J-DA
JOURNAL
DEKSTOP APPLICATION

Received: 27 Mar 2025 | Revised: 30 Apr 2025 | Accepted: 13 May 2025 | Published: 6 Jun 2025 E-ISSN: 2961-9688, P-ISSN: 2961-9696

DOI: 10.59431/jda.v4i1.659

RESEARCH ARTICLE

D.P.F Delphi AndroidNative Components: A Framework for Native Android Development with Delphi XE5

B. Yaghobi [1*]

[1] SourceForge Community, Iran.

Correspondence

[1*] SourceForge Community, Iran.
Email: b.yaghobi@sourceforge.net

Funding information
SourceForge Community, Iran.

Abstract

The development of Android applications using Delphi XE5 and the D.P.F AndroidNative Components represents a significant step in unifying traditional desktop programming with the demands of modern mobile computing. This research explores how Delphi's object-oriented and component-based design can be extended to the Android environment through native integration using the Java Native Interface (JNI) and Android NDK. The study begins by outlining the technical architecture of Delphi XE5, emphasizing the FireMonkey framework, JNI bridging, and SDK interoperability. It continues with an overview of the D.P.F AndroidNative Components library, which provides Delphi-accessible wrappers for core Android widgets and APIs such as buttons, lists, dialogs, and system services. Through a structured development workflow—covering environment configuration, UI design, event-driven programming, and deployment—the framework enables developers to create fully native Android applications using Object Pascal while maintaining Delphi's familiar development model. Experimental implementation demonstrates that applications built with D.P.F components achieve native performance, efficient memory handling, and seamless interaction with Android system features. Compared with other cross-platform frameworks like Flutter or Xamarin, D.P.F offers a unique balance of native execution and Delphi's rapid application development efficiency. Although challenges remain, including limited maintenance and partial component coverage, the framework proves valuable for developers and educators aiming to extend Delphi's potential into the Android domain. The study concludes that D.P.F AndroidNative Components not only reaffirm Delphi's adaptability in the evolving mobile ecosystem but also highlight the importance of community collaboration, open-source sustainability, and cross-paradigm learning in software development.

Keywords
Delphi XE5; AndroidNative Components; JNI; Native Android Development; Mobile Application Framework.

1 | INTRODUCTION

The emergence of mobile computing during the late 2000s and early 2010s marked a decisive transformation in software development, reshaping how applications are conceived, built, and optimized. Android, as an open-source operating system, became a global leader due to its adaptability, extensive device range, and active developer ecosystem (Google, Android Developer Documentation). This shift required traditional desktop programmers to master new paradigms, including event-driven lifecycles, touch-based interfaces, and platform-specific APIs. Developers accustomed to Delphi, C++, or .NET environments faced a steep learning curve in adopting Java and later Kotlin, as well as tools such as Android Studio (Phillips *et al*., 2013; Mascorro *et al*., 2017). The introduction of the Android Native Development Kit (NDK) offered a path to integrate C and C++ code for computation-intensive operations, providing significant performance improvements but also introducing complexity in managing cross-language interoperability (Son & Lee, 2011; Mascorro *et al*., 2017). In response to these challenges, Embarcadero released Delphi XE5 in 2013, integrating the FireMonkey (FMX) framework to support cross-platform development across Android and iOS from a unified codebase (Embarcadero Technologies, FireMonkey Documentation). While FMX accelerated multi-platform delivery, its abstraction layer limited access to native APIs and controls, resulting in applications that occasionally diverged from platform-specific interaction standards (Cantù, 2013). To address this limitation, the D.P.F AndroidNative Components project (SourceForge, 2013) was developed to bridge Delphi's component-based architecture with Android's native environment.

