Silesian
University
of Technology

FINAL PROJECT

Application for supporting dice game

Oluwamayokun Moyinoluwa SOFOWORA
Student identification number: 299531

Programme: Informatics
Specialisation: Informatics

SUPERVISOR
dr inż. Jacek Stój
DEPARTMENT OF DISTRIBUTED SYSTEMS AND INFORMATIC DEVICES
Faculty of Automatic Control, Electronics and Computer Science

Gliwice 2025

Thesis title

Application for supporting dice game

Abstract

This paper presents the development and evaluation of a mobile application, built with Kotlin, that enhances dice games by integration of artificial intelligence (AI) and image recognition. The application features an AI algorithm that adapts to the player's skill level, providing a dynamic and challenging opponent across various dice games. Additionally, it includes a dice image recognition system for real-time detection of dice rolls, ensuring a seamless gaming experience. The app simulates several classic dice games, such as *Pig*, *Balut*, and *Greed*. This paper also discusses the technical challenges encountered during the implementation of these features. It compares the application's performance and user experience to traditional dice games, with a focus on the integration of AI and the impact of image recognition on gameplay.

Key words

Kotlin, Android development, Computer Vision, Jetpack Compose

Tytuł pracy

Aplikacja do wspomagania gry w kości

Streszczenie

W tym artykule przedstawiono rozwój i ocenę aplikacji mobilnej, zbudowanej w języku Kotlin, która ulepsza gry w kości poprzez integrację sztucznej inteligencji (AI) i rozpoznawania obrazu. Aplikacja zawiera algorytm AI, który dostosowuje się do poziomu umiejętności gracza, zapewniając dynamicznego i wymagającego przeciwnika w różnych grach w kości. Ponadto zawiera system rozpoznawania obrazu kości do wykrywania rzutów kością w czasie rzeczywistym, zapewniając płynne wrażenia z gry. Aplikacja symuluje kilka klasycznych gier w kości, takich jak *Pig*, *Balut* i *Greed*. W artykule omówiono również wyzwania techniczne napotkane podczas wdrażania tych funkcji. Porównano wydajność aplikacji i wrażenia użytkownika z tradycyjnymi grami w kości, ze szczególnym uwzględnieniem integracji AI i wpływu rozpoznawania obrazu na rozgrywkę.

Słowa kluczowe

Kotlin, rozwój Androida, Wizja komputerowa, Kompozycja Jetpack

Contents

Chapter 1

Introduction

In recent years, mobile gaming has seen rapid growth, yet many games fail to deliver the complexity and engagement necessary for long-term player retention [1]. Traditional dice games, with their simple rules and strategic depth, offer an opportunity to innovate within the mobile gaming space. However, many mobile dice games often fall short due to *predictable AI behavior* and a lack of meaningful player feedback [1]. This can lead to gameplay that feels repetitive, with easily exploitable strategies and minimal opportunities for players to improve [2]. Such shortcomings often result in disengaged players who struggle with steep learning curves, lacking any real insights into their progress.

1.1 Objectives

This thesis presents the development of a mobile application designed to support and enhance dice games through the integration of *artificial intelligence (AI)* and *image recognition technology*. The goal of this project is to create an immersive, intelligent, and interactive mobile gaming experience by addressing the following key objectives:

1. To implement an adaptive AI system that adjusts to the player's skill level across various dice games.

2. To develop an image recognition system that allows users to simulate dice rolls using their device's camera.

3. To enhance the user experience by providing dynamic gameplay, real-time interaction, and a seamless transition between the physical and digital worlds.

[1] Dice games have a rich history dating back thousands of years, with early examples found in ancient cultures such as Mesopotamia and Egypt. The simplicity and randomness of dice rolls have made them a staple in games of chance and strategy.

1.2 Project Requirements

This section outlines the essential requirements that the project aims to fulfil to achieve its objectives.

- Support for classic dice games such as *Pig*, *Balut*, and *Greed*.
- Integration of an AI engine capable of adapting to player strategies and behaviors.
- Real-time image recognition to detect dice patterns and simulate rolls.
- A mobile application built with *Kotlin* and *Jetpack Compose* for a responsive and intuitive interface.
- Efficient performance with real-time gameplay and minimal latency.

The application is built using *Kotlin* and *Jetpack Compose*, offering a blend of traditional dice game mechanics and modern technological enhancements. The AI component provides a *challenging and adaptive opponent*, while the image recognition system offers a seamless gaming experience that bridges the gap between the physical and virtual worlds.

1.3 Thesis Structure

This thesis is structured into seven chapters, each addressing a specific aspect of the project.

1. Chapter 1 introduces the mobile gaming landscape, along with the project's goals and requirements.
2. Chapter 2 outlines the system requirements, architecture design, data model, user interface, and AI model design.
3. In Chapter 3, the problem is accessed, exploring the mechanics of dice games, the role of artificial intelligence (AI) in gaming, and the architecture of mobile applications.
4. The external specifications are detailed in Chapter 4, covering the development of the Android application, integration with RoboFlow API, and the implementation of player analytics and game mechanics.
5. Chapter 5 covers the internal specifications, including unit testing, integration testing, and user acceptance testing, as well as performance analysis and evaluation of the AI model.
6. Verification and validation processes are discussed in Chapter 6, ensuring the system meets its requirements and functions correctly.

7. Finally, Chapter 7 summarizes the key elements of the thesis, reflecting on its achievements, limitations, and offering suggestions for future improvements.

Chapter 2

Requirements and tools

The project requirements outline the system's expected functions and performance, detailing the features and capabilities that meet user needs. This forms the foundation for designing and implementing the system's core functionalities.

2.1 Functional Requirements

Functional requirements define the specific behavior and functions of the system. For the dice game application, these include:

1. **Basic Functions**: The system must implement fundamental dice game mechanics, including:
    - Ability to play classic dice games
    - Rolling dice with randomized results
    - Holding selected dice between rolls
    - Banking scores based on game rules
    - Displaying current turn results and game progress
2. **AI Opponent**: The system should feature an AI opponent that dynamically adapts to player behavior and skill level.
3. **Game Variants**: The application should support a variety of dice games, providing diverse gameplay options.
4. **Player Analytics**: The system should track and analyze player statistics to inform AI decision-making.
5. **User Interface**: The application should provide an intuitive and consistent user interface for all game variants.

6. **Real-Time Feedback**: Users should receive performance feedback to understand and improve their gameplay strategies.

7. **Image Recognition**: The system should accurately detect and recognize physical dice through the device camera, detecting dice faces in real-time and accurately recognizing pip values.

2.2 Non-functional Requirements

Non-functional requirements are essential for ensuring the quality and performance of the system. They help address issues such as latency, scalability, usability, reliability, and security. For the application, the following non-functional requirements are identified:

1. **Performance**: The application should deliver a smooth user experience on mobile devices, with minimal latency in AI decision-making.

2. **Scalability**: The system should be able to handle an increasing number of users and game variants.

3. **Usability**: The user interface should be easy to navigate and accessible to diverse users.

4. **Reliability**: The application should consistently provide accurate AI behavior and player analytics, maintaining user satisfaction.

2.3 Description of Tools and Technologies

The selection of tools and technologies for this project was driven by several key considerations, those of which include development efficiency, modern Android development practices, and the specific requirements of building a dice game with computer vision capabilities. Each tool was chosen to address specific aspects of the development process, from the core programming language to specialized libraries for UI components and testing. The following sections detail these choices and their contributions to the project's success.

2.3.1 Technologies and Tools

Kotlin

Kotlin is a modern programming language that offers features like null safety, extension functions, and interoperability with Java, making it a preferred choice for Android development. Kotlin was chosen for this project to write the entire application code because of

these advantages: its concise syntax reduces boilerplate and enhances readability, while its features like null safety improve application stability by reducing the risk of null pointer exceptions. Furthermore, its seamless integration with the Java ecosystem ensures that it can readily use Java libraries and existing Android code, and the language's ability to be compiled down to bytecode means increased cross-platform support. These advantages together result in enhanced productivity and maintainability [3].

Roboflow

Roboflow is a computer vision platform that offers tools for dataset management, model training, and deployment [4]. Roboflow was selected for this project because it significantly simplifies the process of building and deploying a custom dice detection model. The platform's efficient dataset preparation and augmentation capabilities helped improve the robustness of the model despite not having a large custom dataset [5]. Moreover, its tools for model training and optimization allowed for the efficient creation of a model suitable for mobile deployment. Importantly, Roboflow's managed API provides a seamless way to integrate the trained model into the Android application, enabling real-time inference capabilities that are crucial for the interactive dice-recognition component of the application, thereby enhancing the application's performance and reliability.

Figma

Figma is a collaborative interface design tool that was used to create the application's initial designs and prototypes [6]. Figma was used due to its collaborative nature allowing different stakeholders to collaborate during the design process and provide fast feedback. The tool provided a good level of functionality, and was used to create the user interface that could be implemented in the project, saving time during the development process, reducing effort.

Git

Git is a distributed version control system that allows developers to track changes in their code-base through GitHub, collaborate with others, and manage project history efficiently. Git, hosted by GitHub, was chosen for source code management in this project because it enables collaborative development, allowing multiple developers to work on the project simultaneously without encountering conflicts. Additionally, Git's ability to track code versions is essential for maintaining the project's history, enabling developers to revert to previous states and maintain overall code integrity efficiently [7].

Android Studio

Android Studio is the official integrated development environment for Android development, providing tools for building, testing, and debugging Android applications. Android Studio was used because it provides a comprehensive toolset that enables an efficient workflow. Its features, including a code editor, debugging tools, and emulator support, are essential for developing Android applications quickly and effectively. The deep integration with the Android platform helps in ensuring the application works well on the intended target platform [8].

Jetpack Compose

Jetpack Compose is a modern toolkit for building native Android UI, offering a declarative approach that simplifies UI development and enhances code readability. It provides modular re-composition, allowing UI elements to update independently, which optimizes UI rendering efficiency. Although comparisons suggest that XML-based layouts may achieve better rendering speeds in certain scenarios, Jetpack Compose was chosen for its advantages in terms of flexibility, maintainability, and modern approach to Android UI development [9]. Jetpack compose also allows for fast and iterative development, which is beneficial for this project [10].

2.3.2 Libraries

Dagger-Hilt

Dagger-Hilt is a dependency injection library for Android that simplifies the setup and management of dependencies in Android applications. Dagger-Hilt was used in this project because it greatly reduces boilerplate code related to dependency injection, improving code modularity and making the project more testable. This is especially important for larger projects, allowing for more structured and organized code. It allows the application to easily scale, and can help with long-term maintainability [11].

Lottie Animation

Lottie is a library for rendering animations in real-time, allowing developers to use animations created in Adobe After Effects in their applications. Lottie was selected for its capabilities in rendering vector-based animations smoothly and efficiently, enhancing the application's user interface. Lottie allowed animations to be created by UI/UX designers using their own tools and then be added into the application without requiring the implementation to be done by programmers. This allows for smoother animations and reduces the technical complexity and improves the user experience [12].

Vico Charts

Vico Charts is a library for creating interactive and customizable charts in Android applications, providing a variety of chart types and features. Vico Charts was chosen because it offers a wide range of chart types and features that make it easy to represent data visually and dynamically. Vico Charts allows users to more easily interpret their data, and the library's extensive customization options enable the data to be displayed attractively and consistently with the user interface [13].

