2025

Production Test Tool

INTERNSHIP REALISATION DOCUMENT

MARCELINA SIWKA

Contents

List of Tables and Figures:

1. Introduction

This document presents the realization of my internship assignment, titled “Production Test Tool,” completed at Tokheim Belgium, part of Dover Fueling Solutions (DFS). It focuses on the development and implementation process of the project and connects directly to the initial project plan that outlined the goals, scope, planning, and risks.

The assignment took place within Team Rocket, a group responsible for the new generation payment terminal development (Crypto NOVA). While DFS operates globally with around 24,000 employees, the office in Turnhout specializes in software systems for payment terminals, with a team of 60–70 people. Specifically in the team rocket there are four developers, and 2 QA testers related to the task.

The primary objective of the assignment was to create a Production Test Tool, an Android-based application. That will be able to verify whether terminal devices, provided by a third-party manufacturer (PAX), are fully functional before further software installation. Since Tokheim is only developing the software for these devices, it is crucial to ensure the hardware is operational before deploying the company's Android Payment App and shipping the devices to use in field. Reliability, product quality, and customer satisfaction are critical business goals for Tokheim, making the testing process a fundamental part of the workflow.

It is important to mention that the tool is aimed towards the manufacturing workers that are responsible for the device testing in Bladel, Tokheim Netherlands. The app is supposed to be as straightforward and easy to use as possible. All the instructions on the screen, needs to be explicitly explained, so there is no doubt what the user needs to do to pass in the specific test. It was taken into consideration while creating the descriptions and all the instructions visible on the screen.

The context for this assignment also stemmed from the limitations of the existing PAX tools: they either allowed only individual unit tests or contained too many unnecessary test steps in their "burn test" feature. Therefore, Tokheim required a new solution. An efficient, user-friendly, and fully automated Production Test Tool tailored specifically to Crypto NOVA’s and manufacturing worker’s needs.

Although the selection of technologies (Android Studio, Kotlin, and Jetpack Compose) was already determined before the internship began, I conducted my own analysis to better understand the available options. This document presents a weighted decision matrix to assess the technical choices and reinforce why they were appropriate.

The realization of the project followed a phased approach: Initialization phase, Implementation Phase, Unit Test Phase and Finalization Phase.

Throughout the internship, agile methodologies (specifically SAFe Agile) were applied, using Jira and bitbucket to organize tasks, track progress, and manage version control. Daily stand-ups and sprint reviews (at the end of each sprint – every 2 weeks) provided structured feedback and guided the project trajectory.

In this document, I will cover:

- An overview of the tools and frameworks used, supported by a weighted decision matrix,
- The technical implementation details: how the tool was built, key functionalities, and the patterns and architecture applied,
- Code examples to illustrate the most important concepts and solutions,
- Reflections on my learning process, especially as someone with a background in Artificial Intelligence rather than application development,
- Final conclusions based on the work completed and the knowledge gained throughout the internship.

This realization document highlights the technical and analytical efforts that led to the creation of the Production Test Tool, contributing directly to Tokheim's manufacturing standard assurance processes and quality of work for direct workers.

2. Analysis

This chapter reviews the research and analysis that was done in preparation for the implementation of the Production Test Tool. Although the main technologies had already been chosen before the start of the project, it was important to investigate other alternatives and better understand the reasoning behind the selected tools.

The analysis focuses on three key areas: programming languages for Android development, user interface design approaches, and integrated development environments (IDEs). Each section compares different available options, highlights their advantages, and disadvantages, and reflects on why the chosen technologies were appropriate. To support this evaluation, a Weighted Ranking Method was used, allowing an objective comparison based on factors such as development efficiency, maintainability, and integration with the project requirements.

2.1. Android Programming Languages

The choice of programming language plays a crucial role in the development process, influencing both the speed and maintainability of the application. In the case of Android development, there are several languages that can be considered, each offering different advantages and challenges. They are listed below.

Although the project used Kotlin as the main language, it was important to explore other options as well, such as Java, C++ (via the Native Development Kit), and some alternative approaches that are explained later in the text. In the following sections, I will present a brief overview of each language and assess its suitability for Android app development.

2.1.1. Java

Java was the official programming language for Android development for an exceptionally long time. Even today, a considerable number of Android applications are still created and maintained using Java, even though many companies are now moving towards Kotlin (Deepak, The rise of Kotlin: Android development in 2025, 2025). One of the reasons Java remains popular is its strong community and the massive amount of online support and resources available, which can be immensely helpful when facing technical challenges.

However, Java as a language brings its own level of complexity. Developers need to be cautious about many aspects, such as handling null exceptions, checked exceptions, and the overall verbosity of the code. Compared to newer languages, Java requires writing a lot of boilerplate code to achieve even relatively simple

functionality. For someone starting out in Android development, this can make the initial learning curve quite steep. Java itself is not a bad language. It's incredibly powerful, but from a modern app development perspective, it might sometimes feel heavy and less efficient, especially when compared to more recent alternatives like Kotlin.

In the context of this project, Java could technically have been used, but considering the desire for more concise, safer, and modern code, it would not have been the most efficient choice.

2.1.2. Kotlin

Kotlin is a relatively "fresh" programming language when compared to older and more established languages. It was introduced in 2017, initially promoted by Google as a secondary, official language for Android development alongside Java. However, in 2019, things changed significantly: Kotlin became the official and preferred language for Android app development, and it has maintained this status ever since.

Since its introduction, Kotlin has been consistently upgraded and improved. It quickly gained popularity among developers, with many large companies migrating their Android projects from Java to Kotlin (like Meta for example) (Luizzi, 2024). The main reason behind this shift is Kotlin’s ability to reduce boilerplate code and simplify complex structures, making development faster and less error prone.

Compared to Java, Kotlin feels much more modern. It was clearly designed with the idea of learning from the mistakes and limitations of older languages. For example, Kotlin reduces the risk of null pointer exceptions by introducing safe handling mechanisms. There is no longer a need to end every line with a semicolon, which might seem like a small thing, but when developing large applications, such small improvements make the experience cleaner and more efficient.

In general, Kotlin can be considered a language that strives to make development as compact, safe, and intuitive as possible. It fits very naturally into modern Android development workflows, especially when combined with tools like Jetpack Compose. Given these advantages, Kotlin was not just a reasonable choice for this project. It was the obvious one.

2.1.3. C++ and the Native Development Kit (NDK)

C++ is another language that can be used in Android development, although it is important to note that you cannot build a fully functioning Android application using only C++. Instead, C++ is used through something called the Android Native Development Kit (NDK), which allows developers to integrate native C++ code into their Android apps.

The NDK is particularly useful when working with hardware-level operations or when reusing existing C/C++ libraries. In the context of my project, the use of the NDK was essential. The PAX libraries, responsible for interacting with key hardware components such as the magstripe reader, chipcard reader, and barcode scanner, were all provided in native code. Without the NDK, it would not have been possible to connect these functionalities properly to the Kotlin-based application.

Although working with C++ adds an extra layer of complexity to the project, such as managing memory manually or handling more low-level details, it was necessary to bridge the gap between the Kotlin application and the native hardware functionalities provided by the PAX terminal. This experience also gave me valuable insight into how Android apps sometimes need to combine higher-level and lower-level code to deliver complete functionality.

2.1.4. Other Options

While Java, Kotlin, and C++ are the most common languages for Android app development, there are also other alternatives worth mentioning. One of them is C#, which shares many similarities with Java in terms of structure and syntax. In fact, programming in C# often results in cleaner and simpler code compared to Java, which could make it a strong candidate for Android development.

However, there is one important limitation: C# is heavily tied to the Windows ecosystem because of its dependence on the .NET framework. Originally, C# applications could only run on Windows systems, which was a major barrier for Android development. This limitation was later addressed through the creation of Xamarin, a cross-platform framework that allows developers to write native Android (and iOS) applications using C# and share a large part of the codebase across different platforms.

Apart from C#, there are several other possibilities for building Android apps:

- Python (through frameworks like Kivy),
- HTML, CSS, JavaScript combinations (e.g., via Adobe PhoneGap or Apache Cordova),
- Dart (used with Flutter, a popular toolkit for cross-platform app development),
- Corona SDK (a lightweight framework for building apps and games).

However, these technologies are more suitable for cross-platform or rapid prototyping purposes, rather than building hardware-integrated applications like the Production Test Tool. Given the specific requirements of this project (tight integration with native device hardware and a need for maximum stability) they were not considered viable options.

2.1.5. Weighted Decision Matrix: Programming Languages

Before presenting the Weighted Decision Matrix, it is important to briefly explain the categories that were used to evaluate the programming languages. They were evaluated based on the following aspects:

- Development efficiency → How quickly and easily an application can be developed using the given language, considering factors like code readability, amount of boilerplate, and overall simplicity.
- Community Support → How strong and active the developer community is for a given language, which impacts how easily one can find support, examples, libraries, and updates.
- Integration with Android → How naturally and efficiently the language collaborates with Android development tools and native Android features.
- Future maintainability → How well the code written in the language is likely to hold up over time, in terms of readability, scalability, and the ease of introducing updates or improvements.

Which language is the best for the Production Test Tool?

Decision Factors	Weight	Java	Kotlin	C++
Efficiency	**4**	**3**	**5**	**2**
Community Support	**1**	**5**	**4**	**3**
Integration with Android	**3**	**4**	**5**	**3**
Future Maintainability	**2**	**3**	**5**	**2**
	Total:	**35**	**49**	**24**

Table 1: Weighted Decision Matrix of programming language choices

The Weighted Decision Matrix clearly shows that Kotlin is the best fit for the development of the Production Test Tool. It scored the highest across the most important categories, especially in development efficiency, integration with Android, and future maintainability. Java remains a strong option, due to its extensive community support, but its heavier syntax and higher complexity made it less ideal for this project. C++ was necessary for specific native functionalities but would not be suitable as the main language for a full Android application.
This analysis confirmed that using Kotlin as the primary development language was the right decision for this project. Additionally: The application created for the main Crypto Nova android payment platform is also being developed in Kotlin, and after conducting the analysis, I can fully see the reasoning behind it.

2.2. UI Design Approaches

When it comes to building the User Interface (UI) in Android applications, there are two main approaches available: using traditional XML layouts or following the newer

Jetpack Compose trajectory. Both methods have their own advantages and disadvantages, and it is important to understand their differences to make an informed choice for any project.

DFS had already selected Kotlin as the main programming language, the focus is specifically on how UI can be created using Kotlin. It is worth noting that XML layouts have traditionally been used with both Java and Kotlin, while Jetpack Compose is a framework built natively for Kotlin and works most efficiently in that environment.

In this section, I will discuss both approaches, highlighting their key features, strengths, and potential drawbacks. I will start by introducing the XML-based method, which has been the standard for many years, before moving on to Jetpack Compose, the more modern and declarative alternative.

2.2.1. XML Layouts

XML was the original and dominant way of creating views in Android development from the very beginning. The idea behind XML layouts is simple: use a markup language to define the structure and present UI elements within different layouts. This method has been referred to as the "layout way" of building Android applications.

For a long time, XML was the industry standard for UI creation, and it is still widely used today. Many companies continue to rely on XML layouts, even though more projects are starting to migrate toward Jetpack Compose. (Asoyan, 2025)

However, there are several drawbacks associated with the XML approach. First, XML layouts tend to generate a lot of boilerplate code. Developers need to define specific structures to handle UI components and their attributes, which can lead to bloated and harder-to-maintain codebases. As the UI becomes more complex, the XML files themselves can become lengthy and difficult to manage, which increases the risk of errors and slows down development.