The framework extends Delphi by wrapping essential Android SDK components—such as user interface controls, notifications, and system APIs—into Delphi-accessible interfaces, allowing developers to create native Android applications without abandoning familiar workflows (Embarcadero Delphi XE5 Documentation; Afonso *et al*., 2016). Unlike cross-platform tools such as Flutter, Xamarin, or React Native, which rely on intermediate rendering layers, D.P.F AndroidNative Components directly interact with native Android controls through the Java Native Interface (JNI) and the NDK, ensuring lower latency and more precise resource management (Huang *et al*., 2014; Ruggia *et al*., 2023). This approach aligns with ongoing research emphasizing the performance and responsiveness benefits of native code integration within Android (Kalkov *et al*., 2012; Afonso *et al*., 2016). Furthermore, the hybrid nature of D.P.F supports a balanced development model that leverages Delphi's rapid application development (RAD) paradigm while providing flexibility for platform-level optimizations, thus reducing redundancy in maintaining separate native codebases for multiple platforms. The project's scope encompasses visual components (buttons, lists, text fields, pickers), system APIs (e.g., Android.Widget, Android.OS, Android.Net, Android.R), and specialized controls such as clocks, chronometers, and alerts. Its design philosophy emphasizes accessibility for Delphi developers seeking to transition into Android development with minimal dependence on Java, while maintaining adherence to Android's design and performance standards. By integrating the NDK and leveraging direct JNI access, D.P.F AndroidNative Components illustrate how established development environments like Delphi can adapt to the evolving demands of mobile computing—combining cross-platform flexibility with the robustness of native performance (Afonso *et al*., 2016; Ruggia *et al*., 2023; Huang *et al*., 2014). Ultimately, this integration represents a step forward in unifying traditional programming methodologies with modern mobile frameworks, reaffirming Delphi's relevance in the landscape of Android-native application engineering.

2 | TECHNICAL ARCHITECTURE

Delphi XE5 represents a significant milestone in extending Delphi's capabilities into mobile application development, primarily through the integration of several interrelated technologies that enable close interaction between Object Pascal and the Android operating system. Central to this evolution is the FireMonkey (FMX) framework, which provides a cross-platform environment for designing responsive and visually dynamic user interfaces with hardware-accelerated rendering and component-based structure. The framework promotes design reusability and accelerates development by applying a consistent programming model familiar to traditional Delphi developers, thereby easing their transition into mobile development (Taghavifard *et al*., 2020; Azadi *et al*., 2020). Within this environment, visual and non-visual components—such as controls, property editors, and event handlers—are structured according to Delphi's long-established component-oriented paradigm. This architectural continuity significantly reduces the learning effort required for developers moving from desktop to mobile application design while maintaining flexibility in UI customization and device adaptability (Vermeulen *et al*., 2022). A critical aspect of Delphi XE5's technical design lies in its ability to interface seamlessly with Java through the Delphi-to-Java bridge, a mechanism that facilitates bidirectional communication between Delphi code and Java classes. This bridge enables Delphi developers to invoke Java methods, instantiate Android classes, and utilize native Android SDK features without directly coding in Java (Taghavifard *et al*., 2020). It also manages type conversions, memory references, and exception handling automatically, ensuring stable interaction between Delphi's managed runtime and Android's Java-based environment. Through this bridge, developers can integrate custom Java

libraries, third-party SDKs, and native APIs directly within Delphi projects, eliminating the need for full proficiency in Java programming while still leveraging the comprehensive Android ecosystem.

