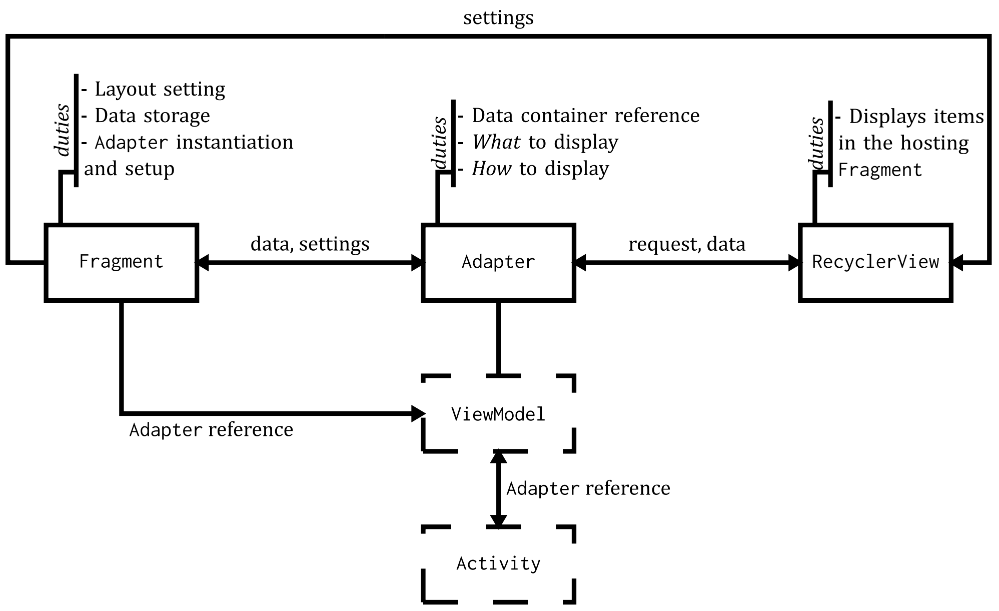

Fig. 2. An illustration of the relationship between the Fragment, Adapter and RecyclerView components and of their respective main duties.

3.3. Responsive User Interface with Multithreading

Asynchronous processing is key to a responsive user interface. In other words, the main thread, which is in charge of UI calls, cannot afford to run lengthy operations as that would freeze the user interface. Therefore, it is a common practice to conduct such lengthy operations concurrently in another thread. Information exchange between the two threads, the main/UI thread and the secondary/worker thread, becomes thus critical. Data retrieval from the Internet is an example of lengthy operation. Even though it is a well-known issue, it is nevertheless interesting to investigate how it can be implemented with modern and recommended techniques as of the time of writing in the case of Android.

Although a thread can be easily created with the Thread class, the HandlerThread class allows to additionally setup a message loop for the thread, which is convenient for controlling the new thread. When the worker thread has finished its execution, it is common to report to the main thread so that the UI is updated as, for instance, new data have become available.

To this end, the worker thread is initialised with a reference to the main thread and, optionally, a callback function: the callback function contains UI code, which thus has to be executed by the main thread, hence the reference to the UI thread passed to the worker thread. In other words, the callback function is called from the worker thread (indirectly: it is posted for execution to the main thread's message loop), however it is run in the main/UI thread. A ViewModel can also be passed to the worker thread to facilitate data exchange between threads.

Sample code run in the main thread, for instance defined in Activity, is given in Listing 4, and sample worker thread code in Listing 5. Data exchange between threads with a ViewModel is briefly exemplified in these listings.

Listing 4. Responsive UI with a worker thread for executing lengthy operations asynchronously – sample code run in the main thread.

```
public class MainActivity extends AppCompatActivity {
  @Override protected void onCreate(Bundle savedInstanceState) {
        // ... (not directly related code)
        // Instantiate worker (secondary thread initialisation)
        Worker worker = new Worker(new Handler(Looper.getMainLooper()), this::onUpdate, viewModel);
        // Get data asynchronously
        worker.requestData();
    }
    public void onUpdate() {/*UI calls*/}//Callback function: called by worker but run in UI thread
}
```