Another crucial point is that XML separates the UI definition from the business logic, meaning developers often need to "connect" views manually to the Kotlin (or Java) code through mechanisms like findViewById(). This can make the code less intuitive and introduce additional layers of complexity, especially in larger projects.

2.2.2. Jetpack Compose

Jetpack Compose is an innovative approach to building user interfaces for Android applications. Compose was first introduced by Google in 2019 as a preview version, and by 2021 it was officially released and ready for full production use (Bellini, 2001). Compose is a fully declarative UI toolkit, designed to create user interfaces in a more intuitive and efficient way compared to the traditional XML method.

One of the main advantages of Jetpack Compose is its declarative syntax. Like Swift UI for Apple devices, Compose allows developers to define how the UI should look based on the current state, and the framework itself takes care of updating the UI when that state changes. This removes the need for manual UI updates and reduces the chance of introducing UI-related bugs, making applications easier to maintain and test.

Another key benefit is that while both XML and Compose can achieve the same functional results, Compose allows for much more concise and readable code. It streamlines the development process, making it easier to understand, maintain, and extend applications over time.

Additionally, Jetpack Compose is natively written in Kotlin, which makes it a perfect match for modern Android development. Developers can fully leverage Kotlin's features inside the UI code, leading to even cleaner and more powerful implementations.

Today, Jetpack Compose is heavily promoted by Google as the modern and stable solution for Android UI development (Google, brak daty). Its close integration with Android Studio and natural fit with Kotlin makes it a very comfortable and developer-friendly choice for new projects.

2.2.3. Weighted Decision Matrix: UI Design Approaches

Before presenting the Weighted Decision Matrix, it is important to explain the categories that were used to compare the two UI approaches:

- Development efficiency → How quickly and easily a user interface can be built using the given approach, considering the amount of boilerplate and the complexity of structuring the views.
- Readability and Maintainability → How clear and understandable the UI code remains over time and how easy it is to make changes and updates.
- Integration with Kotlin → How naturally the UI approach works with Kotlin language features and how well it fits into a modern Kotlin approach.

Decision Factors	Weight	XML Layouts	Jetpack Compose
Development efficiency	3	3	5
Readability and Maintainability	2	3	5
Integration with Kotlin	1	2	5
	Total:	17	30

Table 2: Weighted Decision Matrix of UI design approaches

The matrix results show that Jetpack Compose outperforms XML layouts across all important categories. The fact that Compose is fully integrated with Kotlin, combined with its clear and concise code structure, makes it a much better fit for modern Android development. The ability to build UIs faster and maintain them more easily is a huge advantage, especially in projects where time and code clarity matter.

Choosing Jetpack Compose for this project allowed for a more efficient development process and ensured that the application would be easier to maintain and extend in the future.

2.3. IDE Comparison

Choosing the right development environment is another key aspect of building an Android application. A well-matched IDE can make the entire coding process smoother, more efficient, and less error prone. Since Kotlin has already been selected as the main programming language, this comparison will focus on IDEs that offer strong support for Kotlin-based Android development.

In this section, I will take a closer look at how different IDEs perform when working with Android apps. What functionalities they offer, how they can assist during development, and what limitations they might have. The tools I will focus on are Android Studio, IntelliJ IDEA, and Visual Studio Code. I will also briefly mention a few lesser-known alternatives. It's interesting to explore what each environment brings to the table and how they differ in terms of usability, features, and developer experience.

2.3.1. Android Studio

Android Studio is the official Integrated Development Environment (IDE) for Android app development, created and maintained by Google. The entire tool is focused specifically on building Android applications, which makes it very practical and efficient for projects like this one.

One of the biggest advantages of Android Studio is that it comes with a wide range of built-in tools tailored for Android development. It includes an integrated Android emulator, a visual layout editor, and other features like code completion, live previews, and advanced debugging options. All these helps speed up development and reduce the chance of errors.

Another important aspect is that Android Studio is built on top of IntelliJ IDEA, but with customizations that serve the Android ecosystem more directly. This means that it combines the stability and power of IntelliJ with tools designed specifically for mobile development.

What also makes Android Studio especially appealing is that it is completely free, with no paid tiers or limitations. It continues to receive regular updates throughout the year, trying to accommodate the best user experience. The large and active developer community also means that support, tutorials, and solutions are widely available, which is incredibly useful when working under time pressure or dealing with specific problems.

2.3.2. IntelliJ IDEA

IntelliJ IDEA comes in two separate versions: a free Community Edition and a paid Ultimate Edition. While the free version is already very capable, the full potential of the IDE is unlocked with the paid version, which includes advanced features and tools that many consider essential for large-scale or enterprise-level development. Because of this, IntelliJ IDEA is often seen as a great option for companies that have the resources to invest in premium tooling and want a more extensive development environment.

One of the key differences between IntelliJ IDEA and Android Studio lies in its versatility. IntelliJ is a general-purpose IDE, meaning it can be used to develop not only Android apps but also web applications, desktop tools, backend systems, and more. This wide scope can be seen as both an advantage and a drawback, depending on what you need. While it provides flexibility for developers working on multiple platforms, it's not as tightly focused on Android development as Android Studio is.

Because Android support is just one of many features within IntelliJ IDEA, it might not always feel as specialized or streamlined as Android Studio, especially for someone working solely on mobile apps. However, for developers or teams working across different types of applications, the broader functionality of IntelliJ might be a valuable benefit.

2.3.3. Visual Studio Code

Visual Studio Code (VSC) is a very flexible and widely used IDE that supports a broad range of programming languages and technologies. It has gained popularity for its lightweight interface, fast performance, and the ability to customize the development environment using an extensive library of plugins. For many developers, it is the go-to tool for web development, scripting, and even backend services.

Although it is technically possible to develop Android applications in VSC, it is no longer a widely recommended option. Compared to specialized environments like Android Studio or IntelliJ IDEA, Visual Studio Code lacks built-in support for Android-specific features. Developers often need to install multiple extensions, configure external emulators, or rely on physical devices using ADB to test their applications.

These extra steps make the setup more complex and less efficient, especially when compared to the out-of-the-box solutions offered by Android Studio.

Over time, as Android Studio became the default tool for Kotlin development and IntelliJ remained strong for Java, the need for Android development in VSC started to fade. While VSC is still a powerful and capable editor, it no longer holds a significant place in the Android ecosystem, and many developers have gradually moved away from it in this context.

2.3.4. Other IDEs

Besides the main options discussed above, there are several other IDEs that can technically be used for Android development. However, most of them are either very lightweight, platform-dependent, or require additional setup to function as a complete development environment. In many cases, they are not fully standalone, meaning that you might need to configure external tools or integrate SDKs manually to get them working properly.

These IDEs tend to be more niche or specialized, and their use often depends on personal preferences rather than technical superiority. Examples include tools like NetBeans IDE, Xamarin Android, or AIDE (Android IDE). While they can certainly work for some developers, especially those with specific use cases or working environments (as for example: developing cross-platform apps using C# and .NET). They are rarely used in professional Android development workflows today.

It is also important to clarify that the focus of this analysis was on native Android development. This means solutions that are stable, officially supported, and aligned with industry standards. For that reason, the less popular or alternative IDEs were not considered in the final evaluation.

2.3.5. Weighted Decision Matrix:

Before presenting the matrix, it's useful to explain the criteria that were used to evaluate each IDE. These categories were selected based on what I found most relevant when working on a native Android project with Kotlin:

- Android-specific features → Focuses on how well the IDE supports Android development out of the box (tools like emulators, layout editors)
- Kotlin Integration → How naturally and fully the IDE supports Kotlin (syntax, debugging and project structure)
- Development efficiency → How IDE helps speed up development (code suggestions, build tools, debugging support)
- Community and Support → How active the developer community is, how easy to find help online, and how often the IDE gets updates or new features.

Decision Factor	Weight	Android Studio	IntelliJ IDEA	Visual Studio Code

Android-specific features	4	5	4	2
Kotlin Integration	3	5	5	3
Development efficiency	2	5	4	3
Community and support	1	5	4	4
	Total:	50	43	27

Table 3: Weighted Decision Matrix of IDEs for Android Development

The results of the matrix confirm that Android Studio is the most suitable environment for this project. Its full focus on Android development, excellent Kotlin support, and built-in tools such as the emulator and layout editor make it the most complete solution out of all options. IntelliJ IDEA scored slightly lower, mostly because Android-specific features are not its primary focus. Visual Studio Code, while flexible and widely used in other contexts, lacks many built-in Android features (Emulator integration, Layout Editor) and requires significant additional setup to be usable for this kind of project.

2.4. Summary of Analysis

The goal of this chapter was to better understand the tools and technologies used in the development of the Production Test Tool. Even though the tech stack had already been defined when the internship began, I conducted an independent analysis to validate the choices and explore alternatives. This helped me gain insight into why each technology was chosen and whether it truly fit the specific requirements of this project.

The evaluation covered three main areas: programming languages, UI design approaches, and integrated development environments. Each section included a comparative analysis supported by a Weighted Decision Matrix, which made it possible to assess the options objectively based on criteria like efficiency, integration, maintainability, and usability.

The final conclusions confirmed that the selected stack is indeed well-suited for this project:

- Kotlin was the strongest candidate for the programming language due to its modern syntax, concise structure, and native integration with Android.

- Jetpack Compose outperformed XML layouts in terms of flexibility, code readability, and compatibility with Kotlin, making it the best option for building the user interface.

- Android Studio, as a purpose-built IDE for Android development, proved to be the most complete and practical environment. With built-in tools, excellent Kotlin support, and strong community backing.

Together, these technologies form a reliable and efficient stack for developing the Production Test Tool. The analysis not only justified the original choices but also strengthened my own understanding of Android development best practices.

3. Technical Implementation

This chapter describes the technical implementation of the Production Test Tool. After completing the analysis and defining the appropriate tools, this part focuses on how the solution was built. From the initial setup, through architectural choices, to specific features and testing logic.

Since the application was created entirely from scratch, every part of the solution had to be designed, built, and integrated manually. The goal was to deliver a fully functional tool that could test a wide range of hardware components on PAX terminals in a fast, reliable, and structured way. The application had to be easy to use for manufacturing workers, but also maintainable and scalable from a technical point of view.

In this chapter, I will walk through the most important aspects of the implementation: How the project was set up using Gradle, how navigation and state handling were implemented using Jetpack Compose, how specific UI components were built and made reusable, and how each test works. Both from a technical and user-facing perspective.

Throughout the process, my focus was on creating a solution that is clean and intuitive. Tailored specifically to the exact needs of the requirements. Where possible, I opted for reusable logic, shared states, and clear test flow control to ensure consistency for the user.

3.1. Project Setup and Gradle Configuration

As part of the initial setup, I used Gradle, the standard build system for Android, to define the application modules and include all necessary dependencies. Some of the most important libraries I added were:

- Jetpack Compose, used for the entire UI structure,
- Hilt, for dependency injection and cleaner architecture,
- PAX SDK, required for accessing hardware components like the chipcard reader, barcode scanner, and printer,
- Other AndroidX and Kotlin-related libraries for lifecycle management, navigation, and UI testing.

I also configured NDK integration to work with native libraries delivered by PAX. This required ABI filtering and proper linking of .so files to ensure that features like magstripe reading and contactless support functioned correctly on the device.

Since the entire setup was done manually, I gained a much better understanding of how Gradle works behind the scenes - from dependency resolution to build variants.

This early phase was a critical foundation for everything that followed in the development process.