The integration of the Android Software Development Kit (SDK) within Delphi further enhances the interoperability between Delphi and Android by enabling access to multiple API levels, automated permission management, and manifest configuration. These features ensure that applications conform to Android's platform requirements while taking advantage of advanced system functionalities such as sensor access, networking, and file handling (Azadi *et al*., 2020). Furthermore, Delphi XE5 provides built-in deployment tools for compiling, packaging, and deploying Android applications directly to physical devices or emulators, thereby offering a complete development pipeline that aligns with professional Android development standards. At the core of this architecture is Delphi's Java Native Interface (JNI) support, which bridges Object Pascal and Java at the native level. JNI integration allows Delphi code to call Java methods, manipulate Java objects, and access Android system services. The framework automatically generates Delphi interface definitions for Java classes, creates wrapper methods with appropriate type mappings—such as converting Delphi strings to Java String, integers to int, and Boolean types correspondingly—and handles memory management between Delphi's manual reference model and Java's garbage-collected environment. Proper synchronization ensures that object references are preserved, preventing premature collection and circular dependencies during runtime. Within this context, the D.P.F AndroidNative Components framework encapsulates these low-level operations behind a simplified Delphi interface, allowing developers to work with Android-native features without needing to manage complex JNI interactions manually. The component architecture of D.P.F AndroidNative Components follows a consistent and modular design philosophy. Each Delphi component inherits from standard base classes such as TComponent or TControl and serves as a wrapper around a corresponding native Android widget. The components expose properties that mirror Android attributes, provide Delphi-style events corresponding to Java listeners, and define methods that invoke native widget operations. Internally, each Java wrapper object maintains a reference to the Android widget, handling initialization, lifecycle management, and cleanup. Property methods employ getter and setter patterns that invoke underlying Java methods, perform necessary type conversions, and trigger event notifications when values change.

Event handling mechanisms register Java listeners and translate Android callback events into Delphi-compatible event procedures, ensuring responsive interaction between the user interface and application logic (Vermeulen *et al*., 2022). This design preserves Delphi's familiar event-driven model while integrating smoothly with Android's activity and view lifecycles. The D.P.F framework itself is organized into modular units, allowing developers to include only the functionality required for specific projects. The core framework module provides foundational classes, interoperability utilities, and memory management routines that support the entire component hierarchy. Specialized modules encapsulate wrappers for Android's native widgets—such as TDPFJButton, TDPFJTextView, TDPFJEditText, TDPFJImageView, and TDPFJListView—as well as dialog and notification components including TDPFJAlertDialog, TDPFJProgressDialog, TDPFJDatePickerDialog, and TDPFJToast. Additional modules include advanced controls like TDPFJAnalogClock, TDPFJChronometer, and TDPFJNumberPicker, alongside wrappers for essential system-level APIs such as Android.Widget, Android.Net, Android.OS, and Android.R. This modular organization improves maintainability, supports incremental extension, and ensures clear separation of concerns between UI, logic, and system interaction. Collectively, these architectural components demonstrate how Delphi XE5 and the D.P.F AndroidNative Components framework enable efficient, native-level Android development within a familiar and highly productive programming environment. The seamless integration of FireMonkey, JNI bridging, SDK access, and component-oriented design provides Delphi developers with both flexibility and control, effectively combining the convenience of cross-platform tools with the performance advantages of native execution.

3 | COMPONENT LIBRARY OVERVIEW

The D.P.F AndroidNative Components library encompasses a broad collection of user interface and system elements designed to streamline native Android application development using Delphi. The core user interface components include TDPFJButton, TDPFJTextView, TDPFJEditText, TDPFJCheckBox, and TDPFJRadioButton. The TDPFJButton component wraps the native Android Button widget, offering properties for text, text color, background, enabled state, and visibility. It supports events such as OnClick, OnLongClick, and OnTouch, along with methods for programmatic clicking, focus management, and layout control. TDPFJTextView provides text display functionality with adjustable properties for content, size, color, font style, alignment, and maximum line count, while supporting events like OnClick and OnLongClick. For text input, TDPFJEditText extends the native EditText widget with configurable properties including hint text, input type (text, numeric, email, password), maximum length, and keyboard behavior. It handles events such as OnTextChanged, OnFocusChanged, and OnEditorAction, supporting features like cursor control and text selection. Meanwhile, TDPFJCheckBox and TDPFJRadioButton enable boolean and mutually exclusive selections respectively, allowing for interactive user options within forms or settings interfaces. In handling images and media, TDPFJImageView serves as a wrapper for the Android ImageView widget. It supports properties for image sources (files, resources, bitmaps), scaling

modes (fitXY, centerCrop, centerInside), tint color, and accessibility descriptions. The component also provides methods for efficient image loading from different sources and includes optimizations for memory management, which is critical for performance in image-heavy applications. It supports user interaction through OnClick and OnLongClick events.