Timber

Timber is a logging library for Android that provides a simple and flexible API for logging messages, making it easier to manage log output in Android applications. Timber was used for logging in this project to simplify and standardize the process of logging events and debugging information. Timber removes the clutter that comes from logging, and offers an elegant and easily readable interface and functionality for developers, thereby aiding in debugging and monitoring the application's behavior [14].

JUnit 5

JUnit 5 is composed of several modules, including JUnit Jupiter which is a combination of the programming and extension model for writing tests and extensions in JUnit 5 [15]. JUnit 5 was chosen as the testing framework because of its modern architecture which allows for cleaner and more maintainable tests. It provides advanced testing functionalities and allows for an ease of development, allowing for the implementation of unit and integration tests with ease.

Chapter 3

Problem Analysis

Developing a dice game that uses computer vision and artificial intelligence presents unique challenges. This chapter systematically explores these challenges and the solutions implemented in the proposed solution. It is organized into several sections, each addressing a key aspect of the project:

- **Computer Vision**: Explores the technical hurdles in dice detection and recognition, including lighting variations, perspective distortions, background complexities, and real-time processing requirements. It also discusses the image preprocessing techniques employed to enhance detection accuracy.

- **Model Architecture**: Details the design and implementation of the AI model, including the selection of pre-trained object detection models, training on custom datasets, and integration with Roboflow's Inference API for efficient dice recognition.

- **Asynchronous Processing**: Describes the use of Kotlin Coroutines to manage background tasks, ensuring a responsive user interface. This section covers coroutine scopes, launching coroutines, suspend functions, and error handling mechanisms.

- **Gameplay**: Analyzes the core game mechanics, including the scoring algorithm, adaptive AI behaviors, and the overall game flow. It highlights how these elements contribute to an engaging and balanced gameplay experience.

- **Existing Solutions**: Reviews current approaches and technologies in virtual dice games, and dice detection and recognition, comparing various models and platforms to contextualize the thesis solution's contributions and improvements.

By addressing these components, the chapter lays the foundation for understanding the technical and strategic decisions that underpin the development of a robust and engaging dice game application.

3.1 Computer Vision

Computer vision plays an important role in seamlessly integrating physical dice into digital gameplay. This section delves into the challenges and solutions associated with dice detection and recognition, highlighting the critical techniques and advancements in the field.

Computer vision is an essential component in developing applications that interact with the physical world through visual data. It encompasses a wide range of techniques and algorithms designed to enable machines to interpret and understand visual information [16]. One of the foundational works in this domain is Szeliski's comprehensive guide, which explores fundamental algorithms and their applications across various sectors. This work provides invaluable insights into image processing, feature detection, and machine learning techniques that are crucial for advancing modern computer vision applications.

The advent of deep learning has revolutionized computer vision, particularly through convolutional neural networks (CNNs) that have demonstrated exceptional performance in image classification tasks [17]. These networks have enabled more accurate and efficient processing of visual data, thereby enhancing the capabilities of computer vision systems.

A significant advancement in real-time object detection is the YOLOv3 architecture introduced by Redmon and Farhadi. YOLOv3 balances speed and accuracy effectively, making it highly suitable for applications requiring real-time performance [18]. This architecture has been instrumental in improving the efficiency of object detection tasks, including the recognition of dice in dynamic environments.

Current surveys underscore the continuous innovations and emerging trends in deep learning algorithms for image classification. These advancements emphasize the growing applicability and performance enhancements of computer vision technologies in various applications [19]. The integration of these cutting-edge techniques is fundamental to overcoming the challenges inherent in dice detection and recognition, thereby facilitating a more immersive and interactive gaming experience.

3.1.1 Challenges in Dice Recognition

The implementation of accurate dice recognition presents several technical challenges:

- **Lighting Variations**: Dice faces appear differently under various lighting conditions. This includes shadows that can obscure the patterns of pips on the dice, reflective surfaces causing glare, and significant differences between indoor and outdoor lighting that affect contrast and visibility.

- **Perspective and Orientation**: The system must handle dice captured at different angles, which affects how pips appear in the image. Multiple dice can overlap or

occlude each other, and the distance between the camera and dice impacts pip visibility and overall recognition accuracy.

- **Background Complexity**: Various playing surfaces can affect detection reliability. Similar patterns in the background may trigger false positives, while moving backgrounds, such as when playing on unstable surfaces, can further complicate the detection process.
- **Real-time Processing Requirements**: The system must process frames quickly for a responsive user experience. This involves careful management of battery consumption and memory usage, requiring optimized processing algorithms and efficient resource management.

3.1.2 Image Preprocessing

The system employs a sophisticated preprocessing pipeline using Android's built-in Bitmap processing capabilities and the AndroidX Core-KTX library for image manipulation. The preprocessing steps, illustrated in Listing 3.1, utilize several key components:

```
1   private suspend fun preprocessImage(bitmap: Bitmap): Bitmap {
2       return withContext(Dispatchers.Default) {
3           try {
4               // Step 1: Convert to RGB if needed
5               val rgbBitmap = ensureRGBFormat(bitmap)
6
7               // Step 2: Enhance contrast and normalize lighting
8               val enhancedBitmap = enhanceContrast(rgbBitmap)
9
10              // Step 3: Scale while maintaining aspect ratio
11              val scaledBitmap = scaleWithAspectRatio(enhancedBitmap, TARGET_SIZE)
12
13              // Step 4: Apply noise reduction
14              val finalBitmap = reduceNoise(scaledBitmap)
15
16              Timber.d("Preprocessing completed successfully")
17              finalBitmap
18          } catch (e: Exception) {
19              Timber.e(e, "Error during image preprocessing")
20              // Fallback to basic scaling if enhancement fails
21              Bitmap.createScaledBitmap(bitmap, TARGET_SIZE, TARGET_SIZE, true)
22          }
23      }
24  }
```

Listing 3.1: Image Preprocessing Pipeline

- **RGB Conversion**: Uses Android's ColorSpace API to ensure consistent color representation
- **Contrast Enhancement**: Implements a histogram equalization algorithm through Android's ColorMatrix

- **Aspect Ratio Scaling**: Utilizes Android's createScaledBitmap with custom calculations for dimension preservation
- **Noise Reduction**: Applies a Gaussian blur filter using Android's RenderScript (now deprecated) or equivalent Core-KTX functions

The implementation is shown in Listing 3.1, with each step carefully orchestrated to maintain image quality while optimizing for dice detection.

This preprocessing pipeline ensures optimal image quality before the detection phase, which is further detailed in the Model Architecture section.

3.2 Model Architecture

The dice recognition system leverages a pre-trained object detection model from Roboflow [5]. The model processes images at 640x640 resolution and was trained on a custom dataset of 250 images, supporting six distinct classes representing dice faces 1-6. Developed and hosted on Roboflow's platform, it provides efficient object detection capabilities through their API service.

The model is then accessed through Roboflow's Hosted Inference API, with preprocessing handling as shown in Listing 3.1.

The preprocessing pipeline starts by converting the image to RGB format to standardize color space, then contrasts enhancement and lighting normalization to improve visibility, particularly under varying conditions. The image is then resized while preserving its aspect ratio to maintain recognition accuracy, then noise reduction removes unwanted artifacts that could interfere with recognition, especially in suboptimal conditions. If preprocessing fails, the system defaults to basic scaling to ensure usability.

- RGB format conversion:
- Image scaling to the required 640x640 dimensions
- Confidence threshold (set at 0.4 for reliable detections)

For each detected die, the model outputs bounding box coordinates, confidence scores, and class labels, which the application processes to update the game state.

The detection pipeline evolved throughout development, starting with a basic implementation and later expanding to include more sophisticated preprocessing and validation. The pipeline was later enhanced to improve detection reliability through:

- **Advanced Preprocessing**: Implementation of RGB format conversion, contrast enhancement, and adaptive scaling while maintaining aspect ratios.

- **Robust Validation**: Addition of comprehensive detection validation including aspect ratio checks, minimum size requirements, and position validation.
- **Quality Filters**: Implementation of confidence threshold and noise reduction techniques.

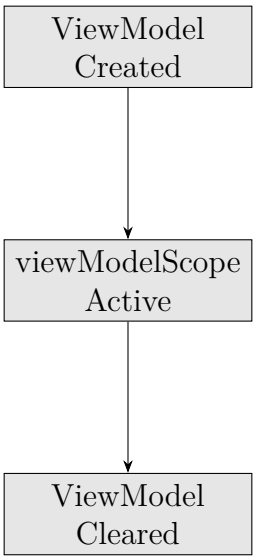
This enhanced pipeline significantly improved detection accuracy and reliability across various lighting conditions and capture scenarios.

3.3 Asynchronous Processing

Kotlin Coroutines are utilized to efficiently manage asynchronous updates, ensuring the application remains responsive. This section explores the benefits of coroutines and their role in handling background tasks within the application.

3.3.1 Coroutine Scope

The `viewModelScope` is tied to the lifecycle of the ViewModel. This ensures that coroutines are automatically canceled when the ViewModel is cleared, preventing memory leaks and unnecessary processing.



3.3.2 Launching Coroutines

The `launch` function starts a new coroutine, allowing non-blocking execution. This is crucial for processing tasks like dice value detection or data loading, which can be time-consuming. The `viewModelScope` is tied to the ViewModel's lifecycle, ensuring that any running coroutines are automatically cancelled when the ViewModel is cleared, preventing memory leaks and unnecessary background work.

```
viewModelScope.launch {
    // Load vibration setting
    dataStoreManager.getVibrationEnabled()
        .collect { enabled -> _vibrationEnabled.value = enabled }
}
```

Listing 3.2: Launching a Coroutine

3.3.3 Suspend Functions

Suspend functions are a key feature of Kotlin's coroutine system, marking functions that can be paused and resumed. These functions can only be called from within a coroutine or another suspend function, ensuring proper asynchronous execution.

```
fun rollDice() {
    if (isRolling.value || !isRollAllowed.value) return
    viewModelScope.launch {
        trackDecision()
        trackRoll()
        _isLoading.value = true
        val results = diceManager.rollDiceForBoard(_selectedBoard.value)
        if (_vibrationEnabled.value) provideHapticFeedback()
        // Process game state after rolling
        val newState = processGameState(results)
        _gameState.value = newState
    }
}
```

Listing 3.3: Suspend Function Example

In Listing 3.3, the `rollDice` function is designed to manage the dice-rolling process within the game. It is executed within a coroutine scope using `viewModelScope.launch`, which allows it to perform asynchronous operations without blocking the main thread. This ensures that the UI remains responsive while the dice are being rolled.

The function begins by checking if a roll is already in progress or if rolling is not allowed, returning early if either condition is true. This prevents unnecessary operations and ensures that the game logic is executed only when appropriate.

Within the coroutine, the function tracks the player's decision and roll actions, providing valuable data for game analytics. It then sets a loading state to indicate that a roll is in progress. The actual dice rolling is performed by the `diceManager`, which returns the roll results.

If vibration feedback is enabled, the function provides haptic feedback to enhance the user experience. Finally, the function processes the game state based on the roll results and updates the game state accordingly.