3.2. Jetpack Compose Navigation

After setting up the project structure and adding the required dependencies, the next crucial step was handling navigation between different parts of the application.

In Android development, navigation plays a central role. It forms the foundation of user interaction in most modern applications. With proper navigation logic, it becomes possible to move between screens smoothly, while maintaining clarity in the code and consistency in the user experience.

In this section, I will explain how navigation is implemented using Jetpack Compose, what the core concepts behind it are, and how they were applied specifically in the Production Test Tool.

3.2.1. Traditional vs safe-type navigation

As the section title suggests, the navigation in this application was implemented using Jetpack Compose. One of the biggest changes brought by Compose is the ability to define navigation paths directly in Kotlin code, instead of relying on XML files or string-based route definitions. Since Jetpack Compose is tightly integrated with the Kotlin language, it enables a more flexible and declarative way of controlling screen transitions.
In the Compose navigation model, there are three essential components:

- **NavHost**: the container that holds all composable destinations and listens for navigation changes. It represents the main structure of the navigation graph.
- **Composable screens**: every screen is defined as a @Composable function that combines layout and logic in one place.
- **Navigation Controller**: the element responsible for handling navigation actions like moving to the next screen or passing arguments.

When I first started implementing navigation in the project, I used the traditional Compose approach. The one based on manually defined string route names. This method was also used in the reference project (Crypto NOVA), so it felt like a natural place to start. The idea was simple: every screen was assigned a route in the form of a string, and the navigation controller would move between them by referring to those strings.
However, during one of the code reviews, an alternative was introduced: type-safe navigation. At the time, it wasn't part of the official requirement, but it was mentioned as something worth exploring. After doing some research, I decided to fully refactor the navigation layer and implement it in the type-safe way.
The concept is based on using sealed classes and serializable types to define the possible navigation destinations. Instead of passing plain strings to identify screens, each destination is defined as an object. This has two major advantages:

1. The compiler verifies the navigation logic at build time, reducing the risk of runtime errors caused by typos or incorrect route names.
2. The code becomes more readable and structured, since all destinations are clearly defined in one place, and route arguments can be passed as typed objects.

The actual implementation used a sealed class called Destination, where each screen is represented as a data object. The NavHost then uses these types directly to register composable destinations. Thanks to that, navigation became not only safer, but also easier to follow and manage in a growing project.

```
sealed class Destination {
    @Serializable
    data object MainScreen: Destination()
    @Serializable
    data object TouchScreen: Destination()
```

Figure 1: Destination class

```
@Composable
fun Navigation(
    navController: NavHostController,
    appNavigator : IAppNavigator,
    modifier: Modifier = Modifier){
    val viewModel: TestFlowViewModel = viewModel()
    NavHost(
        navController = navController,
        startDestination = Destination.UnitTestScreen,
        modifier = modifier
    ) {
        composable<Destination.MainScreen>{
            MainScreen(appNavigator)
        }

        composable<Destination.TouchScreen> {
            TouchScreenTest(viewModel, appNavigator)
        }
```

Figure 2: Navigation component responsible for navigating between the screens

3.2.2. The role of AppNavigator and IAppNavigator

The main flow of the Production Test Tool is linear and predefined. Each terminal must go through a strict sequence of hardware tests, starting with the touchscreen test and ending with the camera test. Since this order is fixed, the navigation logic needed to reflect that. Not just by showing the screens in a certain order, but by enforcing it within the application architecture itself.
According to best practices in Android development, it is generally recommended to avoid exposing the NavController directly inside composable screens (Developers). Instead, navigation logic should be abstracted and centralized. That is where the idea of AppNavigator came in.
The AppNavigator is a wrapper around navigation logic. Its purpose is to manage the full test sequence without exposing the underlying navigation implementation to the UI layer. To achieve this, a list of Destination objects was created to represent each test screen in the correct order. The navigator keeps track of the current index and exposes methods like goToNext(), reset(), or goToPage(...) to control the flow. When goToNext() is called, it sends the next screen to be displayed through a Channel<Destination>.
To make navigation injectable and lifecycle safe, I also defined an interface called IAppNavigator and provided it via Hilt using a @Module. This allowed me to inject the same navigator instance throughout the application, while keeping the implementation details hidden.
This approach made the navigation layer:

- **Type-safe**: every screen is a sealed object, not a fragile string,
- **Reusable**: navigation logic is shared across all tests,
- **Decoupled**: composables don't need to know anything about NavController.

By managing screen flow from a single source of truth, the app remained clean and easy to extend, even when the number of test screens grew.

```
interface IAppNavigator {
    val navigationChannel: Channel<Destination>
     fun goToNext()
     fun reset()
     fun goToPage(page: Destination = Destination.UnitTestScreen)
}
```

Figure 3: interface responsible for managing paths

```kotlin
@Module
@InstallIn(SingletonComponent::class)
object NavigatorHiltModule {
    @Provides
    @Singleton
    fun provideAppNavigator() : IAppNavigator = AppNavigator()
}
```

Figure 4: Singleton for providing AppNavigator across the application

```kotlin
class AppNavigator @Inject constructor(
) : IAppNavigator {
    // Order of the tests
    private val destinations=listOf(
        Destination.TouchScreen ,
        Destination.DisplayScreen ,
        Destination.LEDScreen ,
        Destination.BeepScreen ,
        Destination.MagStripeReaderScreen ,
        Destination.ChipCardReaderScreen ,
        Destination.ContactlessReaderScreen ,
        Destination.PrinterScreen,
        Destination.BarcodeScannerScreen ,
        Destination.CameraScreen ,
        Destination.TestOverviewScreen

    )

    private var currentIndex=0
    override val navigationChannel=Channel<Destination>(Channel.BUFFERED)

    override fun goToNext() {
        if (currentIndex < destinations.lastIndex + 1) {
            navigationChannel.trySend(destinations[currentIndex])
            currentIndex++
        }
    }
```

Figure 5: Implementation of IappNavigator interface

3.3. Time Management in App

Time-based logic plays a crucial role throughout the Production Test Tool. In many parts of the application, actions are tied to timers: whether it's showing a screen for a fixed amount of time, limiting the duration of a test, or managing automatic transitions between steps.

Most tests have a strict time limit, ensuring that they don't run indefinitely or block the test flow. In some cases, the timer is clearly visible to the user. For example, when counting down the time left to complete an action. In other situations, the timer runs silently in the background, and the user is not aware of the exact time constraints.

This section explores how different types of timers were implemented in the app, how they are used across screens, and why different approaches were needed depending on the context.

3.3.1. Two Timer Types

The first one was designed to be used inside screens; for example, to count down the time remaining for the user to complete a task. Since these timers are part of the UI, they were created as composable functions. One limitation with composables is that they can't be suspended. In other words, they cannot include blocking or asynchronous operations like you would with standard coroutines.

That's why the LaunchedEffect block was used to create a reactive, countdown-style timer. It updates the UI every second and calls a callback when the time is up. Depending on the parameters passed, this timer can be either shown to the user or

run silently in the background.

```
@Composable
fun CountdownTimer(
    timeGiven: Int,
    isVisible: Boolean,
    onFinished: () -> Unit
) {
    var timeLeft by remember { mutableIntStateOf(timeGiven) }


    // Basic countdown timer
    LaunchedEffect(key1 = timeLeft) {
        while (timeLeft>0){
            delay(1000L)
            timeLeft--
        }
        // When time is finished
        onFinished()
    }

    if(isVisible) {
        Text(text = stringResource("Time left: %1$s" , timeLeft))
    }
}
```

Figure 6: Timer related to the composables (countdown visible for the users)

The second timer is not related to UI at all. It's a pure logic-based timeout function, used when the program needs to wait for something to happen within a specific period. A good example is when the app is expecting a card to be inserted or a barcode to be scanned. If nothing happens before the deadline, the test should fail and move on. This timer is implemented as a suspend function, which means it can run asynchronously in a coroutine without blocking the main thread. It checks the condition every few milliseconds (based on the configured interval), and if the condition is met before the deadline, it returns true. Otherwise, it waits until the timeout ends and returns false.

```kotlin
suspend fun timerWithTimeout(
    timeoutMillis: Long,
    checkInterval: Long = 100L,
    condition: suspend () -> Boolean
): Boolean {
    val deadline = System.currentTimeMillis() + timeoutMillis
    while (System.currentTimeMillis() < deadline){
        if(condition()) {
            return true
            break
        }
        delay(checkInterval)

    }
    return false
}
```

Figure 7: Logic timer (internal, not visible to the users)

Using two distinct timers helped me separate UI logic from control logic, making the code cleaner and easier to maintain. It also prevented the app from freezing or becoming unresponsive during tests, which is essential for a tool that must work reliably in a fast-paced production environment.

3.3.2. Extension Functions for Epoch Time

Kotlin, as a programming language, offers several interesting features (such as Smart Casting, Null safety) that I found both powerful and practical. Especially coming from an artificial intelligence background, where most of my previous projects were written in Python. One of the Kotlin features I really appreciated during this project was extension functions.

An extension function allows you to add new functionality to an existing class without modifying its source code or using inheritance. This is particularly useful when working with types or libraries that you cannot directly change. Such as platform-level classes or third-party APIs.

In my case, I used an extension function to convert epoch time (represented as Long) into a readable “mm:ss” format. Before that, the values being passed around were just raw numbers representing milliseconds since 1970, which may not be easily readable for everyone. By creating this extension on Long, I could call

.epochToReadableDate() on any long value, and it would instantly give me a formatted result.

```
fun Long.epochToReadableDate() : String {
    val instant = Instant.ofEpochMilli(this)
    val dateTime = LocalDateTime.ofInstant(instant , ZoneId.systemDefault())
    val formatter = DateTimeFormatter.ofPattern("mm:ss")
    return dateTime.format(formatter)
}
```

Figure 8: Extension functions that change epoch time into readable format

This made the code much cleaner and easier to reuse. Instead of having to repeat formatting logic every time I wanted to show a duration or timestamp, I now had a single utility function that could be used throughout the app, as if it were part of Kotlin's standard library.

3.4. Test Execution Flow and State Handling

Even though each test in the application checks something different, from a chipcard reader to a printer or a touchscreen, the general flow behind them is always the same. The test starts, the user performs an action (or not), and based on that, the test either passes or fails. Because this logic is repeated across all test types, it made sense to create a shared, reusable structure that could support the same behavior throughout the entire testing sequence.

The idea was to build a system that is consistent, user-friendly, and easy to maintain. Instead of writing a completely new flow for each individual test, I designed a common structure for test screens and a shared mechanism for tracking test states and outcomes. This way, new tests can be added quickly, and existing ones remain easy to understand and manage.

The upcoming section will explain how the main test sequence is controlled, how test states are tracked and updated, and what decisions were made to keep the flow intuitive for the user and logical from the development side.

3.4.1. The Test Sequence

In the standard test flow of the application, each device must go through a fixed sequence of steps. The order of the tests is always the same, regardless of how many times the application is used or restarted. The goal is to make sure that every essential component of the terminal is checked in a consistent and reliable way. The sequence is as follows:

1. Touchscreen test

2. Screen test
3. LED test
4. Beeping test
5. Magstripe reader test
6. Chipcard reader test
7. Contactless reader test
8. Printer test
9. Barcode scanner test
10. Camera test

Each of these tests checks a different part of the device and requires a different type of user interaction. From simple screen taps to inserting a card or scanning a barcode. Still, they all follow the same logical structure.

3.4.2. Shared Test States

Although each test in the Production Test Tool validates something different, they all follow the same basic flow. Every test begins with an explanation screen, then enters an active state where the actual functionality is tested, and finally ends with either a success or failure. These three phases, preparation, execution, and result, repeat across every test.