Listing 5. Responsive UI with a worker thread for executing lengthy operations asynchronously – sample code run in the worker thread.

```
public class Worker {
    private final Handler resultHandler;
    private Handler workerHandler;
    public Worker(Handler handler, Runnable callback, ViewModel viewModel) {
        resultHandler = handler;
        initThread(callback, viewModel);
    }
    private void initThread(Runnable resultCallback, ViewModel resultViewModel) {
        HandlerThread handlerThread = new HandlerThread("workerHandlerThread");
        handlerThread.start(); // Start thread
        workerHandler = new Handler(handlerThread.getLooper(),
            new Handler.Callback() {
                @Override public boolean handleMessage(@NonNull Message message) {
                    if(message.what == 1) { // Notify the main thread that new data are ready
                        resultViewModel.sampleMethod(); // E.g. modifies the data held by the ViewModel
                        resultHandler.post(resultCallback); // resultCallback is run in the main thread
                        return true; // The message has been processed
                    }
                    else return false;
                }}); // Setup worker's message loop
    }
    public void requestData() { // Called by the main thread
        workerHandler.post(new Runnable() { // Execute in the worker thread
            @Override public void run() {
                // ... lengthy operation ...
                workerHandler.sendEmptyMessage(1); // Notify worker's message loop of completion
            }});
    }
}
```

Finally, this multithreading approach could be further improved by including the worker thread code directly into the ViewModel: an observer on the data held by the ViewModel is declared inside the main thread (i.e., UI thread), and upon modification of those data the observer is notified. Subsequently, the observer, which is run by the main thread, can update the UI accordingly.

4. Discussion

It is interesting to see that the description of the user interface in a resource file, as done currently with an XML file in the case of Android applications, is far from being a new concept: this technique can be at least traced back to dialog box templates of the native Microsoft Windows API. This earlier development method – still perfectly valid, and employed as of today – is not relying on the XML format though: the XML standard was only published in 1998 [15], and the Windows API for creating dialog boxes based on a dialog box template defined in a resource file is several years older. For instance, sample code demonstrating the usage of the Windows API function DialogBox() and a sample dialog box template are already given in [16] and one year later the DialogBox() function is mentioned in [17].

In addition, a similar technique of declaring UI elements properties in a resource file can also be found early with the Motif toolkit of the X Window System. As explained in Section 2, the Motif approach is however different in that GUI widgets need to be instantiated in the source code file with function calls, assigning to each widget a name. The widget name is then reused in the resource file to declare additional widget properties, such as colours, font and label string [11].

This evolution was not obvious considering that various GUI toolkits and libraries embraced the advent of object-oriented programming, such as Java's Abstract Window Toolkit (AWT) and Swing, dismissing the external UI template resource file approach in favour of source code inline widget (object) instantiation (i.e. OOP). (The author is unaware of any official attempt to provide AWT or Swing with external UI template resource file support.)

Application reliability and robustness have undoubtedly benefited from modern UI development techniques. For instance, garbage collection, data sharing between views via ViewModel, and thread-safe considerations when relying on multithreading like for responsive UI are major advances. Moreover, although event-driven programming has been in use for decades, the modern event management system, for instance with anonymous class instances and lambdas when registering event handlers (callbacks), has contributed to further source code streamlining and thus increasing of the overall program robustness.

Now, there are also drawbacks to this modern UI development approach. For instance, it is easy to get lost in the midst of the numerous classes and available UI widgets. This issue is nothing new though: it became prominent with the rise of (fully) oriented programming, notably with the Java and .NET frameworks. This is to be compared with the few predefined Windows API widget classes and the even fewer related API functions.

Somehow related, the introduction of new classes, and thus new approaches, for UI development inside a same framework is also concerning given that previous solutions often become deprecated, even if still usable (but for how long?). This is the case for example of Android API's AsyncTask designed for safe multithreading with the UI. Consequently, API documentation and code examples also rapidly become outdated since relying on deprecated components.

In addition, virtual machines required to run Java bytecode or .NET managed code can negatively impact performance. (It can be noted that Android replaced its virtual machine (the Dalvik virtual machine [18]) for running applications with the Android Runtime (ART) which translates bytecode to native code.) Besides, automatically generated code and implicit operations (e.g. the instantiation of a ViewModel), that is code used but not directly exposed to the programmer, can rapidly become confusing. Once again, this is not a new issue: refer, for example, to message map macros like ON_COMMAND of the Microsoft Foundation Classes (MFC) [19].

Finally, it has come to our attention that while conventional UI development such as X11 Motif could easily be done directly and entirely by the programmer, this has become increasingly difficult with modern UI development such as techniques described in Section 3. Precisely, a rapid application development environment (RAD) such as Android Studio in our case seems almost compulsory, at least highly recommended given the complexity (in terms of the numerous classes, namespaces, etc.) of both the involved source code (e.g. Java) and UI template resource code (i.e. XML). Furthermore, there is a significant loss of freedom for the developer: for the sake of application robustness, many constraints are, directly or indirectly, forced upon the programmer by the ecosystem. In the case of Android, the application life cycle and the data exchange mechanism between fragments are two examples of such constraints (although usually just called "recommendations"). So, from this point of view, it can be said that the end user is favoured over the programmer.