The use of coroutines in this function allows for efficient management of asynchronous tasks, ensuring that the game logic is executed smoothly and without interruption. This design pattern is essential for maintaining a responsive and engaging user interface in a coroutine-based architecture.

3.3.4 Error Handling

The `try-catch` block within the coroutine handles exceptions, ensuring errors are logged and managed gracefully. This prevents crashes and maintains application stability.

```
viewModelScope.launch {
    try {
        val detections = roboflowRepository.detectDice(bitmap)
        _detectionState.value = if (detections.isNotEmpty()) {
            DetectionState.Success(detections)
        } else {
            DetectionState.NoDetections
        }
    } catch (e: Exception) {
        Timber.e(e, "Error detecting dice")
        _detectionState.value = DetectionState.Error(e.message ?: "Unknown error")
    }
}
```

Listing 3.4: Error Handling in Coroutines

The coroutine runs in the background, ensuring the main thread remains responsive to user interactions. Listing 3.4 illustrates the implementation of error handling within a coroutine. It processes collected data to update a LiveData property, enabling the UI to dynamically reflect any changes.

3.3.5 Interface Responsiveness

In interactive applications, maintaining a responsive user interface (UI) is critical to delivering a seamless user experience. By offloading computationally intensive tasks, such as AI decision-making and data processing, to background threads, the main UI thread remains available for handling real-time user interactions. This approach minimizes UI lag, ensuring that animations, gestures, and updates occur smoothly without delays.

For instance, in the context of dice games, background tasks such as calculating potential AI strategies or updating game states are delegated to coroutine-based background threads in Kotlin. This concurrency model enables a separation of concerns, where the UI layer focuses solely on rendering and responding to user input while backend logic operates asynchronously. The result is a user experience that feels intuitive and highly responsive, even under computationally demanding scenarios.

3.4 Gameplay

Making a game with lively gameplay presents many challenges, particularly when integrating adaptive artificial intelligence and ensuring a balance between challenge and accessibility. This section delves into the solutions employed to address these challenges.

3.4.1 Scoring Algorithm

The scoring system implements different calculations for each game variant:

Balut Scoring

The scoring for the Balut variant is calculated as:

$$S_{category} = w_c \times b_s \times (1 + (r - 1)(1 - l)) \times m_s \tag{3.1}$$

where $S_{category}$ represents the final score for a category, w_c is the category weight (1.0-2.5), b_s is the base score from dice combination, r is the random factor (0.5-1.5), l is the AI skill level (0-1), and m_s is the multiplier based on game state (0.6-1.2).

Pig Scoring

For the Pig variant, the turn score is calculated as:

$$S_{turn} = \begin{cases} \sum_{i=1}^{n} d_i, & \text{if } d_i \neq 1 \text{ for all rolls} \\ 0, & \text{if any } d_i = 1 \end{cases} \tag{3.2}$$

where S_{turn} represents the score for a turn, n is the number of rolls in the turn, and d_i is the value of the ith dice roll.

Greed Scoring

For the Greed variant, the turn score is calculated as:

$$S_{greed} = \sum_{i=1}^{k} (v_i \times m_i) + b \tag{3.3}$$

where S_{greed} represents the final score for a turn, k is the number of scoring combinations, v_i is the value of each scoring combination, m_i is the multiplier for each combination type (e.g., triples, straights), and b is any bonus points for special combinations. The scoring combinations are:

$$v_i = \begin{cases} 1000, & \text{if triple ones} \\ 200 \times n, & \text{if triple } n \text{ (where } n > 1) \\ 100, & \text{if single one} \\ 50, & \text{if single five} \\ 1500, & \text{if straight (1-6)} \\ 750, & \text{if three pairs} \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

3.4.2 Adaptive AI

In the game, the Adaptive AI simulates a dynamic and intelligent opponent that makes strategic decisions based on the player's play style and the current game state. The AI is integrated into different parts of the gameplay, including the classic games of *Pig*, *Greed*, and *Balut*.

In the game of *Balut*, when it is the AI's turn, it must decide whether to roll, hold, or bank dice based on the current game state. This decision-making process is implemented in the `handleAITurn` function. In the game of *Greed*, the AI determines which dice combinations to keep after each roll, aiming to maximize its score while minimizing risk. Meanwhile, in the game of *Pig*, the AI's decision to roll or bank is influenced by both the current score and the opponent's score, as well as the overall target score. Throughout these games, a game tracker records the AI's decisions, allowing for continuous refinement of its behavior.

The `handleAITurn` Function

The `handleAITurn` function in the game of *Balut* guides the AI's actions during its turn. It considers factors such as the number of rolls remaining, the values of the dice, and the potential scores of different categories. Depending on the game state, the AI either selects a category to score (when no rolls are left) or decides which dice to hold for the next roll. The Listing 3.5 illustrates the function:

```
1   private fun handleAITurn(
2       diceResults: List<Int>, currentState: BalutScoreState
3   ): BalutScoreState {
4       gameTracker.trackDecision()
5       if (currentState.rollsLeft <= 0) {
6           // AI chooses a category
7           val category = chooseAICategory(diceResults, currentState)
8           gameTracker.trackBanking(ScoreCalculator.calculateCategoryScore(diceResults,
    category))
9           return scoreCategory(currentState, diceResults, category)
10      }
11
12      // AI decides which dice to hold
13      gameTracker.trackRoll()
14      val diceToHold = decideAIDiceHolds(diceResults)
15
16      return currentState.copy(
17          rollsLeft = currentState.rollsLeft - 1,
18          heldDice = diceToHold
19      )
20  }
```

Listing 3.5: handleAITurn Function

The `shouldAIBank` Function: Adapting to Player Style

In various dice games, the AI's decision-making process is influenced by the human player's observed play style. By analyzing the player's tendencies, the AI adjusts its risk-taking behavior accordingly. The implementation varies across different games but follows a common principle: **aggressive players encourage riskier AI behavior, while cautious players make the AI more conservative**.

Pig: Adjusting Risk in Banking Decisions In the game of *Pig*, the AI decides whether to bank its current turn score or continue rolling. The AI considers the player's style before making this decision. Listing 3.6 demonstrates this adjustment:

```
val playerAnalysis = statisticsManager.playerAnalysis.value
val playerStyle = playerAnalysis?.playStyle ?: PlayStyle.BALANCED

// Adjust based on player style
val styleAdjustment = when (playerStyle) {
    PlayStyle.AGGRESSIVE -> -0.1  // More risk-taking
    PlayStyle.CAUTIOUS -> 0.1     // More conservative
    else -> 0.0
}
```

Listing 3.6: Player Style Adjustment

Here, the AI retrieves the player's style from the statistics manager. An aggressive player causes the AI to reduce its tendency to bank, encouraging riskier rolls, while a cautious player leads the AI to secure points more readily.

Greed: Dynamic Banking Thresholds In *Greed*, a more complex dice game, the AI incorporates both **a base banking threshold** and **situational adjustments**. The function in Listing 3.7 demonstrates how the AI modifies its strategy:

```
val baseThreshold = when (playerAnalysis?.playStyle) {
    PlayStyle.AGGRESSIVE -> 1200  // AI takes greater risks
    PlayStyle.CAUTIOUS -> 900     // AI banks earlier
    else -> 1000                  // Default threshold
}

// Adjust for game state
val situationalAdjustment = when {
    aiTotalScore == 0 -> -200  // Willing to risk when no points are banked
    aiTotalScore >= 8000 -> -300  // More aggressive near victory
    currentTurnScore >= 2000 -> 500  // Encouraged to bank large scores
    else -> 0
}

// Final threshold considers randomness for unpredictability
val finalThreshold = (baseThreshold + situationalAdjustment).coerceIn(800, 2000)
```

Listing 3.7: AI Banking Strategy in Greed

In this case, an aggressive opponent leads the AI to play more recklessly, increasing the threshold before banking. Conversely, a cautious opponent results in a lower threshold, making the AI secure its points earlier.

Balut: Play Style Influences Dice Holding and Category Selection *Balut* introduces additional layers of decision-making, including **dice-holding strategies** and **category selection**. The AI modifies its strategy based on the player's style:

Dice Holding:

```
val aiStyle = when (playerAnalysis?.playStyle) {
    PlayStyle.AGGRESSIVE -> 0.7  // More likely to hold high-value dice
    PlayStyle.CAUTIOUS -> 0.5    // More selective
    else -> 0.6
}
```

Listing 3.8: AI Dice Holding in Balut

Here in Listing 3.8, an aggressive player makes the AI hold more dice in an attempt to go for high-risk, high-reward combinations.

Category Selection:

```
val categoryScores = availableCategories.associateWith { category ->
    val baseScore = ScoreCalculator.calculateCategoryScore(diceResults, category)
    val weight = when {
        category == "Five of a Kind" -> if (baseScore > 0) 2.5 else 0.0
        category == "Large Straight" -> if (baseScore > 0) 2.2 else 0.0
        else -> 1.0
    }
    weight * (if (playerAnalysis?.playStyle == PlayStyle.AGGRESSIVE) 1.2 else 1.0)
}
```

Listing 3.9: AI Category Selection in Balut

Here in Listing 3.9, the AI prioritizes high-value categories when facing an aggressive player, taking bigger risks. Against a cautious player, it leans towards safer choices.

Across *Pig*, *Greed*, and *Balut*, the AI **adapts dynamically** to the player's style. This adaptability ensures a more engaging and challenging experience, making the AI feel responsive rather than static. Whether banking, holding dice, or selecting categories, the AI's decisions reflect the player's behavior, leading to a more immersive gameplay experience.

3.4.3 Game Mechanics

The game mechanics are crucial for delivering an engaging and intuitive gameplay experience. They are designed to ensure clear rules and interactions for both players and the AI, enabling a seamless game flow. An important component of the implementation is the `handleTurn` method, as shown in Listing 3.10. This method differentiates between player and AI turns and manages key actions such as dice holding and roll counting. Its

modular design supports clear separation of player and AI logic into distinct methods, reducing complexity and improving maintainability. This structure makes it easy to add new game modes or modify the AI behavior without disrupting existing functionality.

```
fun handleTurn(
        currentState: GameScoreState.PigScoreState, diceResult: Int? = null
    ): GameScoreState.PigScoreState = when (currentState.currentPlayerIndex) {
            AI_PLAYER_ID.hashCode() -> handleAITurn(currentState, diceResult)
            else -> handlePlayerTurn(currentState, diceResult)
        }
```

Listing 3.10: handleTurn Function

During its turn, the AI uses a blend of predefined rules and probabilistic decision-making to evaluate the game state and select the optimal strategy. Meanwhile, the game mechanics prioritize user experience by offering clear visual and interactive cues, ensuring that players can focus on strategy without being hindered by the interface.

In the `handleTurn` method as shown in Listing 3.10, the method differentiates between player and AI turns and manages key actions such as dice holding and roll counting.

- **AI Turn Handling:** If the current player's index matches the AI player's identifier, the method delegates the turn to `handleAITurn`, which implements the AI's decision-making logic. This includes evaluating the game state, deciding which dice to hold, and determining whether to bank a score or re-roll.

- **Player Turn Handling:** If the turn belongs to a human player, the method invokes `handlePlayerTurn`, which processes the player's actions, such as selecting dice to hold and performing a roll.