To keep this consistent and reusable, I introduced a shared TestUiState structure that includes three key properties:

- testId: uniquely identifying the current test,
- currentStep: representing the current stage (not started, in progress, passed, or failed),
- testPassed: a Boolean that records the outcome.

The test steps themselves are defined as a sealed class called TestStep. This allows for clearly defined and type-safe states that are shared across all test screens: NotStarted, InProgress, Passed, and Failed.

By using the same TestScreen composable structure for all tests and controlling the UI via a shared state, the user experience stays consistent. Regardless of what specific component is being tested. When the test begins, the state changes to InProgress. Once it ends, it is updated to either Passed or Failed. This logic is applied automatically through a shared ViewModel, which tracks the current test and maintains statistics such as start time and number of attempts.

The biggest advantage of this approach is that only the content of the test needs to change. Everything else, from buttons to transitions, is controlled by the same code. This makes the system easier to extend and reduces the chance of errors when new tests are added.

```
data class TestUiState(
    var testId: String = "",
    var currentStep: TestStep = TestStep.NotStarted,
    var testPassed: Boolean? = null
)

sealed class TestStep {
    data object NotStarted : TestStep()
    data object InProgress : TestStep()
    data object Passed : TestStep()
    data object Failed : TestStep()
}
```

Figure 9: Definition of tests and possible states related to them

```
@Composable
fun TestScreen(
    title : String ,
    description : String ,
    uiState : TestUiState ,
    onStart : () -> Unit ,
    onRetry : () -> Unit ,
    onNext : () -> Unit ,
    onExit : () -> Unit,
    content : @Composable (() -> Unit),

){
    var testFinished by remember { mutableStateOf<Boolean>(false) }
    when(uiState.currentStep){
        is TestStep.NotStarted -> (
                MainScaffold {
                    Column(
```

Figure 10: Skeleton for every test screen

```
@HiltViewModel
class TestFlowViewModel @Inject constructor(
    private val printer: PrinterObject
) : ViewModel()  {
    var uiState by mutableStateOf(TestUiState())
        private set
    private var _testStatistics = mutableMapOf<String, TestStatistics>()
    private var currentTestId: String = ""

    fun startTest(
        testId: String,
        testName: String
    ){
        currentTestId = testId
        if(!_testStatistics.contains(testId)){
            _testStatistics[testId] = TestStatistics(testName = testName)
        }
        // +1 to attempts
        _testStatistics[testId]?.attempts = (_testStatistics[testId]?.attempts ?: 0) + 1
        // current time
        _testStatistics[testId]?.startTime = System.currentTimeMillis()
        uiState = uiState.copy(
            testId = testId,
            currentStep = TestStep.InProgress,
            testPassed = null)

    }
```

Figure 11: Implementation of a shared flow between the tests

```
@OptIn(ExperimentalCoroutinesApi::class)
@Composable
fun BarcodeScannerTest(viewModel : TestFlowViewModel, appNavigator : IAppNavigator){
    var uiState = viewModel.uiState
    LaunchedEffect(Unit) {
        viewModel.resetTest()
    }
    TestScreen(
        title = stringResource("Barcode scanner test"),
        description =stringResource("After clicking the START button please scan the barcode..."),
        uiState = uiState,
        onStart = {viewModel.startTest(
            testId = "bar_code_scanner_test",
            testName = "Barcode Scanner test"
        )},
        onRetry = {viewModel.startTest(
            testId = "bar_code_scanner_test",
            testName = "Barcode Scanner test"
        )},
        onNext = {appNavigator.goToNext()},
        onExit = {appNavigator.goToTestOverview()}
    ) {
```

Figure 12: Exemplary way of implementing the screen

3.5. Screens and UI Design

As mentioned earlier, the tests in this application all follow the same logical structure. So, it made sense for them to also share a consistent visual structure. Keeping the UI uniform not only improves the user experience, but also prevents confusion and helps ensure that the flow feels natural and predictable.

To support this, a set of base UI components was created and reused across all test screens. The idea was to separate the layout and interaction logic from the specific functionality of each test. Buttons, headers, timers, and state-based transitions all behave the same way across the entire application.

The only part that differs from one test to another is the actual content shown during the “In Progress” phase. Since each test checks something different, the screen needs to reflect what is being asked from the user or what hardware response is expected. Outside of that, the structure remains the same, which made the application easier to build, maintain, and scale.

3.5.1. Test Screens

Each test in the Production Test Tool is built on the same screen structure, reusing common layout components and state handling. This was a conscious design

decision, since all tests follow the same logical flow, as described earlier: starting, executing, and ending with either a passed or failed state.

The application begins and ends at the Main Screen. This is the only screen that appears outside of the test flow. From here, the user can either start the full production test sequence (as defined in the assignment) or access individual unit tests. Once the full test sequence is completed, the user is brought back to this screen. It also contains the only Exit button that allows leaving the application entirely.

Figure 13: Main Screen of the application

The test screens themselves are unified in both behavior and layout. When a test is started, it enters the In Progress state and shows the content specific to that test. For example, interaction with the touchscreen or scanning a barcode. Once the test is completed, the result is displayed using a consistent layout.
If the test passes, a message appears briefly before the app automatically moves to the next test.

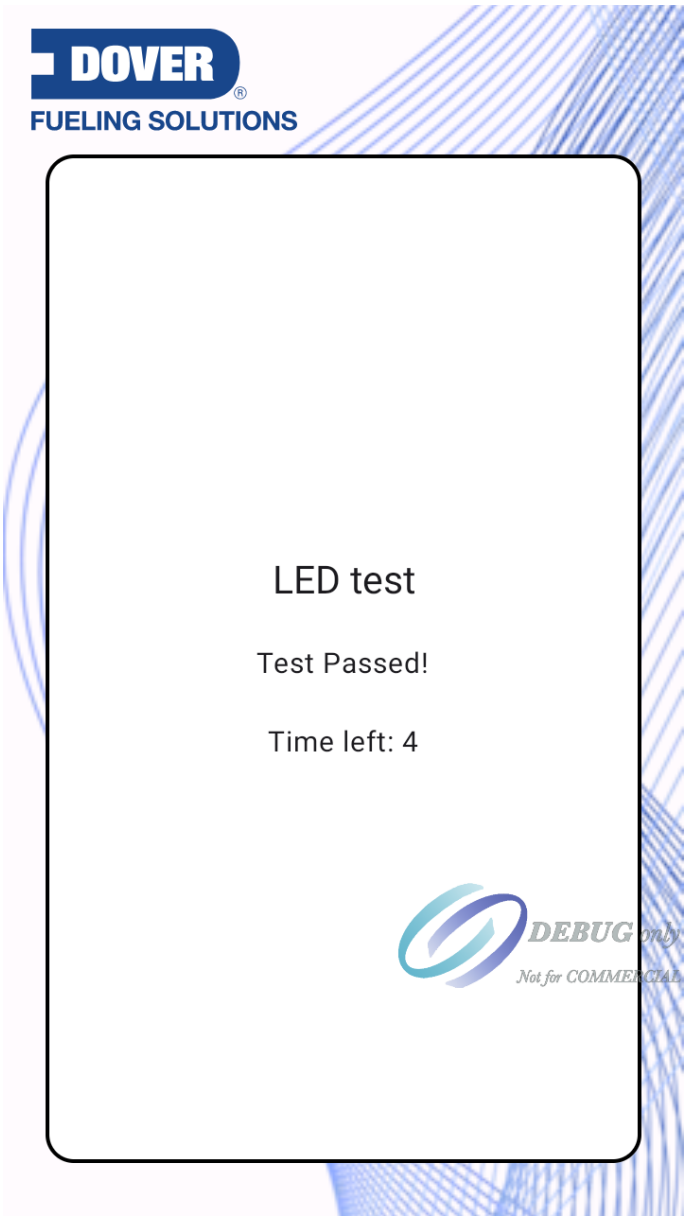

Figure 14: Exemplary transitions between the tests

If the test fails, the screen shows a clear failure message along with buttons for Retry and Exit.

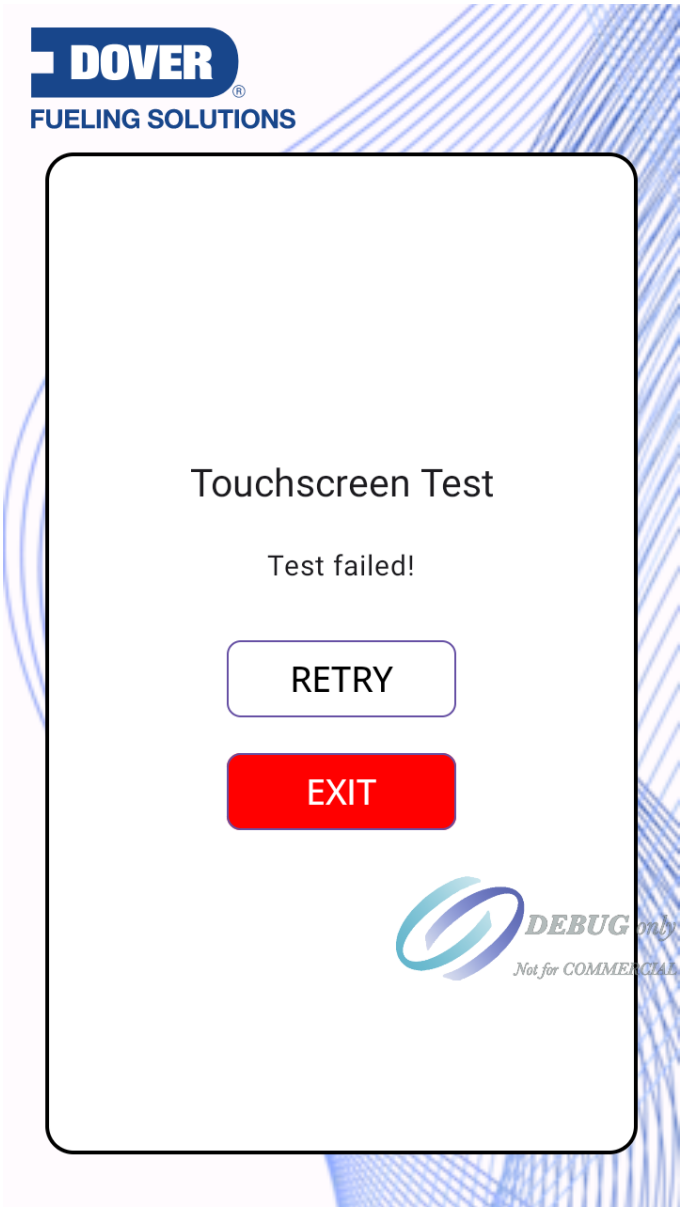

Figure 15: Exemplary screen of the test failing

By using the same base screen and visual language for all tests, the application remains consistent and predictable for users. The only thing that changes is the central content. Everything else is managed and rendered by shared logic. This approach helps reduce development time, avoids duplication, and ensures a smooth experience for anyone using the tool.

3.5.2. Universal Components

To ensure consistency and avoid duplicating layout code, I created a set of universal UI components that are reused throughout the entire application. Since all test screens follow the same structure and logic, it made sense to standardize their appearance and behavior as much as possible, while keeping them flexible.

The main visual structure of the app is defined using a custom **Scaffold**, which acts as the base design frame for all screens. This includes the background pattern, the company logo at the top, and a white centered content box where test-specific logic is displayed. Every screen in the application is built on top of this base.