For list-based and selection interfaces, TDPFJListView enables the display of scrollable lists of items by wrapping the Android ListView widget. It supports adapter binding, item selection modes, divider customization, and scroll tracking. Events such as OnItemClick, OnItemLongClick, and OnScroll facilitate interactive list behavior. Though complex in nature, the component simplifies data binding and view recycling while maintaining native performance. Additionally, TDPFJNumberPicker provides intuitive numeric input with properties for minimum and maximum values, current value, wrap selector behavior, and custom display labels. It generates OnValueChanged events when users adjust values, making it ideal for selecting quantities, dates, or times. Date and time functionality is implemented through components such as TDPFJDatePicker, TDPFJTimePicker, TDPFJChronometer, and TDPFJAnalogClock. TDPFJDatePicker manages date selection with adjustable properties for current, minimum, and maximum dates, along with calendar and spinner modes. TDPFJTimePicker allows time selection with 24-hour format options and OnTimeChanged event handling. TDPFJChronometer measures elapsed time with methods for starting, stopping, and resetting the timer, accompanied by OnChronometerTick events for regular updates. TDPFJAnalogClock offers a traditional analog display for time visualization, adding an aesthetic dimension to time-related interfaces. Progress and feedback components include TDPFJProgressBar, TDPFJProgressDialog, and TDPFJToast. TDPFJProgressBar indicates task progression with adjustable properties such as progress value, maximum range, indeterminate mode, and various styles including horizontal bars and circular spinners. TDPFJProgressDialog provides a modal dialog for long-running operations, with properties for title, message, progress state, and cancelation behavior, along with methods for showing, dismissing, and updating progress. TDPFJToast offers non-intrusive notifications for brief user feedback, allowing customization of message text, duration, and screen position.

Dialog management is supported by components such as TDPFJAlertDialog, TDPFJDatePickerDialog, and TDPFJTimePickerDialog. TDPFJAlertDialog supports a range of dialog types, including confirmation prompts, message boxes, and selection lists, with properties for title, message, icons, and button configurations. It can also host custom views, providing flexible interaction models. The TDPFJDatePickerDialog and TDPFJTimePickerDialog components integrate date and time pickers within dialog windows, maintaining consistency with Android's native user experience for date and time input. At the foundation of the visual hierarchy lies the TDPFJView component, which wraps the base Android View class. It defines layout properties such as width, height, margins, and padding, as well as background, visibility, and focus management. It also handles interaction events including OnClick, OnLongClick, OnTouch, and OnFocusChanged. Although not often used directly, TDPFJView serves as a structural and functional base for all other visual components and can be utilized for custom layouts or as a container. Beyond user interface elements, the D.P.F library also includes wrappers for key Android system APIs. The Android.Widget module provides access to widget utilities and layout managers; Android.Net supports network operations such as HTTP requests, URL connections, and connectivity monitoring; Android.OS manages system services like handlers, message queues, bundles, and parcelables; and Android.R grants access to system resources such as drawables, strings, and layouts. These API wrappers extend Delphi's reach into the Android operating environment, enabling integration with native services and system-level capabilities. Overall, the D.P.F AndroidNative Components library provides Delphi developers with a cohesive framework for creating applications that blend Delphi's familiar development environment with the power and flexibility of the Android platform. Through its consistent component model, rich event handling, and deep integration with native APIs, the library offers a practical bridge between Delphi's rapid development paradigm and the performance of native Android applications.

4 | RESULTS AND DISCUSSION

4.1 Discussion

4.1.1 Development Workflow

The development of Android applications using Delphi XE5 and the D.P.F AndroidNative Components follows a systematic workflow that unites Delphi's traditional programming environment with Android's native architecture. The process begins with configuring the development environment. Developers install Delphi XE5 or later with mobile development support, including the Android SDK, NDK, Java Development Kit (JDK), and necessary Android device drivers. After preparing the tools, developers download the D.P.F AndroidNative Components library from *SourceForge* and install it into the Delphi IDE using the menu path: Component → Install Packages → Browse to D.P.F package → Confirm installation. Subsequently, Android SDK paths are configured in Tools → Options → Environment Options → SDK Manager, ensuring that SDK versions and paths are correctly set. Developers then create testing profiles, which can be configured for either physical devices using USB debugging or Android emulators. Verification is achieved by creating a new Delphi mobile project; the presence of D.P.F components in the Tool Palette confirms successful setup. Once the environment is ready, developers can start the