By isolating player-specific and AI-specific logic into separate methods, the design enhances code readability and maintainability. This modular approach ensures that updates or adjustments to AI strategies or player interactions can be made independently, maintaining the overall flow of the game.

This structured design allows for a compelling gaming experience by promoting a dynamic AI challenge while ensuring that interactions remain clear and responsive. The thoughtful integration of adaptive AI, clear gameplay mechanics, and user-focused design ensures that the game is accessible to players of all skill levels. Additionally, the modularity of the architecture enables the seamless incorporation of advanced features, such as multiplayer modes or new game variants, without disrupting the core mechanics.

Through these technical and strategic design choices, the project delivers an engaging dice game that is robust and scalable for future enhancements.

3.5 Existing Solutions

This section provides an overview of notable solutions in the domain of dice game applications and image recognition technologies. The analysis focuses on how each solution addresses dice recognition and gaming, while also contextualizing the unique enhancements introduced by this thesis.

3.5.1 D3-Deep-Dice-Detector

The D3-Deep-Dice-Detector utilizes deep learning techniques, specifically convolutional neural networks (CNNs), to detect and recognize dice in images. Its primary strength lies in accurately identifying dice numbers and face values under varying conditions of light and orientation [20].

While excelling at dice recognition, the system is focused mainly on detection. The solution presented in this thesis builds upon this by incorporating real-time updates and dynamic game interactions, further enriching the user experience with multiple game variants and adaptive AI.

3.5.2 Dice Scores Recognition

Dice Scores Recognition automates the scoring of dice games such as Yahtzee by interpreting dice configurations captured in images [21].

This solution targets predefined games with specific scoring rules. The proposed solution, however, expands its capabilities to cover multiple game variants (*Pig*, *Greed*, *Balut*) while introducing adaptive AI for a more interactive and personalized gameplay experience.

3.5.3 Zilch-Dice

Zilch-Dice is a Kotlin-based application designed for the Zilch dice game variant (also known as 10000), supporting Android, Linux, macOS, and Windows clients [22]. It focuses on score tracking, game state management, and player interactions, all aligned with Zilch's ruleset.

Although it caters to a single game variant, the proposed solution supports a wider array of dice games within a unified platform. This extended functionality, coupled with real-time dice detection and integrated adaptive AI, enhances player engagement and enables more dynamic gameplay.

3.5.4 Flutzy

Flutzy is a cross-platform dice game application that offers multiple game modes and

an intuitive user interface [23]. It aims for a seamless experience across different devices.

Flutzy is still under development, whereas the proposed solution is fully realized with Android-specific optimizations, including Jetpack Compose and Clean Architecture principles. The real-time dice detection and interactive gameplay in the proposed system provide a more responsive experience compared to Flutzy's current static modes.

3.5.5 Python-Dice

Python-Dice contains Python scripts for simulating dice rolls and calculating scores according to various game rules [24]. It serves as a backend tool for dice game logic without a dedicated user interface or real-time capabilities.

The proposed solution goes beyond simulation and scoring by offering a complete Android application with a user-friendly interface and integrated real-time dice detection, making it more suitable for end-users looking for an engaging, interactive dice gaming experience.

3.5.6 Dice Detection

Dice Detection by Nell Byler specializes in detecting and counting dice within images using image processing techniques [25]. It focuses on accurately identifying dice face values and quantities.

This thesis extends the capabilities of existing implementations with several key innovations:

- **Unified Approach**: Seamlessly integrates real-time dice detection with multiple game variants in a single application.
- **Adaptive AI**: Implements an intelligent opponent that learns from player behavior, unlike existing solutions that use static game logic.
- **Modern Architecture**: Utilizes current Android development practices with Jetpack Compose and Clean Architecture, providing better maintainability and user experience.
- **Real-Time Processing**: Offers immediate dice recognition and game state updates, compared to the static or delayed processing in existing solutions.

Chapter 4

External Specification

This chapter provides a detailed overview of the external specifications for the application. It outlines the specific requirements for its operation and the installation procedures necessary for a straightforward setup. Additionally, it includes key information to improve user understanding and ensure the application fulfils its intended purpose effectively.

4.1 Hardware and Software Requirements

4.1.1 Hardware Requirements

- Android smartphone with Android 11.0 (API level 30) or higher.
- Camera with support for CameraX.
- Minimum 2GB RAM for standard use of the application, or 3GB RAM for optimal performance on devices with lower memory and processing power.
- Internet connection for the dice recognition module; the other parts of the app do not require internet and will work otherwise.
- At least 100MB of free storage space.
- Processor: ARM-based processor supporting Neon instruction set.

4.1.2 Software Requirements

This section outlines the essential software tools and dependencies required for the development and execution of the application. Ensuring compatibility with these components is crucial for the stability and performance of the system.

- Android Studio Giraffe (2023.1.1) or later.
- Kotlin 2.0.21.

- Gradle 8.10.2 and the corresponding Android Gradle Plugin (e.g. 8.1.2).
- CameraX library version: 1.4.0.

4.2 Installation Procedure

This section details the necessary steps for installing and running the application on an Android device. Users must follow these instructions to ensure proper functionality and security.

4.2.1 APK Download

1. Download the Release APK from the project's repository at GitHub Releases Page.
2. Enable installation from unknown sources in the device's settings. Note that installing apps from unknown sources has security implications; please proceed with caution.
3. Open the APK file and follow the on-screen instructions.

4.2.2 Building with Android Studio

To install the application for debugging purposes, follow these steps:

1. **Clone the Repository**: Start by cloning the project repository from GitHub.

   ```
   git clone https://github.com/Mayokun-Sofowora/kavi.git
   ```

2. **Open in Android Studio**: Launch Android Studio and open the cloned project.
3. **API setup**: To use the dice recognition feature, the Roboflow API key must be configured. This section provides the necessary steps to obtain and configure the API key.
   - Sign in to Roboflow.
   - Navigate to your Project Dashboard.
   - Locate the API Key under the "Settings" or "Security" section.
   - Add the api key to the `local.properties` file:

     ```
     ROBOFLOW_API_KEY=api_key
     ```

4. **Sync Gradle Dependencies**: Allow Android Studio to sync the Gradle dependencies automatically.

5. **Run on a Device or Emulator**: Connect an Android device or start an emulator with a minimum SDK version of 30, then run the application.

In addition to the instructions provided, be sure that:

- A debug build variant is selected in Android Studio.
- USB debugging is enabled in the device's settings.
- You connect an Android device via USB or use an emulator.
- You select the appropriate run configuration in Android Studio.

4.3 Types of Users

- **Regular Users**: Play dice games, use image recognition features, and adjust settings.
- **Developers/Testers**: Access debugging logs, experimental features, and perform system administration tasks.

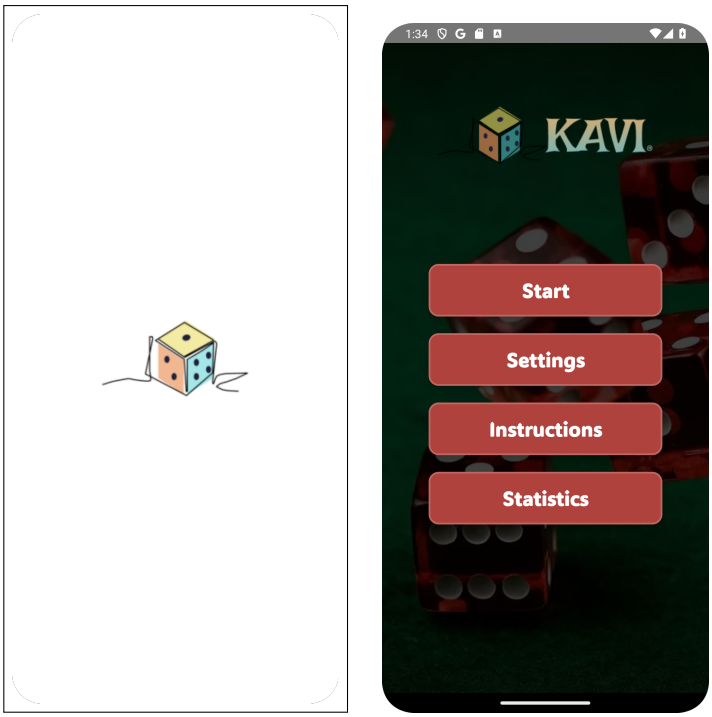
4.4 User Manual

4.4.1 Navigating the App

When starting the game, users are welcomed with a splash screen that displays the application's logo in Figure 4.1a. After a short moment, the main menu is shown in Figure 4.1b. The main menu provides navigation options to access the different features of the application.

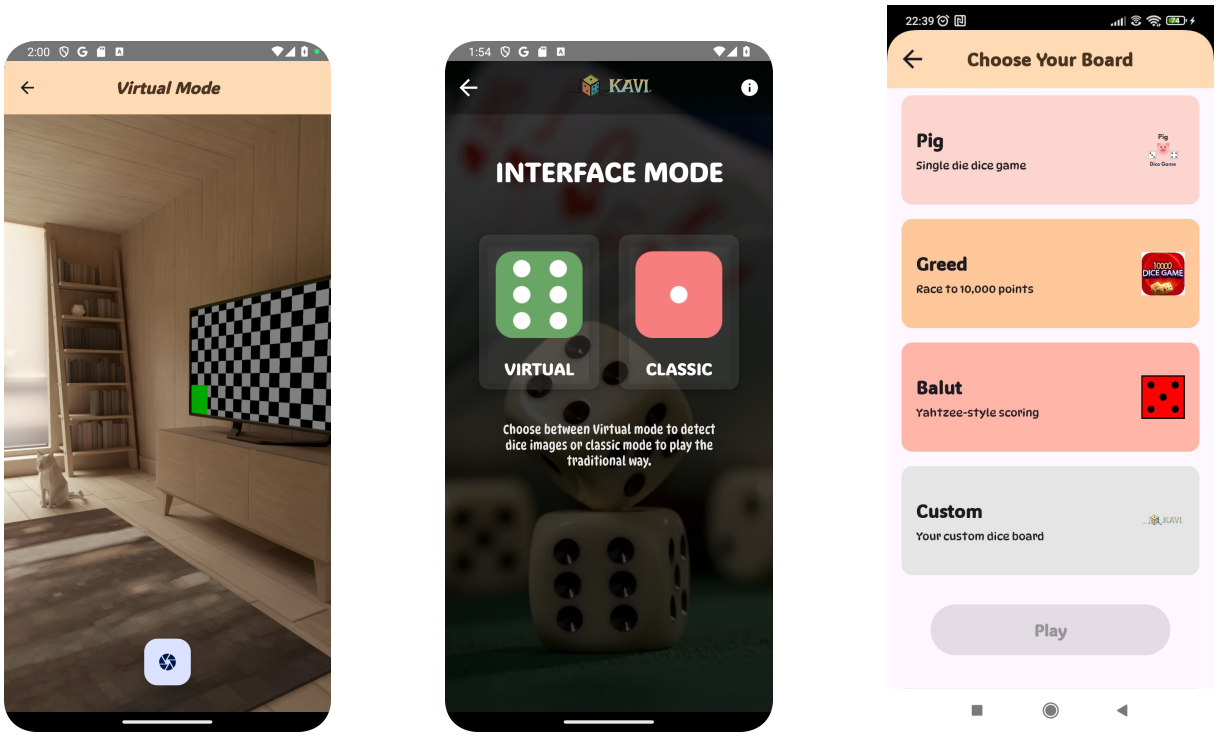
4.4.2 Game Interface

The interface allows users to navigate to different sections of the application, such as the classic boards to play the classic dice games or the virtual screen to use the image recognition feature. The Figure 4.2 shows the navigation options available.