To support interactions across different screens, I also created a **Universal Button** component. It ensures a consistent “look and feel” for all buttons while still allowing customization when needed. The component accepts parameters like color, font size, or width, so it can adapt to different use cases without changing the core layout or styling rules.

In some tests, the user is required to confirm whether a feature is working correctly. For example, after seeing a screen or hearing a sound. For those cases, I introduced a **Universal Dialog** component. It simplifies yes/no confirmations by letting me reuse a single dialog layout across multiple test types. This makes the code cleaner and easier to maintain, as there is no need to recreate the same dialog logic each time.

To make implementation even smoother, I also designed a **Test Base**: a wrapper that contains the main Scaffold and automatically aligns all content within the white content box. It acts like a pre-styled container. So, when a new “In Progress” test screen is built, it is already centered and styled properly, making it easier to focus on the actual test logic.

This modular approach to UI helped reduce code duplication, improved maintainability, and made it easier to apply design changes across the application in a consistent way.

3.6. Descriptions of The Tests

The tests are the most important part of the entire Production Test Tool. They are the core purpose of the application - its main reason for existing. The idea behind the tool is simple: to make sure that each device is working correctly before it is approved for further use. Without the tests, the rest of the application would have no practical value.

In this section, I will go through each individual test included in the standard sequence. For each one, I will briefly explain:

- what the test is checking,
- how it works from the user’s perspective,
- and how it was technically implemented in the application.

This breakdown shows how different hardware components were managed and how the same test structure was adapted to fit different requirements.

3.6.1. Touch Screen Test

The main goal of this test is to verify whether the touchscreen of the device is functioning correctly. It is the very first step in the test sequence, and its successful completion is required to continue.

When the test starts, the user is presented with a grid of boxes displayed on the screen. These boxes can be tapped or swiped. Every time a box is triggered, its color changes to green. The test is considered passed only when all boxes have been activated. This means that the user has interacted with the entire surface area of the screen.

There is a red exit button, which allows you to leave during the duration of the test if you don’t want to continue the flow. Once every other green box turns green, the normally red button also becomes a part of the “normal” grid. Clicking the last button allows the user to proceed to the next test in the sequence: the screen test.

However, if the user does not complete the interaction within 60 seconds, the test automatically fails. In that case, a failure message is shown along with two options: retry the test or exit the Production Test Tool.

The logic behind this test ensures that the touchscreen is responsive across the entire area and that no regions are unresponsive or blocked. It’s also one of the most interactive tests in the sequence, requiring direct user engagement and physical contact with the device (either by tapping or swapping).

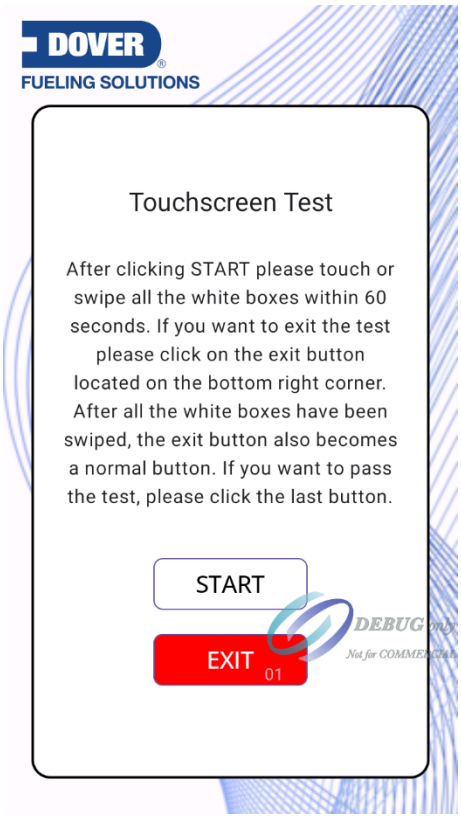

Figure 16: Touchscreen test description

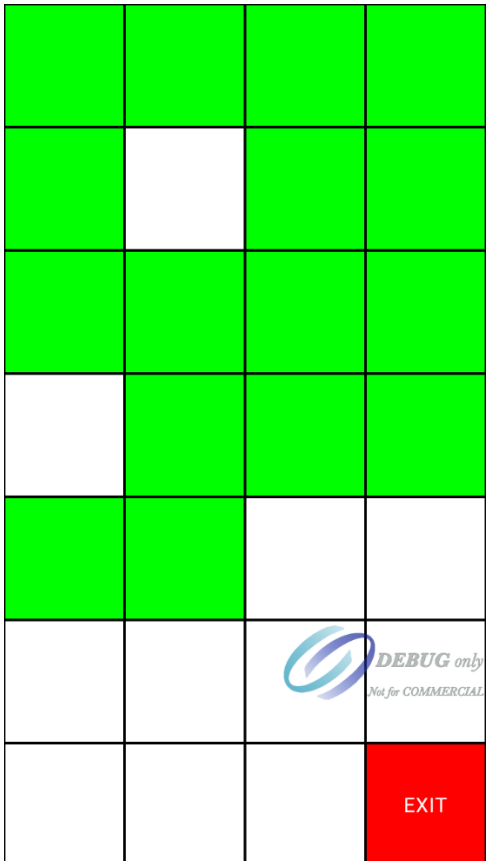

Figure 17: Touchscreen test in progress

3.6.2. Screen Test

The purpose of this test is to check whether the display of the device is functioning correctly, including image clarity, color accuracy, and visibility.

When the test starts, the user sees a predefined picture displayed on the screen for 10 seconds. During this time, they are expected to assess whether everything looks correct: if the colors are accurate, the image is sharp, and there are no visible issues with the screen itself.

After the image disappears, the user is asked to confirm whether everything appeared as expected.

- If the user clicks "Yes", the test is considered passed, and the application automatically proceeds to the LED test.
- If the user selects "No", the test fails. In this case, the user is given two options: retry the test or exit the Production Test Tool entirely.

This test relies on subjective human verification. However, it still is crucial to ensure that the display is fully operational and free from visual defects.

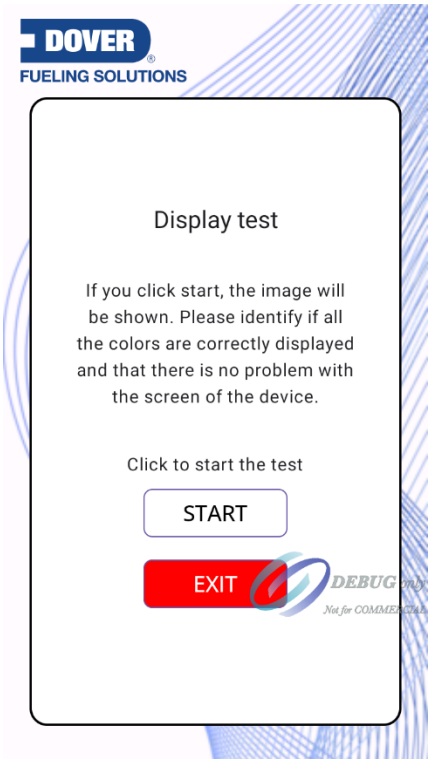

Figure 18: Display test description

Figure 19: Display screen in progress

Figure 20: Display screen user confirmation dialog

3.6.3. LED Test

The test verifies whether all LED indicators on the device are functioning correctly. It checks if each individual light source activates as expected and is clearly visible to the user.

When the test begins, each LED on the terminal (including the magstripe light, chip light, barcode scanner light and breathing) is triggered one by one. Each LED cycles through five different colors, allowing the user to observe whether all colors and positions are displayed correctly.
After the sequence completes, the user is asked whether all lights were visible and correctly lit.

- Clicking "Yes" confirms that the test has passed, and the application automatically proceeds to the Beep Test.
- Clicking "No" fails the test, giving the user the option to retry or exit the Production Test Tool.

This test relies on visual confirmation but follows the same pass/fail logic as the other screens. It ensures that every light source on the terminal is fully operational and ready for use in real-world conditions.

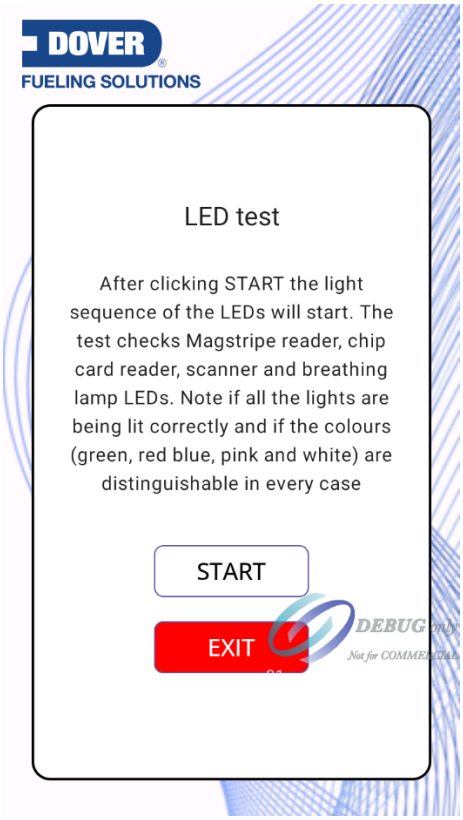

Figure 21: LED test description

Figure 22: LED test exemplary in progress screen

Figure 23: LED screen dialog confirmation

3.6.4. Beeping Test

The purpose of this test is to verify whether the speakers in the PAX device are working correctly. The test ensures that the device can produce sound at different frequencies and that the audio output is clear and recognizable to the user.

When the test begins, the user clicks the "Start" button. The device then plays a simple melody composed of beeps with different hertz frequencies (C5, D5 and E5), allowing the user to evaluate both the clarity and variation of the sound. Once the melody finishes, the user is asked to confirm whether they heard everything.

- Clicking "Yes" means the speaker test has passed, and the application automatically proceeds to the Mag stripe Card Reader Test.
- Clicking "No" fails the test and gives the user the option to retry or stop the Production Test Tool.

This test depends entirely on auditory confirmation, but it follows the same interface and logic structure as the other tests. That makes it intuitive and consistent for the user.

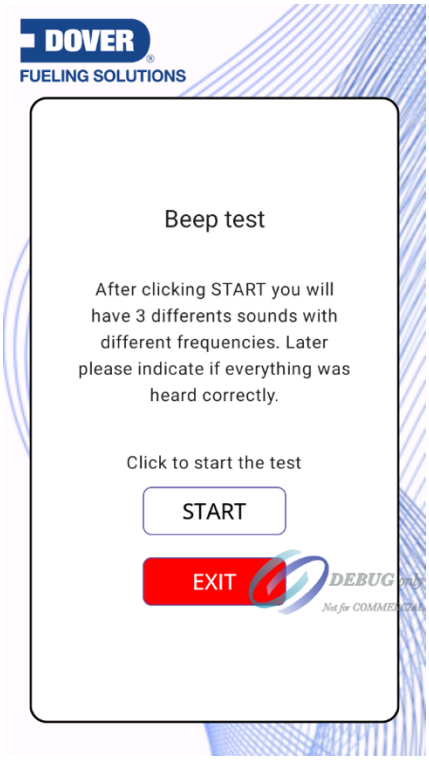

Figure 24: Beep Test description

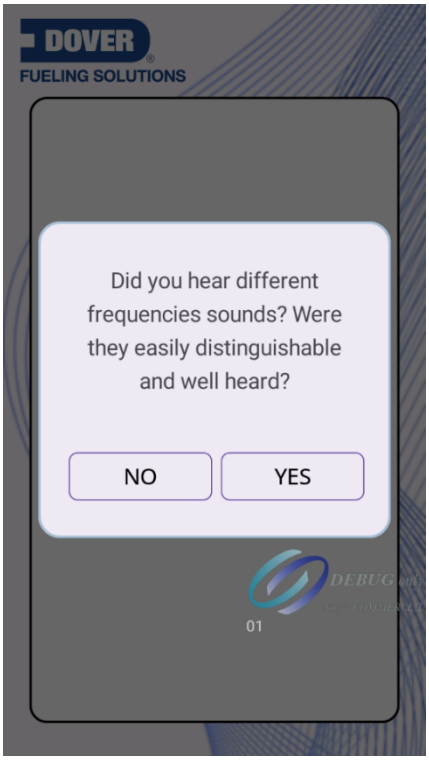

Figure 25: Beep test confirmation dialog

3.6.5. Magstripe Reader Test

The goal of this test is to verify whether the magstripe reader on the PAX terminal can correctly detect and read the data encoded on a magnetic stripe card. In particular, the test checks for both the mechanical status of the card (via switches) and the validity of the three magnetic tracks.