actual application creation process. This involves generating a new Multi-Device Application (Delphi) project and selecting Android as the target platform. The user interface (UI) is designed visually by dragging and dropping D.P.F components—such as buttons, text fields, and labels—onto the form. Each component's properties and events are managed through the Object Inspector, following Delphi's established development model. Event handling is implemented using Delphi's event-driven syntax, such as OnClick or OnTextChanged, enabling responsive interactions within the Android application. Developers then configure Android-specific parameters including permissions, manifest attributes, icons, splash screens, versioning, and target API levels. Finally, the project is built and deployed onto a connected Android device or emulator for testing and debugging using Delphi's integrated debugger and Android's logcat utility.

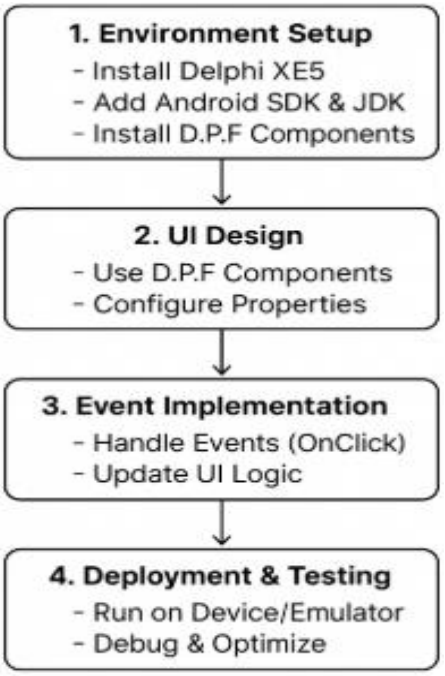

Figure 1. Development Workflow Diagram

This diagram illustrates how the workflow progresses sequentially from environment configuration to deployment and testing, reflecting the integration of Delphi's visual programming model with Android's native system architecture.

4.1.2 Example Implementation and User Interaction

To demonstrate the practical application of this workflow, a simple Delphi–Android project was implemented using D.P.F components. This example showcases a user interface composed of the following elements: TDPFJTextView – displays text labels, TDPFJEditText – captures user input, TDPFJButton – triggers actions on click, TDPFJToast – displays short feedback messages. The following source code illustrates how these components interact to handle user input and provide feedback dynamically. Source Code Example 1: Simple Form with D.P.F Components :

```
procedure TForm1.DPFJButton1Click(Sender: TObject);
var
  InputText: string;
begin
  // Retrieve user input
  InputText := DPFJEditText1.Text;

  // Validate input
  if InputText.Trim.IsEmpty then
  begin
    DPFJToast1.Text := 'Please enter some text';
    DPFJToast1.Show;
    Exit;
  end;

  // Process input
```

```
      DPFJTextView1.Text := 'You entered: ' + InputText;

      // Display confirmation
      DPFJToast1.Text := 'Input processed successfully';
      DPFJToast1.Show;

      // Clear input field
      DPFJEditText1.Text := '';
    end;
```