(a) The Splash Screen (b) The Main Menu

Figure 4.1: Screens displayed when starting the game.



(a) Virtual Mode (b) The Start Screen (c) Classic Mode

Figure 4.2: The Games main interfaces.

4.4.3 Virtual Mode

Virtual Mode integrates real-time dice detection using the device's camera. This mode allows users to play with physical dice, enhancing the tactile and interactive nature of the game. To activate Virtual Mode, navigate to the main menu and select the "Virtual" option (Figure 4.2a). The application will then request permission to access the camera.

Once granted, the camera feed will be displayed on screen. Users can position their physical dice within the camera's view. Once the capture or retake button is pressed, the application will detect and recognize the values of the dice. The detected dice values are displayed on-screen, replacing the camera feed. If the system fails to accurately detect the dice, users can manually correct the detected values using on-screen controls. This ensures accurate scorekeeping even in challenging lighting or dice configurations.

4.4.4 Classic Boards

The application offers a diverse set of game boards, each designed for a distinct dice game experience. These games range from simple "press-your-luck" scenarios to more strategic challenges that require planning and risk management. The primary games offered are *Pig*, *Greed*, and *Balut*. Additionally, the application includes a custom board where users can define their own game rules and scoring mechanisms, and also select the number of dice used in the game. Figure 4.2c shows the classic boards available in the application. These games feature an adaptive AI opponent that adjusts its strategy based on the player's performance and the specific game rules. For a detailed explanation of the AI's behavior in each game, see Section 3.4.2.

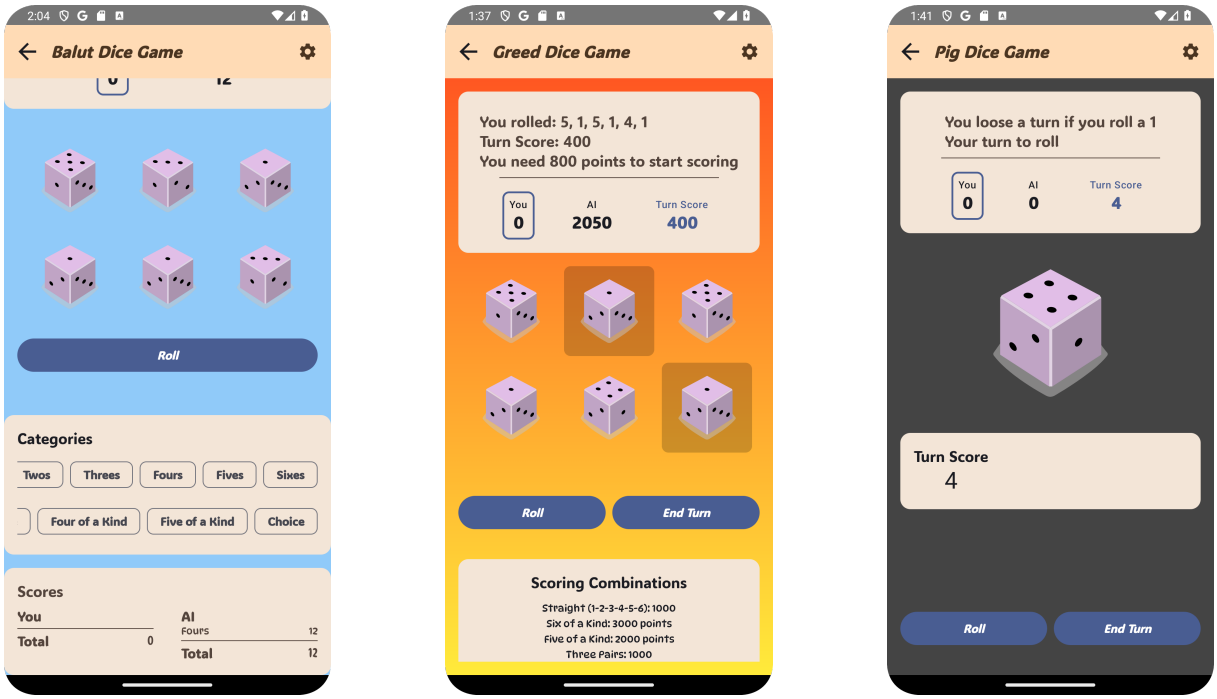
4.4.5 Game Objectives

Pig: The Risk of the Roll

Pig is a simple, engaging game of chance and risk. The goal is to be the first player to reach a total of 100 points. During each turn, players roll a single die, accumulating points with each roll. The key aspect of Pig is the ability to "bank" the points you have accumulated in that turn, however, the risk is that if you roll a 1, you lose all the points accumulated during that turn. The challenge lies in choosing when to press your luck for more points and when to play it safe to avoid losing those points.

Greed: Navigating Scoring Combinations

Greed is a more complex game that rewards strategic decision-making and risk-taking. Each turn begins with the roll of six dice. After the roll, players get to select which dice they want to keep to accumulate points, based on scoring combinations like straights, sets (e.g., three-of-a-kind, four-of-a-kind), and single 1s and 5s. These combinations vary in



(a) Balut Game Board (b) Greed Game Board (c) Pig Game Board

Figure 4.3: Game Boards in the Application

their scoring values, meaning that a key aspect of the game is knowing which combinations you should aim for. Unlike Pig, in Greed, players must accumulate at least 800 points in a turn to start banking them. The winner is the first player to reach a total of 10,000 points.

Balut: Strategic Category Management

Balut is a game of strategy, similar to Yahtzee. Players have up to three rolls per turn using five dice. The core of Balut is strategically filling the scoring categories, such as sets, straights, full houses, and more. Each category can be used only once, so planning and smart selection of which dice to hold is critical. After all categories are filled, the player with the highest total score wins. The game boards are shown in Figure 4.3.

4.4.6 Game Controls: Interacting with the Game

The application provides a user-friendly interface with intuitive controls, allowing players to easily interact with each game and manage their scores. This section talks about the various controls and how they function across different game modes.

Basic Game Controls (Roll, End Turn)

The primary way players interact with the games is through the roll button. In games like *Pig* and *Greed*, this button (shown in Figure 4.4) also serves as an "End Turn" button.

In these games, tapping the button rolls the dice and also ends the turn after a score is obtained.

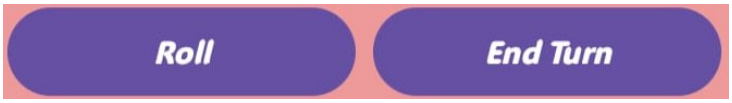


Figure 4.4: Roll and End Turn Button

Player Management and Custom Game Setup

In the custom game board, it is also possible to add players and edit their names (as shown in Figure 4.5). This function can be accessed by tapping a button which allows the player to add and remove player, as well as edit their names. This allows users to set up a new game of their liking.

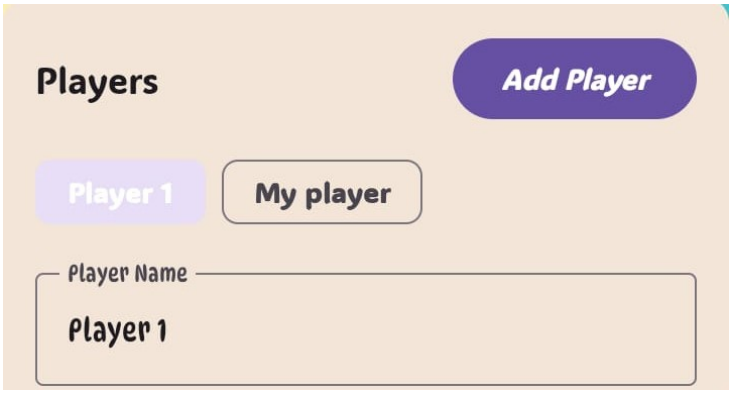


Figure 4.5: Player Management and Editing

Balut-Specific Controls (Category Selection)

Balut introduces the unique feature of category selection. After rolling the dice, a player can select a category in which to score, shown in Figure 4.6. This allows for a more strategic game, where each category can only be used once.

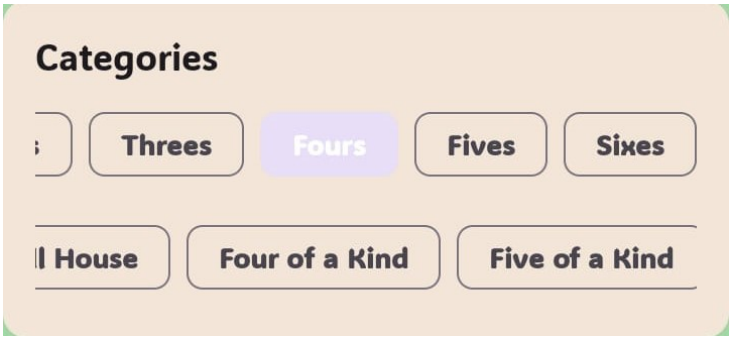


Figure 4.6: Balut Category Selection

Holding Dice

In both *Greed* and *Balut*, players can strategically select dice to hold for the next roll. As seen in Figure 4.7, this is done by tapping on the individual dice on the screen. The selected dice will be saved, and can be rolled again in the subsequent roll.

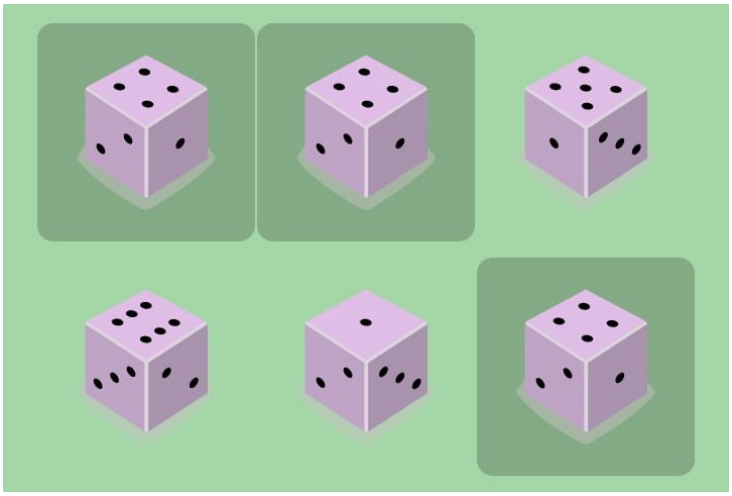


Figure 4.7: Selecting Dice to Hold

Balut Score Function

After a category has been selected in *Balut*, the game also provides an additional button which can be tapped to calculate the score and end the turn. As seen in Figure 4.8.

Figure 4.8: Balut Roll and Score Button

Custom Board Settings

The custom board allows players to set the number of dice that will be used for that specific game. In addition, the board name can also be set, as seen in Figure 4.9. This level of customization gives the users control over how the games are played.