When the test begins, the user is asked to swipe a magstripe card through the reader slot. The application checks the state of the card present, and its tracks, to see whether they can be successfully read.

- If the swipe is valid and all expected data is present (tracks 1-2-3), the test is marked as passed. The user sees a short confirmation, and the application automatically proceeds to the Chip Card Reader Test after 10 seconds.
- If any part of the reading fails, such as a missing track, misaligned swipe, or another status error, a message is shown with appropriate feedback that's easy to understand. In this case, the user can retry the test or choose to stop the entire Production Test Tool.

This test helps ensure that the magstripe reader responds correctly to physical card input and that all required hardware signals and data lines are functioning as expected.

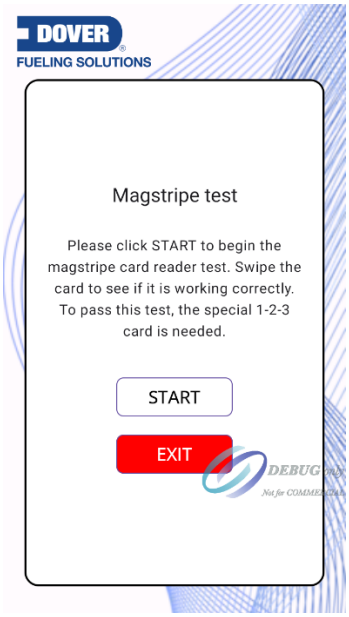

Figure 26: Magstripe test description

Figure 27: Magstripe test in progress

Figure 28: Magstripe test wrong card read

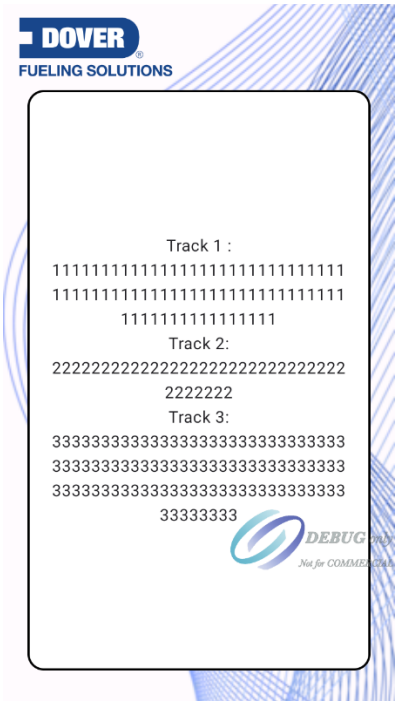

Figure 29: Magstripe Test flow of correct card swiped

3.6.6. Chip Card Reader Test

This test verifies whether the chipcard reader of the PAX device functions correctly. The test ensures that the reader can detect and interpret the data stored on the chip of a smart card.

To begin, the user inserts the chip card into the reader slot and presses the "Start" button. The sequence is valid regardless of whether the card is inserted before or after the button is clicked. Both scenarios are handled identically by the application. Once the card is detected and read successfully, an ATR (Answer to Reset) response is retrieved and displayed on the screen for five seconds. This visual confirmation proves that the card has been properly recognized and processed by the device.

- If the card is read successfully, the test is marked as passed, and the application automatically moves on to the Contactless Reader Test.
- If the reader fails to detect or interpret the card, the test fails, and the user is offered the option to retry or stop the Production Test Tool.

This test ensures that the chip reader responds reliably to real-world interactions, without requiring perfect user timing or card insertion order.

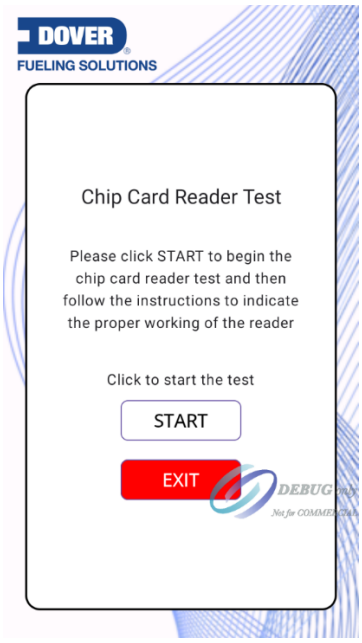

Figure 30: Chip test description

Figure 31: Chip test instructions

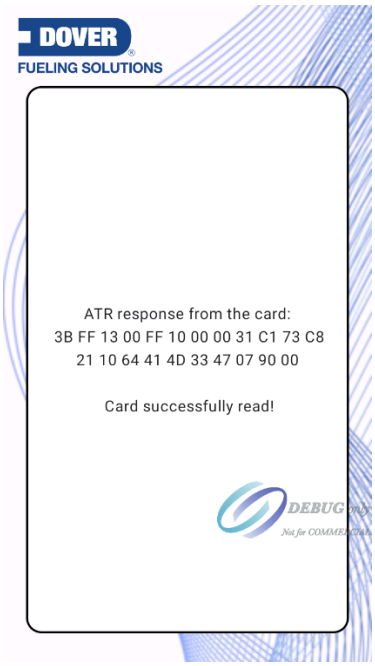

Figure 32: Chip test feedback after reading the card

3.6.7. Contactless Reader Test

This test verifies whether the contactless card reader of the device is functioning properly. The goal is to ensure that a card can be detected and read correctly without physical insertion.

When the test starts, the user is prompted to tap a contactless chip card against the terminal. The reader is present at the top of the device, so the user needs to tap it against the contactless connection icon, which was placed to help the user see where the reader is located (there is no real indicator for that on the actual device). The application listens for card input and validates the received data.

- If the card is detected and the data could be read and then displayed, the test is considered as passed. Basic information from the card (card type, serial number and “other info”) is briefly displayed on the screen, and the application automatically proceeds to the Printer Test after 10 seconds.
- If the data is invalid, incomplete, or not received, the test fails, and an error message is shown. In that case, the user can either retry the test or exit the Production Test Tool.

This test helps verify that the NFC/contactless module of the device is operational and can handle standard card interactions reliably.

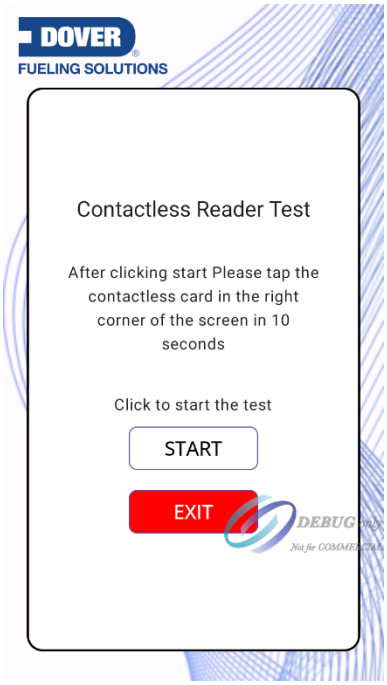

Figure 33: Contactless Card Reader Test

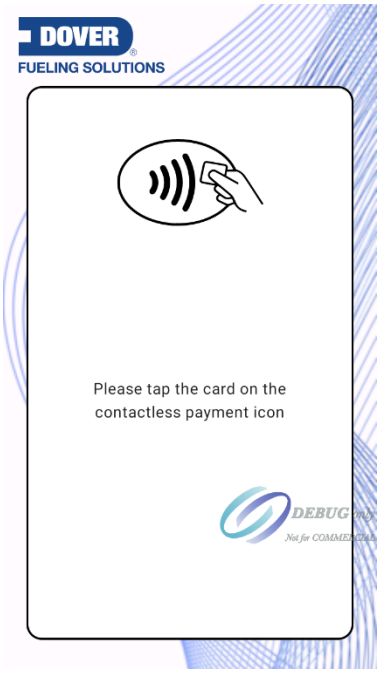

Figure 34: Contactless test in progress

Figure 35: Contactless card flow of no tapping the card in time

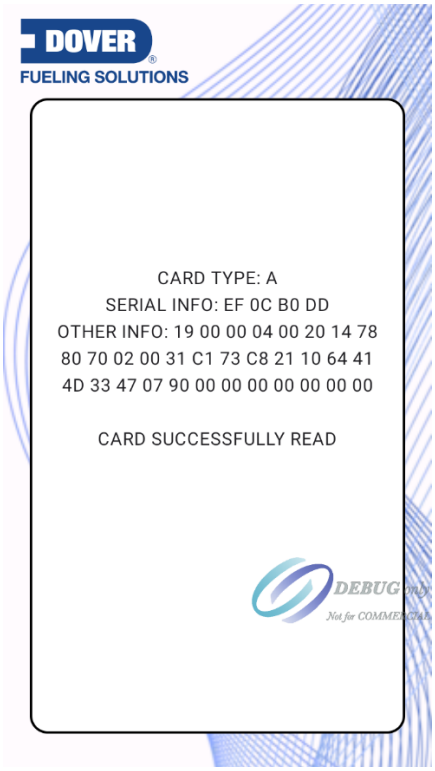

Figure 36: Contactless Test the card is read correctly

3.6.8. Printer Test

The purpose of this test is to check the printer connectivity and basic printing functionality of the external device. It verifies whether the printer, connected via USB cable, can successfully print a simple ticket containing key information. Such as test results and device data.

When the start button is pressed and the test begins, the device attempts to print the ticket automatically. If the process completes without errors and the printer responds correctly, the user is then asked to confirm whether everything was printed properly.

- If the user selects "Yes", the test is marked as passed, and the application automatically proceeds to the Barcode Scanner Test.
- If the user selects "No", the test fails, and the user is given the option to retry, check the printer connection, or exit the Production Test Tool.

This test ensures that the external printer is connected correctly and operational, and that printed outputs are successfully generated as part of the device's final functionality. So, the main idea of this test it to assess the correct functioning of the USB port proper functionality. Because if you think about it, Production test Tool is a crypto Nova testing application. So, if you think about it, why would you need the correct functioning of the printer? It's also a nice-to-have feature, but printers before getting used on the sites are also heavily tested, to make sure the quality is in place.