This example demonstrates how Delphi's event-driven architecture seamlessly translates to Android app behavior. Developers can directly access and modify UI elements using familiar property-based syntax without switching to Java or Kotlin. For more complex applications, components such as TDPFJListView can be used to implement list-based interfaces. This relies on the adapter pattern, which connects data sources (arrays, datasets, or lists) to UI views—allowing dynamic content updates and efficient rendering. Although incomplete in early versions of D.P.F, the structure mirrors standard Android development principles. Dialogs represent another vital interaction pattern in Android applications. The TDPFJAlertDialog component enables developers to create native dialog boxes for user confirmations or alerts while maintaining Delphi's event handling mechanisms. Source Code Example 2: Dialogs and User Interaction :

```
procedure TForm1.ShowConfirmation;
begin
  DPFJAlertDialog1.Title := 'Confirm Action';
  DPFJAlertDialog1.Message := 'Are you sure you want to proceed?';
  DPFJAlertDialog1.PositiveButton := 'Yes';
  DPFJAlertDialog1.NegativeButton := 'No';
  DPFJAlertDialog1.OnPositiveButtonClick := HandleConfirmYes;
  DPFJAlertDialog1.OnNegativeButtonClick := HandleConfirmNo;
  DPFJAlertDialog1.Show;
end;

procedure TForm1.HandleConfirmYes(Sender: TObject);
begin
  // Proceed with confirmed action
  PerformAction;
end;

procedure TForm1.HandleConfirmNo(Sender: TObject);
begin
  // Cancelled by user
  DPFJToast1.Text := 'Action cancelled';
  DPFJToast1.Show;
end;
```

This approach reflects how D.P.F AndroidNative Components preserve Delphi's consistency in code structure, enabling Android developers to build native applications without leaving the familiar Object Pascal ecosystem. By providing direct access to Android's UI and system APIs through Delphi, developers achieve native performance, responsive interaction, and efficient development cycles—bridging two powerful programming paradigms.

4.2 Result

The use of *D.P.F AndroidNative Components* in Delphi mobile development produces several key results and advantages. For Delphi developers, the framework enables the use of a familiar IDE, visual design tools, and component-based methodology while accessing Android's native APIs. It significantly reduces the learning curve since developers can write Android applications using Object Pascal without needing to master Java or Kotlin. Rapid development is supported through visual form design, drag-and-drop interfaces, and fast debugging cycles. Code reuse is enhanced by leveraging existing Delphi libraries and maintaining a single-language codebase. From the application perspective, programs developed with D.P.F components exhibit true native performance, direct Android API access, and a standard user interface consistent with Android's look and feel. The framework ensures efficient memory usage, hardware acceleration, and proper event handling for native touch and gestures. Additionally, it provides developers full access to system services, hardware features, and the latest Android APIs. When compared with alternatives such as Java/Kotlin, FireMonkey, or other cross-platform frameworks like Xamarin, Flutter, and React Native, D.P.F offers a unique balance. It delivers native performance and familiar Delphi development experience, although it may have a smaller community and limited component coverage. Despite these constraints, its open-source nature and extensibility make it suitable for both educational and professional development contexts. In summary, the integration of Delphi XE5 with D.P.F AndroidNative Components

establishes a powerful foundation for building Android applications that combine Delphi's productivity with Android's native capabilities. The workflow—from environment setup, UI design, and event handling to testing—illustrates how Delphi developers can efficiently transition into mobile development while maintaining familiar paradigms and achieving native-level performance.

5 | CONCLUSIONS AND RECOMMENDATION

The **D.P.F Delphi AndroidNative Components** framework represents a significant milestone in the evolution of Delphi as a mobile development platform, providing a seamless bridge between Delphi's traditional programming environment and Android's native ecosystem. By wrapping native Android components within Delphi's familiar Object Pascal syntax, the framework enables developers to build truly native Android applications without abandoning their established tools or workflows. This integration offers key advantages, including access to a comprehensive component library, adherence to Delphi's event-driven model, and direct utilization of Android's native APIs—allowing for both high performance and a reduced learning curve for Delphi developers entering the mobile domain. As an open-source project, D.P.F invites community collaboration and shared development, encouraging enhancements, new component creation, and continuous improvement. Despite challenges such as incomplete component coverage, limited maintenance, and competition from mature cross-platform frameworks, D.P.F remains a valuable resource for developers seeking to extend their Delphi expertise into native Android development. For potential users, the framework is most beneficial for projects requiring native Android functionality, particularly where teams possess strong Delphi backgrounds and prefer rapid visual development through the Delphi IDE. Before full adoption, developers should test the framework through pilot projects, evaluate component adequacy, and plan for long-term maintenance. Contributors are encouraged to participate by improving documentation, addressing bugs, and expanding the component library, while adhering to coding best practices and maintaining backward compatibility. The broader Delphi community plays a vital role in sustaining such initiatives by supporting open-source efforts, sharing knowledge, and fostering collaboration between desktop and mobile development approaches. Ultimately, D.P.F demonstrates the adaptability and enduring relevance of Delphi in the modern development landscape, offering both practical and educational value for those seeking to bridge traditional and contemporary mobile programming paradigms.