Score Modifiers and Reset

Figure 4.10 shows the score modifiers, which enable users to manually adjust their scores. This feature can also be used to keep track of score in other types of games that might not be covered by this application, or in the custom mode. The feature also contains a reset score button that will reset the score of the game to 0. This can be used to easily restart any game or any other custom use. Additionally, this interface also contains the

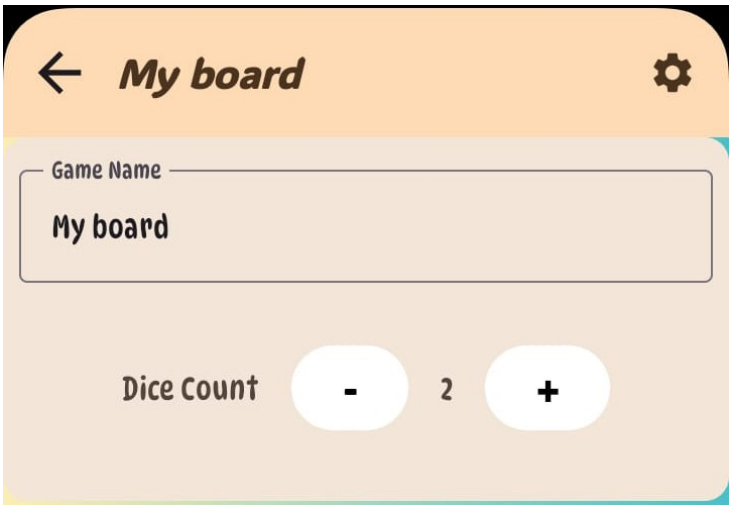


Figure 4.9: Custom Board Settings

functionality of adding a note that will be saved as part of the game, which can be used to keep track of important information of the current game.

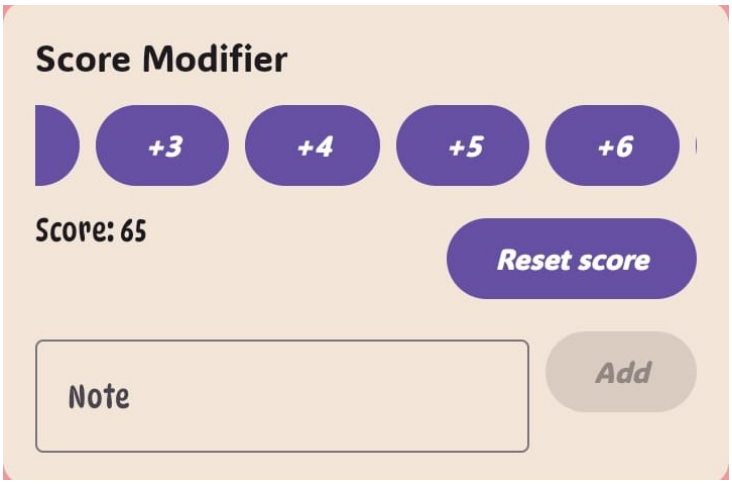


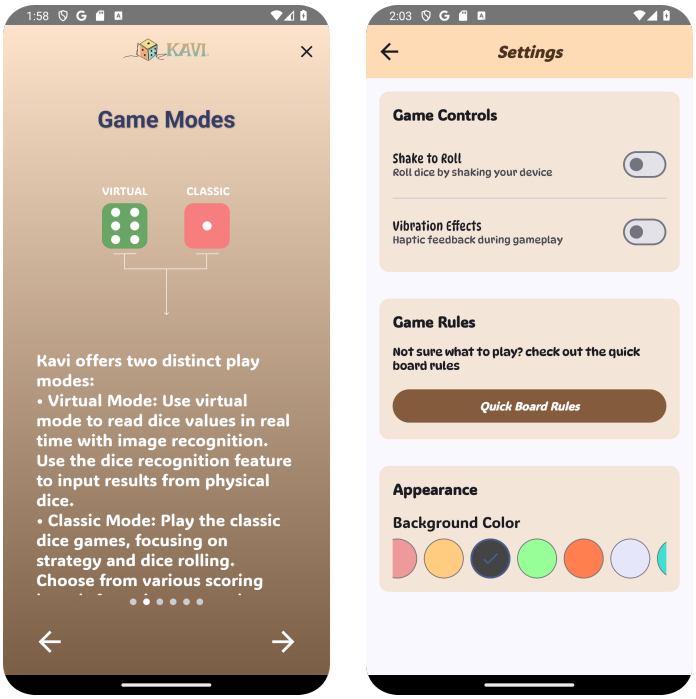
Figure 4.10: Score Modifiers and Reset

4.4.7 Instruction

The instructions screen provides users with detailed guidance on how to play the game. It includes rules, tips, and strategies to enhance the gaming experience. The instructions screen is illustrated in Figure 4.11a.

4.4.8 Settings

The settings screen, on the other hand, allows users to customize their gaming experience, such as enabling vibration, the shake-to-roll function, and board color customization. The settings screen is shown in Figure 4.11b.



(a) Instructions Screen (b) Settings Screen

Figure 4.11: Instructions and Settings.

4.4.9 Statistics

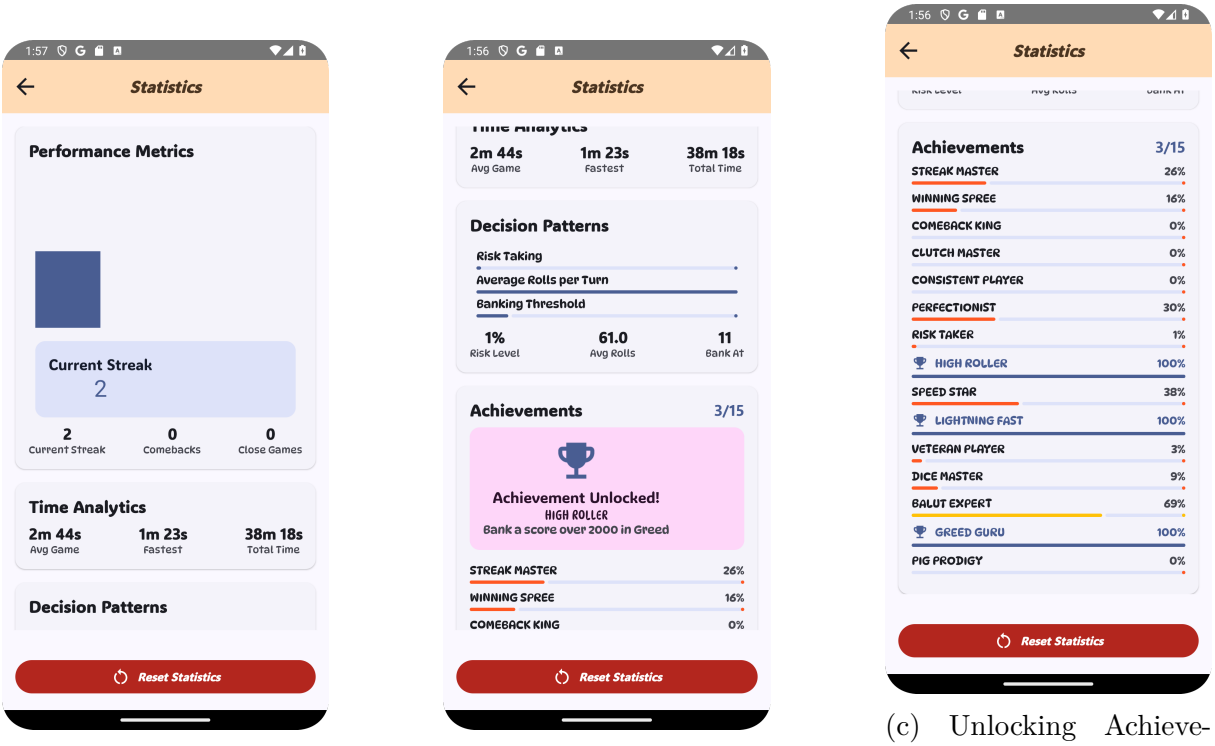
The statistics screen offers users comprehensive insights into their gameplay performance. It displays a variety of data, including win records, average scores, and other pertinent metrics. The Figure 4.12 illustrates the statistics screen, where users can view their achievements, such as "High Roller," "Lightning Fast," and "Greed Guru." Additionally, users can assess their risk-taking tendencies, current winning streaks, comebacks, and close games. The screen also features time analytics, allowing users to track their fastest games, total playtime, and time spent on individual games, providing a detailed overview of their gaming habits.

4.5 System Administration

The administration of the application involves several key tasks to ensure its reliability, performance, and security. These responsibilities are primarily carried out by the developers of the application, and can range from day to day maintenance to handling the occasional unforeseen issues.

4.5.1 Application Maintenance

Regular maintenance of the application is essential to ensure it remains functional and up to date. This involves several tasks, including:



(a) Performance Metrics (b) Decision Patterns (c) Unlocking Achievements

Figure 4.12: Statistics Screen

- **Release Management:** New releases of the application are periodically deployed via GitHub as APK releases, which often require careful planning and testing to ensure compatibility and maintain a consistent user experience. This includes creating new release branches, building the APK, and updating the app with new changes, new features, or bug fixes.

- **Monitoring and Troubleshooting:** Developers are also responsible for monitoring the performance of the application. This may involve checking for crashes, logging error reports, identifying bugs or unexpected behavior, and making changes to solve the issues.

4.5.2 Data Management

Careful management of the application data is critical to ensure its proper functioning. The developers are responsible for *DataStore Management*, which involves taking periodic backups of application settings and user preferences stored with Android DataStore. The data is also cleaned periodically to ensure that the device is not using unnecessary storage. This is essential to maintain consistency and avoid potential data corruption or data loss.

4.6 Security Issues

This section provides a detailed overview of the various security risks that the application may be vulnerable to, based on potential threats and exploitable areas. It also discusses the implemented security measures and planned future improvements to enhance protection. While absolute security is unattainable, the application aims to minimize risks and safeguard user data through best practices and robust security mechanisms.

4.6.1 Data Handling

Application data, including DataStore backups, could be accessed through Android's backup services if a user's Google account is compromised. An attacker may gain unauthorized access if a user falls victim to *phishing* [1], a *data breach*, or other attack vectors.

A compromised Google account could expose backups stored in Google Drive, potentially revealing sensitive user information. To mitigate this risk, the application employs Android's KeyStore to encrypt backup data. While this provides an added layer of security, it does not guarantee complete protection against unauthorized access [26]. Future improvements include implementing user authentication mechanisms and stricter access controls to further enhance data security.

4.6.2 Communication and API Security

Even with HTTPS, data transmitted to the RoboFlow API could be intercepted by attackers, particularly if they are on the same network. Additionally, a malicious server could redirect API requests, exposing sensitive data. While the application enforces HTTPS, it currently does not utilize certificate pinning, making it susceptible to *man-in-the-middle* attacks. Implementing certificate pinning is a planned security enhancement [27].

Rate-limiting mechanisms are in place using a 'CoroutineScope' and a 'MutableStateFlow' to prevent excessive API calls and reduce the risk of application crashes. Input validation is also in place to mitigate *API abuse*. Ongoing efforts focus on further strengthening input validation and implementing more robust rate-limiting techniques.