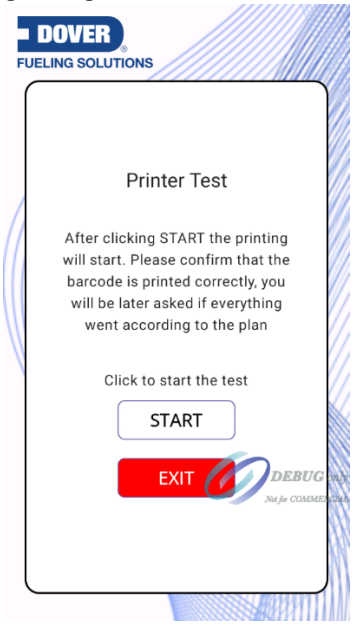

Figure 37: Printer test description

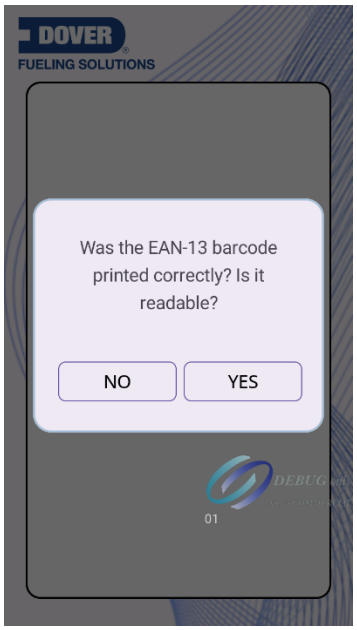

Figure 38: Printer test confirmation dialog

3.6.9. Barcode Scanner Test

The purpose of this test is to confirm that the barcode scanner on the device is functioning correctly. For this test, the application is using the rear camera, which have the possibility to turn on the flashlight, just next to the camera. This way, it would be possible to scan the barcode, even during the night. It verifies whether the scanner can detect, read, and correctly interpret a presented barcode within a limited time frame.

When the test starts, the user is instructed to place a barcode within the scanner's beam. The system waits up to 30 seconds for a valid scan. During this time, the scanner actively monitors for barcode data input.

- If a valid barcode is read successfully and the data matches (so the specific barcode in the list of barcodes passing the test is read), the test is considered passed, and the application automatically continues to the Camera Test.
- If the timeout is reached without a successful scan, or if the scanned data is invalid or does not match the expected format, the test is failed. In both cases, the user will be given the option to retry or exit the Production Test Tool, and the proper feedback is given to the user. So, they know what went wrong or what else was expected.

This test ensures that the device is capable of handling barcode-based operations reliably and within an acceptable response time with a flashlight.

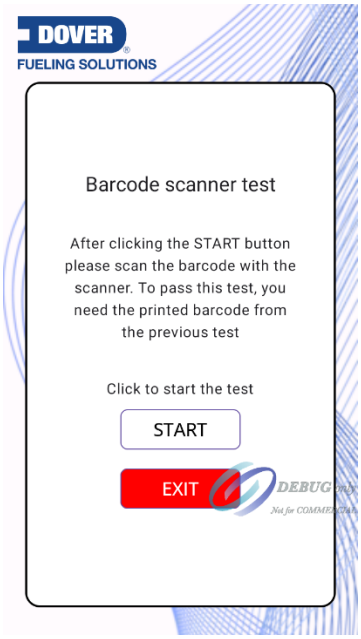

Figure 39: Barcode scanner test description

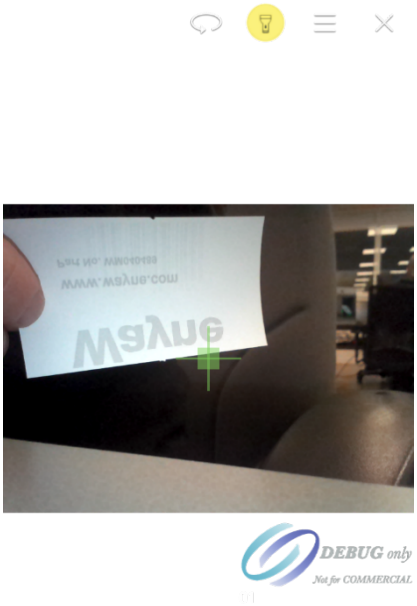

Figure 40: Barcode scanner test in progress

Figure 41: Barcode not scanned

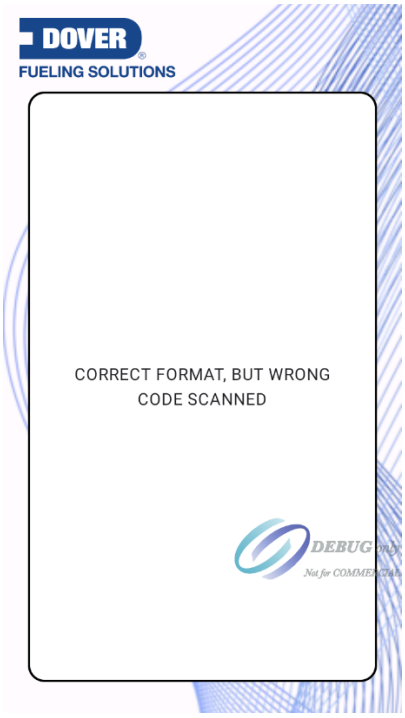

Figure 42: Barcode test - correct format, but wrong content

Figure 43: Barcode test: incorrect format scanned

Figure 44: Barcode scanner: Correct reading of the barcode

3.6.10. Camera Test

The goal of this test is to verify whether the camera module of the device is functioning correctly. The test checks whether the live camera feed is visible, stable,

and displays the expected image quality. This test uses the front camera, where a user can be clearly seen in the frame.

When the test begins, the user sees a real-time preview from the device's camera. After 10 seconds, a dialog appears where they are asked to confirm whether the feed looks correct; meaning that the image is sharp, properly lit, and not distorted in any way.

- If the user clicks "Yes", confirming that the camera is working as expected, the test is marked as passed, and the application automatically proceeds to the final test overview screen after 5 seconds.
- If the user selects "No", the test is considered failed, and the system offers the option to retry or exit the Production Test Tool.

This final test ensures that the camera hardware is operational, and that the device is ready for tasks requiring visual input, such as barcode scanning or photo-based verification.

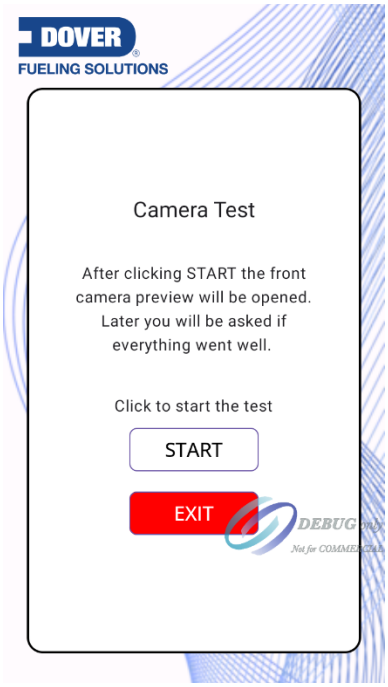

Figure 45: Camera test description

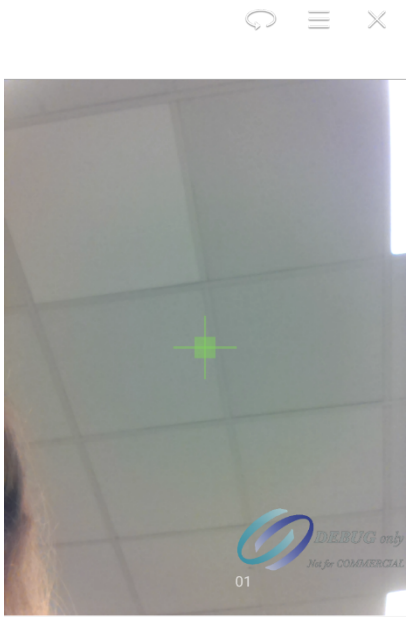

Figure 46: Camera test in progress

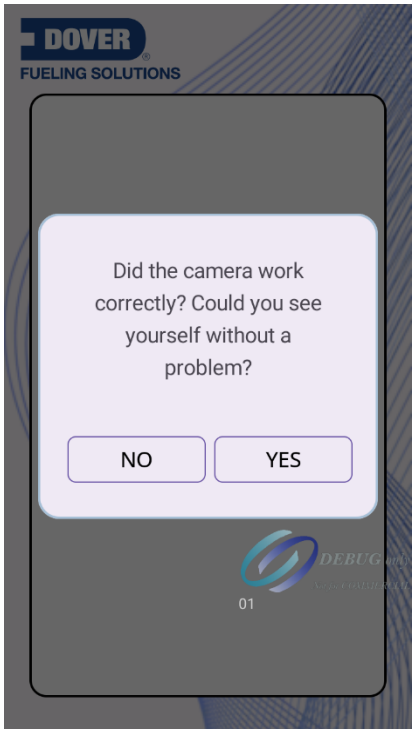

Figure 47: Camera test confirmation dialog

3.6.11. Test Results Overview

The Test Results Overview screen appears at the end of the testing process and serves as a final summary of all completed steps. It shows the outcome of each individual test in the sequence, clearly indicating whether it was passed or failed.

This screen is displayed in two scenarios:

1. When the user completes the full test sequence successfully or with failed tests: the overview is shown automatically after the last test (Camera Test).
2. When the user decides to stop the test flow early: the overview is still shown, containing the results of only the tests that were completed (or failed) up to that point.

The overview provides a clear and structured report of the testing session, allowing the user to review what has been tested and what remains unchecked. At the bottom of the screen, there is an additional option to print the overview as a physical ticket, if the printer is connected and working properly.

This summary ensures transparency in the test results and gives users a final opportunity to document or evaluate the current state of the terminal before it is approved for further use.

At the very beginning of the project, the test was supposed to only contain the information on which test has passed. After working on the screen, I have realized it would be interesting to also put some statistical information like how many different tests in general were taken, how many attempts was there in total, and for the specific tests. The time was also added, so the user can see how long it took inside of all the test. Only the “in progress” phase of the test is counting to the final count on the overview page.

But after one of the QA tests, I was suggested to maybe put some more information on the screen (which later can be printed out as the current ticket). The main idea this time, was to create something that can be taken up by the manufacturing worker, archived and put out in the case that something was wrong with the terminal. So, this way at the very top of the ticket, the serial number of the terminal was added. Next to every test, there was also a date and the hour of the completion. This way, the ticket can also serve as a kind of a proof, if somebody is belittling the performance of the terminal (or if seriously something is wrong) to show that it was working correctly before (because it passed the tests, so it had to function well, otherwise it would be impossible to print the ticket.