ACKNOWLEDGMENTS

Appreciation to b_yaghobi for creating and sharing the D.P.F AndroidNative Components framework, the SourceForge community for hosting and supporting the project, Embarcadero Technologies for Delphi mobile development capabilities, the Android development community for excellent platform and documentation, and all contributors and users who have supported the project.

REFERENCES

Afonso, V., Bianchi, A., Fratantonio, Y., Doupé, A., Polino, M., Geus, P., & Vigna, G. (2016). *Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy.* https://doi.org/10.14722/ndss.2016.23384

Azadi, T., Sadoughi, F., & Khorasani-Zavareh, D. (2020). *Using modified Delphi method to propose and validate the components of a child injury surveillance system for Iran. Chinese Journal of Traumatology, 23*(5), 274–279. https://doi.org/10.1016/j.cjtee.2020.08.007

Embarcadero Technologies. (n.d.). *Embarcadero Delphi XE5 documentation.* Retrieved from https://docwiki.embarcadero.com/

FireMonkey Framework Documentation. (n.d.). *Embarcadero Technologies.*

Google. (n.d.). *Android developer documentation.* Retrieved from https://developer.android.com/

Huang, Y., Rong, C., Wei, J., Pei, X., Cao, J., Jayaraman, P., & Ranjan, R. (2014). *Hybrid polylingual object model: An efficient and seamless integration of Java and native components on the Dalvik virtual machine. The Scientific World Journal, 2014*, 1–13. https://doi.org/10.1155/2014/785434

Java Native Interface Specification. (n.d.). *Oracle Corporation.*

Kalkov, I., Franke, D., Schommer, J., & Kowalewski, S. (2012). *A real-time extension to the Android platform* (pp. 105–114). https://doi.org/10.1145/2388936.2388955

Mascorro, A., Francisco, M., Álvarez, J., & Cruz–Reyes, L. (2017). *Native development kit and software development kit comparison for Android applications. International Journal of Information and Communication Technologies in Education, 6*(3), 5–13. https://doi.org/10.1515/ijicte-2017-0011

Phillips, B., Stewart, C., & Hardy, K. (2013). *Android programming: The Big Nerd Ranch guide.* Big Nerd Ranch.

Ruggia, A., Possemato, A., Dambra, S., Merlo, A., Aonzo, S., & Balzarotti, D. (2023). *The dark side of native code on Android.* https://doi.org/10.36227/techrxiv.21220247.v1

Son, K., & Lee, J. (2011). *The method of Android application speed up by using NDK.* https://doi.org/10.1109/icawst.2011.6163104

SourceForge. (n.d.). *D.P.F Delphi AndroidNative Components project.* Retrieved from http://sourceforge.net/projects/dpfdelphiandroid/

Taghavifard, M., Yousefzadeh, Y., Feizi, K., & Taghva, M. (2020). *Effective components on cash-based intervention to affected people by natural disasters using information and communication technology in Iran. Journal of Rescue and Relief,* 57–66. https://doi.org/10.32592/jorar.2020.12.1.6

Vermeulen, J., Buyl, R., Luyben, A., Fleming, V., & Fobelets, M. (2022). *Defining midwifery autonomy in Belgium: Consensus of a modified Delphi study. Journal of Advanced Nursing, 78*(9), 2849–2860. https://doi.org/10.1111/jan.15209

How to cite this article: Yaghobi, B. (2025). D.P.F Delphi AndroidNative Components: A Framework for Native Android Development with Delphi XE5. *Journal Dekstop Application (JDA)*, *4*(1). https://doi.org/10.59431/jda.v4i1.659.