5. Conclusions

The user interface of software is the cornerstone of human-centred computing and human–computer interaction, two critical issues of our modern, information technology dependent society. In this paper, after having briefly reviewed the evolution of user interface development, we have documented several modern user interface development best practices. The latter has been supported by considering the practical case of Android applications. Precisely, we have considered a very frequently encountered mobile application scenario, involving view switching, asynchronous data retrieval and efficient data visualisation. Importantly, this investigation work has been done in a pedagogical manner so that documented best practices can be easily reproduced within software development projects. Finally, we have also discussed our findings regarding user interface development and its evolution.

Regarding future works, it would be interesting to consider the issue of user interfaces for Far Eastern cultures as discussed, for instance, in our previous paper [20]. In addition, further investigating the impact on hardware resources, notably power consumption, of modern interfaces is another meaningful objective.

Conflict of Interest

The author declares no conflict of interest.

References

[1] Salah, D., Paige, R. F., & Cairns, P. (2014). A systematic literature review for agile development processes and user centred design integration. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (EASE; London, England, 13–14 May)

[2] Barker, C. (2004). Moving from text-based to graphical user interface. *Proceedings of the International Automatic Testing Conference*.

[3] Staggers, N., & Kobus, D. (2000). Comparing response time, errors, and satisfaction between text-based and graphical user interfaces during nursing order tasks. *Journal of the American Medical Informatics Association*, *7(2)*,164–176.

[4] Simpson, T. W., Barron, K., Rothrock, L., Frecker, M., Barton, R. R., & Ligetti, C. (2007). Impact of response delay and training on user performance with text-based and graphical user interfaces for engineering design. *Research in Engineering Design*, *18(2)*, 49–65.

[5] Stephanidis, C., Salvendy, G., Antona, M., Chen, J. Y. C., Dong, J., Duffy, V. G., Fang, X., Fidopiastis, C., Fragomeni, G., Fu, L. P., Guo, Y., Harris, D., Ioannou, A., Jeong, K., Konomi, S., Krömker, H., Kurosu, M., Lewis, J. R., Marcus, A., Meiselwitz, G., Moallem, A., Mori, H., Nah, F. F. H., Ntoa, S., Rau, P. L. P., Schmorrow, D., Siau, K., Streitz, N., Wang, W., Yamamoto, S., Zaphiris, P., & Zhou, J. (2019). Seven HCI grand challenges. *International Journal of Human–Computer Interaction*, *35(14)*, 1229–1269.

[6] Liu, Y., Goncalves, J., Ferreira, D., Xiao, B., Hosio, S., & Kostakos, V. (2014). CHI 1994–2013: Mapping two decades of intellectual progress through co-word analysis. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.

[7] Microsoft Corporation. (2022). CreateWindowA macro (winuser.h). Retrieved from: https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-createwindowa

[8] Microsoft Corporation. (2022). Resource-Definition Statements. Retrieved from: https://docs.microsoft.com/en-us/windows/win32/menurc/resource-definition-statements

[9] Microsoft Corporation. (2022). DialogBoxA macro (winuser.h). Retrieved from:

https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-dialogboxa
[10] Microsoft Corporation. (2022). CreateDialogA macro (winuser.h). Retrieved from: https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-createdialoga
[11] Comrade Corporation. (1994). Motif widgets programming. *ASCII*, Tokyo, Japan.
[12] Sells, C. (2003). *Windows Forms Programming in C#*. Addison-Wesley Professional, Boston, MA, USA.
[13] Nathan, A. (2013). *WPF 4.5 Unleashed*. Sams Publishing, Carmel, IN, USA.
[14] Läufer, K., & Thiruvathukal, G. K. (2018). Managing concurrency in mobile user interfaces with examples in android. In Sushil K. Prasad, Anshul Gupta, Arnold Rosenberg, Alan Sussman, and Charles Weems, editors, *Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms*, 243–285.
[15] World Wide Web Consortium. (1998). Extensible Markup Language (XML) 1.0. REC-xml-19980210.
[16] Hashiguchi, S. (1994). *C/C++ NI YORU WINDOWS Programming Nyūmon*. Softbank Books, Tokyo, Japan.
[17] Pietrek, M. (1995). *Windows 95 System Programming SECRETS*. IDG Books Worldwide, Foster City, CA, USA.
[18] Shoaib, M., Yasin, N., & Abbassi, A. G. (2016). Smart card based protection for Dalvik bytecode — Dynamically loadable component of an Android APK. *International Journal of Computer Theory and Engineering*, *8(2)*,156–160.
[19] Prosise, J. (1999). *Programming Windows with MFC* (2nd ed.). Microsoft Press, Redmond, WA, USA.
[20] Bossard, A. (2021). On bridging the gap between far eastern cultures and the user interface. *Proceedings of the 9th World Conference on Information Systems and Technologies* (WorldCIST; Terceira Island, Azores, Portugal.

Copyright © 2023 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0)