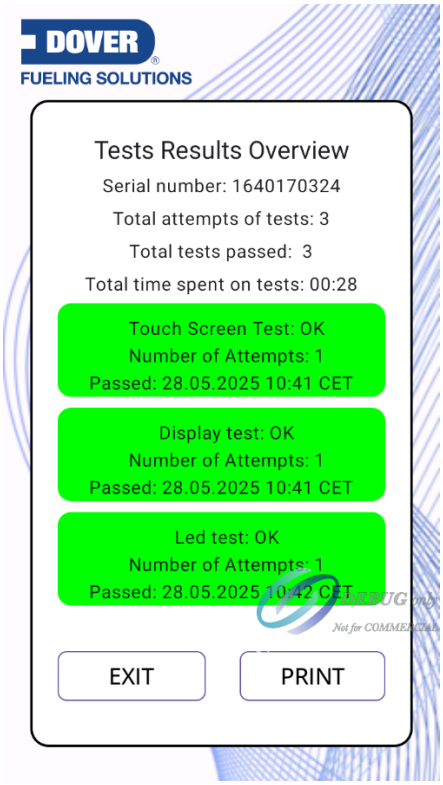

Figure 48: Tests results Overview after normal flow

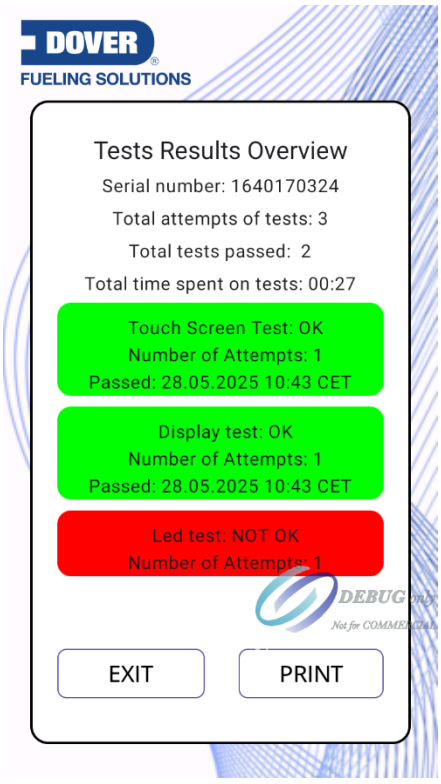

Figure 49: Test results overview after failing a test and quitting

3.7. Unit Test Feature

The Unit Test Feature was an idea that emerged during the planning stage of the project, while I was discussing the assignment with my mentor and defining the overall structure. It was not part of the original requirement, but it quickly proved to be a useful addition to the tool.

The purpose of this feature is to allow the user to run a single, specific test without going through the entire predefined test sequence. This can be especially helpful in situations where only one component needs to be checked or retested. For example, when troubleshooting a failing device or verifying that a fix was successful.

Instead of being locked into the full test flow, the user can simply open the Unit Test Menu, select the desired test (e.g., barcode scanner, LED test, camera), and run it independently. All logic, UI, and functionality remain the same as in the standard flow. The only difference is that the navigation does not proceed automatically to the next test.

This feature improves flexibility, saves time, and makes the Production Test Tool more practical in real-world scenarios where focused testing is sometimes necessary.

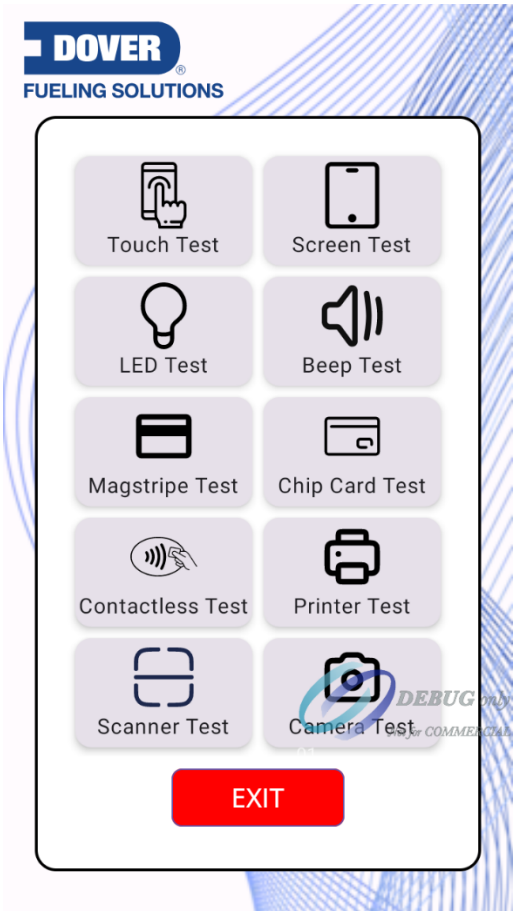

Figure 50: Main screen of unit tests feature

3.7.1. The Flow and Its Implementation

As up till now there was only one flow available in the app. Only passing all the tests guaranteed getting through the application – meaning we only expect fully functional devices to be positively tested. But if for some reason, there is an indication that something should be tested separately, the test unit feature menu allows it. There, it's possible to test everything separately.

Every test is considered a different entity, and they are not tied to each other in any sense. They can be run independently. Most of the tests follow the same flow as in the original version, but there are some exceptions to make sure that the test is assessing the full capacity of testing. There are also some changes in descriptions – to make it very clear to the user (in default version: a manufacturing worker) what to do in the specific test. There is a special menu, designed just for this feature.

Also, because the tests have the same logic (as in the test itself), but they do differ in the look, and in the general flow (so how it is supposed to look in the application, what happens when, that there is no strict connection between the tests), the idea of creating the "base" tests emerged. So, the solution works around creating the base and unit test version of each test. Then take out the parts that are the same, or very similar and then create a different composable which holds them and reuse it in both variants. This way, we get reusable code for every test that is also well thought, and ready in case any changes are needed. Some of the elements are already thought of and can be added beforehand as an argument. We don't need to write a lot lines of code, just match everything we need for the specific moment.

3.7.2. Differences Between Normal Tests and Unit Test Logic

So, as it was stated in a paragraph before – most of the tests stay the same, but some have less restrictions. In this section, they will be briefly explained, to understand the difference between the normal flow and the unit testing:

- Magstripe Test – In unit testing mode, the Magstripe Test operates with fewer restrictions compared to the normal application flow. Any card with a readable magnetic stripe can be swiped, and its tracks will be displayed on the screen. There is no limitation to a specific card; multiple cards may be used. The test is only marked as failed if a timeout occurs. This means that no card was swiped

within the allowed time, or the swiped card did not contain readable tracks and was not detected by the reader.

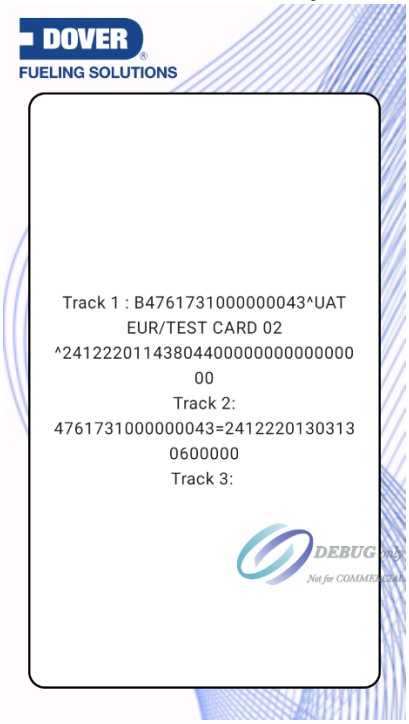

Figure 51: Magstripe Unit Test Result

- Scanner test – A lot of formats of the barcodes can be read. (EAN-13, EAN-8, UPC-A, QR Code, etc). The test is considered correct if any kind of barcode is read. When it's read then the content is displayed on the screen. The only way to fail this test is to run into the timeout, so if none of the barcodes was read by the rear camera on the bottom of the device.

Figure 52: Scanner Unit Test - QR Code

- Printer test – a new screen was created just for this, alongside with the special grid. The idea behind it: the printer should be able to work and print many different things, and if somebody wanted to test all of them at the same time, it would be very inefficient to access the test 3 times, so a special screen was designed, where you can print them all in a smooth flow, without the need to leave the printer unit test, which is perfect for the user experience.

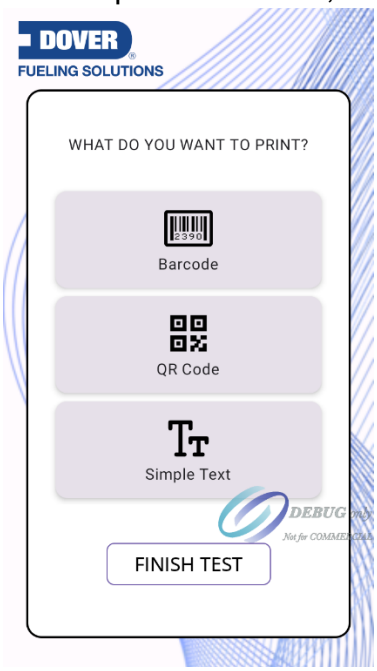

Figure 53: Printer Unit Test Menu

There is also a special check if the printer is connected. If it is not, then there is a special pop-up appearing informing user about the problem. If the lid is opened, there is also a special pop-up that the printer is not fully functional. There is not a moment, where the tool is frozen (or even worse: crash)

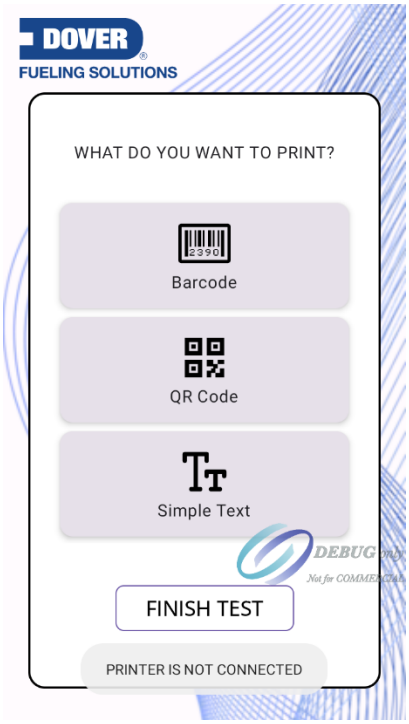

Figure 54: Printer not connected pop-up

4. Conclusion

The goal of this internship was to design and build a tool that could test whether PAX devices are fully functional before being delivered to customers. The focus was on the manufacturing workers though. To ensure that they can test the device in the production environment. The result is a complete application that follows a clearly defined test sequence, covers every relevant hardware component, and provides users with a structured and reliable way to validate each device. It also includes additional features like unit testing and printable summaries, which increase the tool's flexibility and practical value.

Looking back at the process, the project combined many different areas of development: from dependency setup and architecture to UI logic, hardware integration, and reusable components. Since app development was not originally the core focus of my academic background, this assignment pushed me to learn and explore new tools and concepts (especially in Kotlin, Jetpack Compose, and Gradle).

In terms of objectives, I believe the result meets and even extends the expectations outlined in the original project plan. The application is fully functional, scalable, and aligned with real needs in a production environment. It follows best practices and stays modular, which should make future maintenance or extensions easier for anyone continuing the project.

There are always opportunities to build on what exists. Features like database integration to store results, more detailed error feedback for failed tests, or multilingual UI support could further enhance the tool. However, even in its current form, the Production Test Tool delivers what it was meant to be. A reliable and efficient way to verify the quality of PAX devices before deployment and an easy way for manufacturing workers to check it.

References

Deepak, M. (2025, January 7). *The rise of Kotlin: Android development in 2025*. Opgehaald van Medium: https://medium.com/infosecmatrix/the-rise-of-kotlin-android-development-in-2025-d826df946932

Luizzi, J. (2024, December 18). *Translating Java to Kotlin at scale*. Opgehaald van Facebook engineering: https://engineering.fb.com/2024/12/18/android/translating-java-to-kotlin-at-scale

Asoyan, A. (2025, February 23). *Compose vs XML in 2025: Still using it?* Opgehaald van Medium: https://artemasoyan.medium.com/compose-vs-xml-in-2025-still-using-it-%EF%B8%8F-69fad33a2907

Bellini, A.-C. (2001, July 28). *Jetpack Compose is now 1.0: announcing Android's modern toolkit for building native UI*. Opgehaald van android-developers: https://android-developers.googleblog.com/2021/07/jetpack-compose-announcement.html

Google. (sd). *Build better apps faster with Jetpack Compose*. Opgehaald van Android: https://developer.android.com/compose

Developers, A. (sd). *Navigation with Compose.* Opgehaald van Android: https://developer.android.com/develop/ui/compose/navigation