4.6.3 Code Security

Despite the use of code obfuscation, attackers can still reverse engineer the application to extract sensitive information, such as private API keys, or analyze the game logic to develop cheating mechanisms. Code obfuscation is a technique that makes source code more difficult to read and understand by modifying variable names, class structures, and other elements.

[1] Phishing is a type of cyber attack where attackers impersonate legitimate organizations to trick users into revealing sensitive information, such as passwords or financial details.

Attackers may decompile the application to uncover API keys used for external resources or exploit the game logic to create automated cheating bots.

The application leverages R8's obfuscation process to obscure the code and hinder reverse engineering attempts. However, this is not a foolproof solution, as sophisticated attackers may still deobfuscate [2] and analyze the code [28]. Future enhancements include stronger key management strategies and additional security layers to protect sensitive application logic.

4.7 Security Considerations

The application implements several security measures to protect user data and ensure system integrity. These measures are based on Android security best practices and are designed to comply with relevant data protection requirements.

4.7.1 Data Protection

- **Encrypted Local Storage:** User settings and preferences, saved using DataStore, are stored using Android's EncryptedSharedPreferences, which provides encryption at rest using Android's KeyStore.
- **Privacy-Preserving Image Processing:** Images captured by the user are only processed within the app's scope and are not stored or transmitted outside of the device unless explicitly requested by the user through a sharing action. No personally identifiable information is extracted and saved from the image.

4.7.2 System Security

- **Runtime Permission Management:** The application requests only the necessary permissions at runtime (e.g., camera access) and respects the user's choice to grant or deny them. The application uses Android's Permission API to handle the permissions.
- **Secure Communication Channels:** The communication with the RoboFlow API for image recognition is done over HTTPS, which uses TLS/SSL encryption to ensure confidentiality and integrity of the API requests.
- **Data Access Controls:** Only the application can access the DataStore data, and only the specific components of the application that require it can access its under-

[2]Deobfuscation is the process of removing obfuscation from computer code, making it readable and understandable by humans. Obfuscation deliberately conceals or distorts parts of the code to make the program difficult to detect, tamper with, or reverse engineer.

lying data. This reduces the risk of potential data exposure to malicious application or rogue modules of this application.

4.7.3 Future Enhancements

The following security enhancements are considered for future development:

- **User Authentication and Authorization:** Implementing a secure user authentication mechanism (e.g., using Firebase Authentication) and role-based authorization to enhance data protection and prevent unauthorized access to sensitive features, such as statistics or training data.
- **Improved API Security:** Implementing API rate-limiting to prevent abuse and implementing stricter input validation for the RoboFlow API.
- **Regular Security Audits:** Implementing regular penetration testing to evaluate the security and performance of the system.

4.8 Working scenarios

In addition to the scenarios described in the user manual in Section 4.4, this section provides a few practical examples of application usage. Figure 4.13 shows the application's ability to recognize dice in the environment, along with the corresponding labels indicating the recognized dice classes. During development and testing, the image recognition system was primarily tested using digital dice images displayed on a computer screen, as physical dice were not available. While this approach demonstrated the system's capability to detect and classify dice faces accurately, it represents a controlled testing environment. Figure 4.14 shows the custom game board, where the user can personalize the game settings. Figure 4.3 shows the game boards for the different games. Figure 4.15 shows a few additional features of the application, such as the win, loose and exit game dialogues.

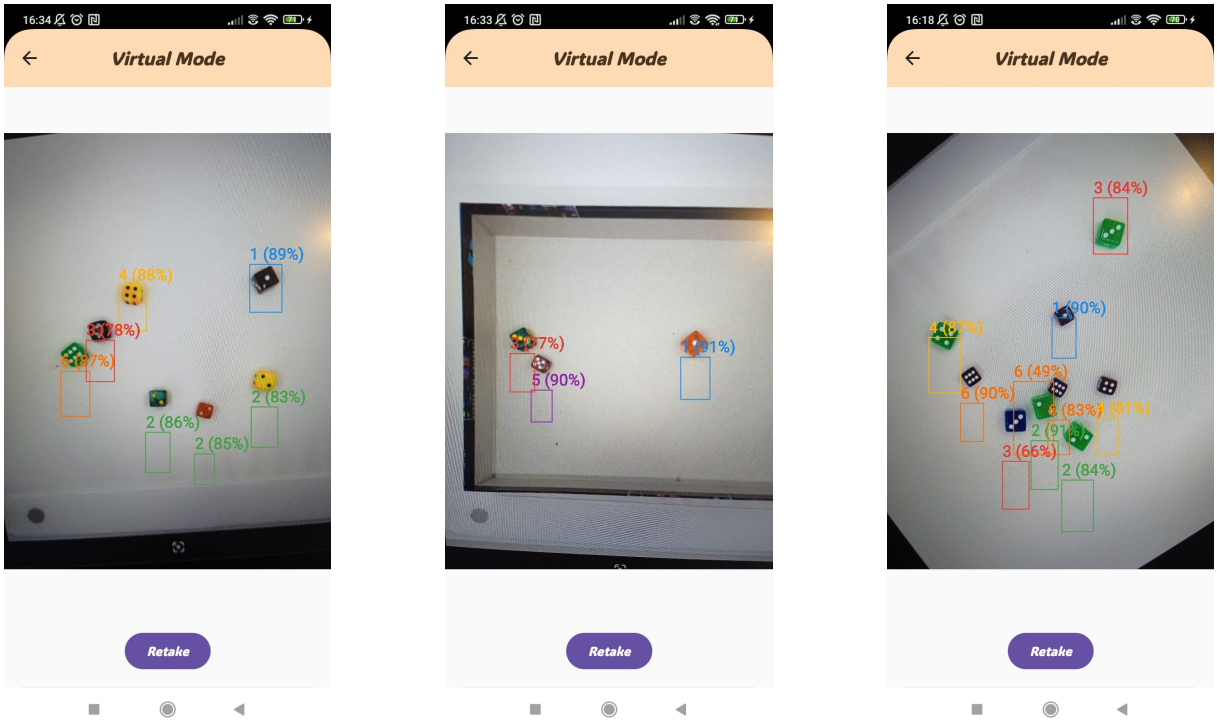


Figure 4.13: Image Recognition of Dice.

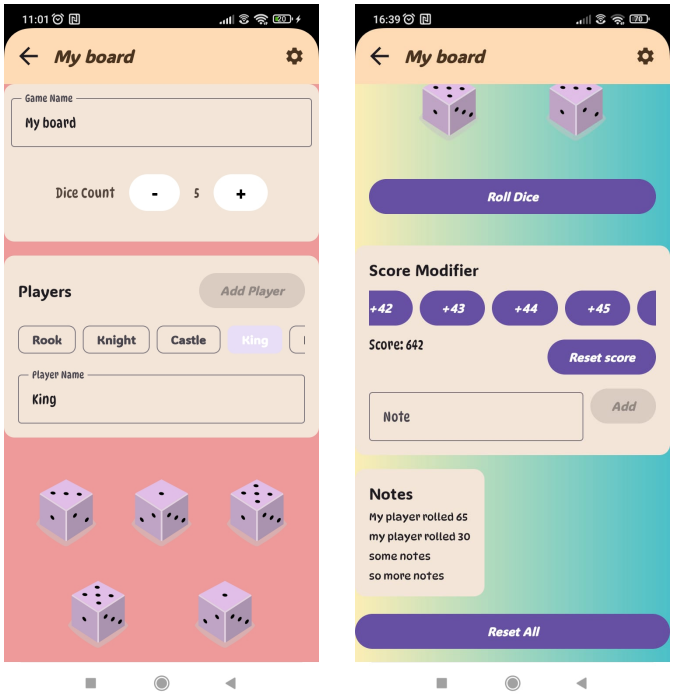
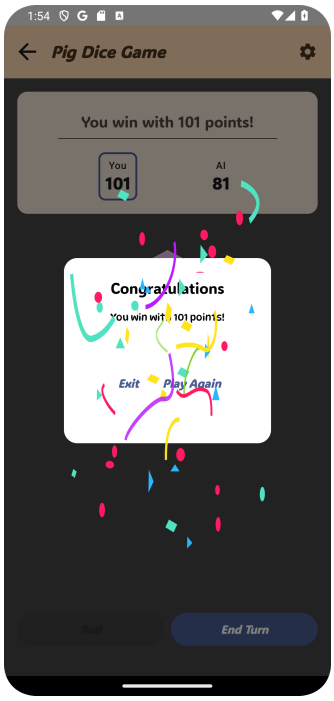
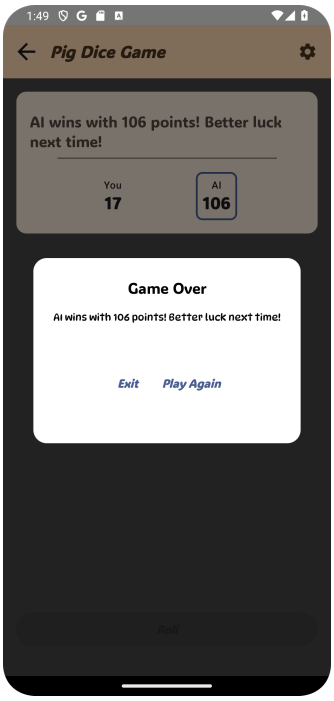


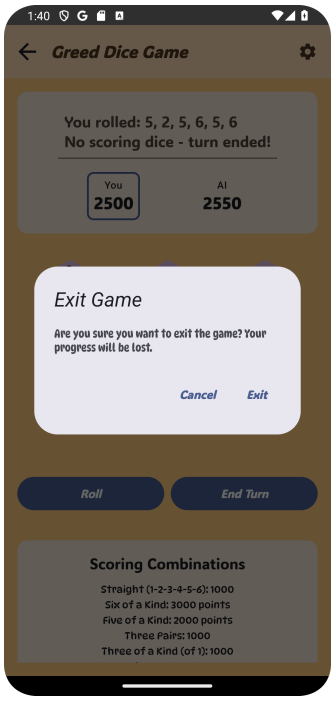
Figure 4.14: Custom Game Board allowing users to personalize their game settings



(a) Winning a Game of Pig with the final score displayed on the board



(b) Losing a Game of Pig, showing the end game scenario



(c) Exiting a game of Greed after finalizing the score

Figure 4.15: Board Features and Game Scenarios

Chapter 5

Internal Specification

This chapter provides a detailed overview of the application's internal workings. It covers the system's concept, architecture, data structures, components, algorithms, design patterns, and relevant UML diagrams.

5.1 System Architecture

The project's architecture follows the modern Model-View-ViewModel (MVVM) pattern, adhering to Clean Architecture principles. This separation of concerns allows for better maintainability and testability of the code.

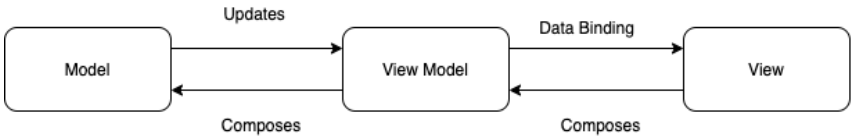


Figure 5.1: High Level MVVM Architecture.

As shown in Figure 5.1, the application is structured into several layers [29]:

- **Application Layer**: Contains the main application logic and navigation.
- **UI Layer**: Responsible for the user interface components, including screens and themes. This layer corresponds to the "View" component of the MVVM pattern.
- **ViewModel Layer**: Manages the data and business logic, providing a bridge between the UI and the Model. This layer corresponds to the "ViewModel" component of the MVVM pattern.
- **Manager Layer**: Handles specific game logic and state management. This layer sits below the view model, and contains the logic for manipulating the models.

- **Repository Layer**: Manages data access and interactions with external sources. This layer is the gatekeeper for the model layer, accessing and manipulating the models before passing it to the application layer.
- **Model Layer**: Represents the data and domain logic of the application. This layer contains the data entities and business rules. This layer corresponds to the "Model" component of the MVVM pattern.

5.2 Use Case Modelling

Use cases describe how users interact with the system, illustrated using UML diagrams that visualize user interactions and the application's features.

Main Menu Use Case

Upon launching the application, users encounter the main menu, which serves as the central hub for interaction. This interface provides access to various game modes, settings, player statistics, and instructional materials, as shown in Figure 5.2.

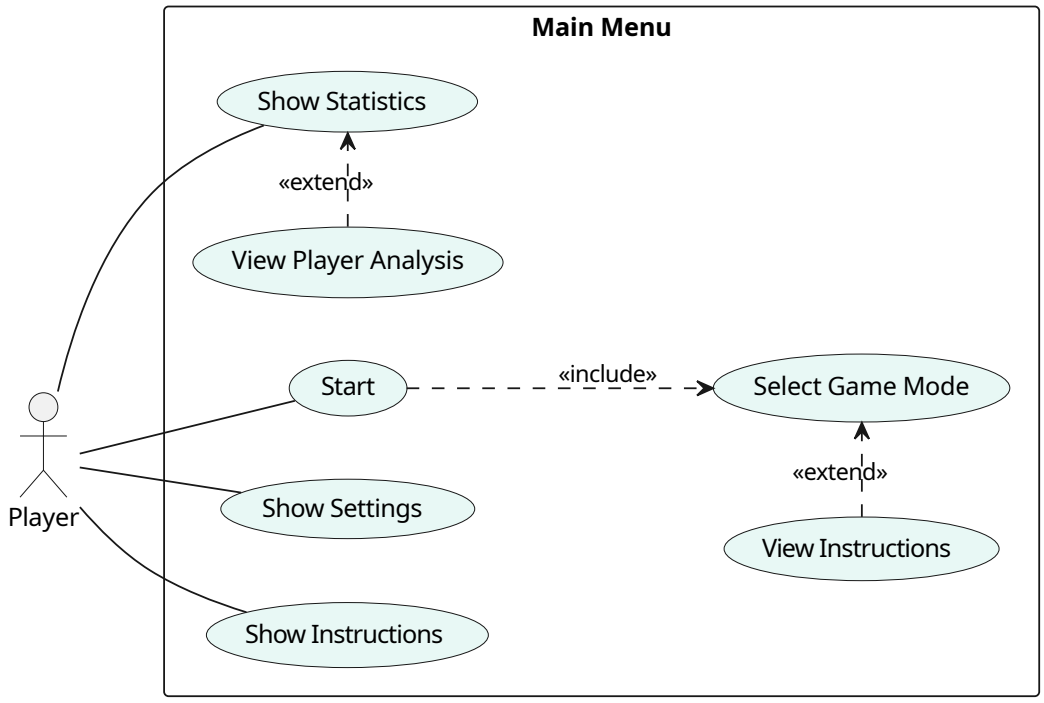


Figure 5.2: Use case diagram for the game's main menu.

Users can start a new game by selecting either the classic or custom game mode, allowing them to engage with predefined rules or configure their own gameplay mechanics. The statistics menu provides insights into player performance, including past game results and analysis. The settings menu enables players to configure parameters such as difficulty level, audio preferences, and visual themes. Additionally, the game rules and instructions

section offers a reference for understanding mechanics and strategies. The application also integrates virtual dice recognition, enabling users to interact with real-world dice using image recognition. Table 5.1 summarizes the key interactions available in the main menu.

Table 5.1: Main Menu Use Case Interactions

Actor	Description
Player	**Start Game**: Initiates gameplay by selecting either classic or custom game modes. **Show Settings**: Provides access to customization options, including difficulty levels and interface preferences. **Show Instructions**: Displays an overview of game mechanics and rules. **Show Statistics**: Enables users to track their performance and game history. **View Player Analysis**: Provides in-depth statistics on past games, allowing players to analyze trends and optimize strategies.

Game Use Case

The game system encapsulates core gameplay mechanics, allowing players to engage in dice-based games in both physical and digital formats. Players roll dice, make strategic decisions, and compete against AI opponents that use heuristic probability to optimize their moves, as shown in Figure 5.3.

Table 5.2: Game Use Case Interactions

Actor	Description
Player	**Play Classic Game**: Engage in traditional dice games with predefined rules. **Play Custom Game**: Define custom rules and gameplay parameters. **Roll Dice**: Roll dice manually or use virtual dice detection. **Hold Dice**: Select specific dice to keep for strategic advantage. **Bank Score**: Secure points to prevent potential losses. **Earn Achievements**: Unlock rewards for reaching milestones. **Use Virtual Mode**: Detect and capture dice roll results using image recognition. **Detect Dice Values**: Process dice images to identify pips and determine values.
AI	**Make Strategic Decisions**: Use heuristic probability to evaluate the game state and optimize moves.

Players can roll dice using the virtual interface or capture real dice rolls via image recognition, enabling seamless integration between physical and digital gameplay. During a turn, players can choose to hold specific dice to refine their strategy or bank points to

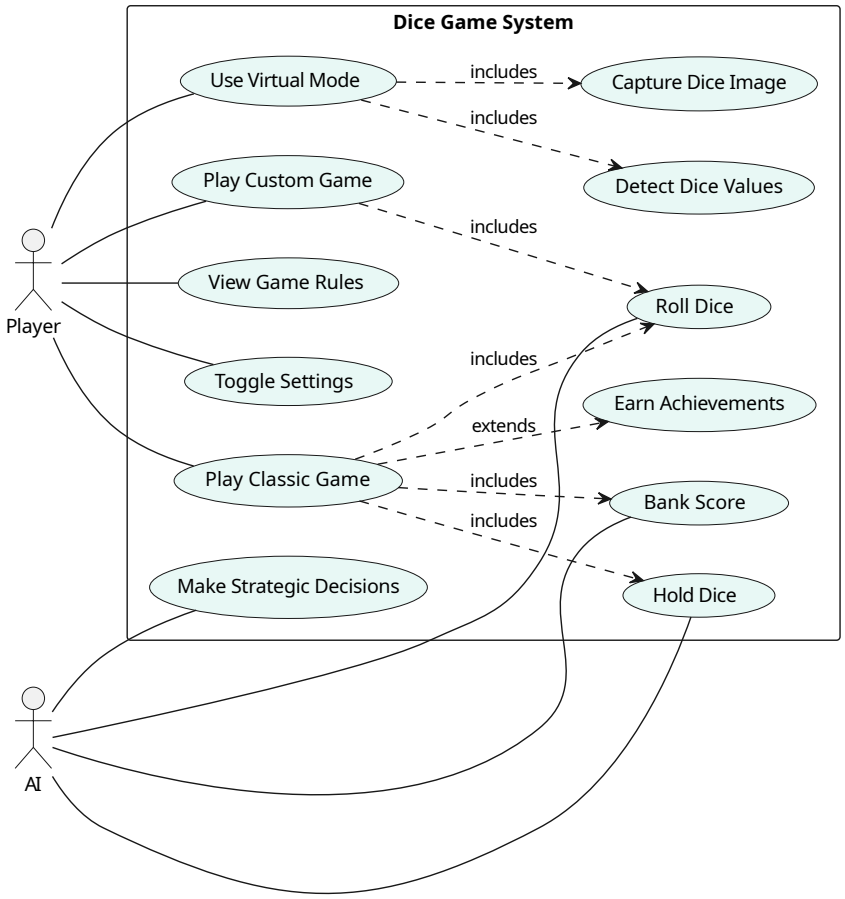


Figure 5.3: Use case diagram for the game's core gameplay.

secure their score. AI opponents analyze game conditions using heuristic probability to make strategic decisions, adding challenge and unpredictability to the gameplay. Table 5.2 outlines the primary interactions within the game system.

5.3 Methodology of Design and Implementation

The design and implementation of the dice game application follow an iterative and incremental development methodology. This approach involves:

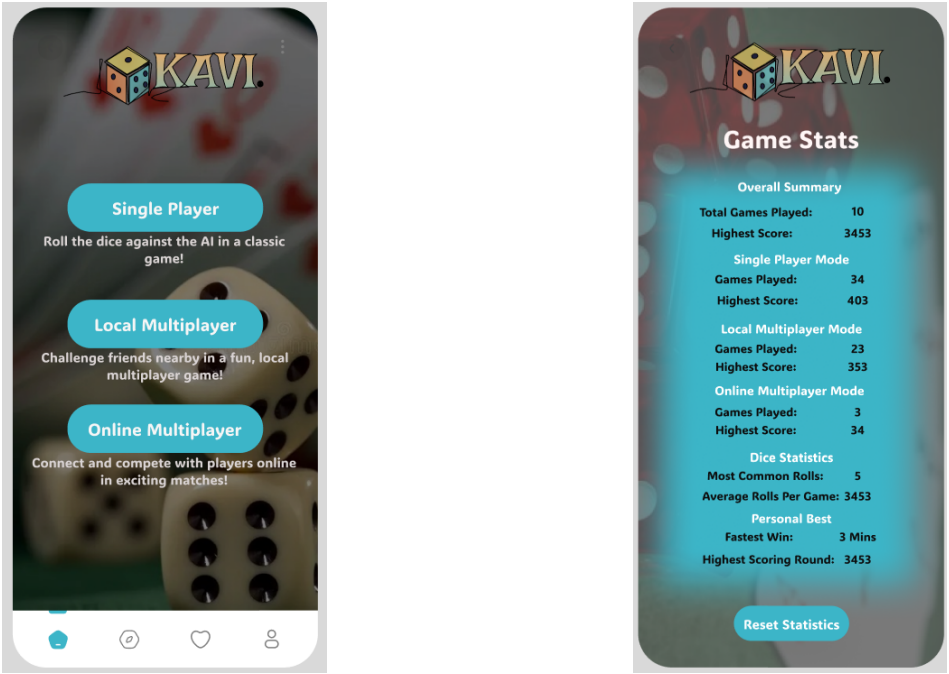
1. **Requirement**: Identifying and documenting functional and non-functional requirements.

2. **Design**: Creating architectural and component designs, including UML diagrams to visualize system interactions.

3. **Implementation**: Developing the application in iterative cycles, focusing on one feature or component at a time.

4. **Testing**: Conducting unit, integration, and user acceptance testing to ensure the system meets requirements using JUnit.

5. **Documentation**: Documentation of the implemented features and future development possibilities.



(a) Design Page 1 (b) Design page 2

Figure 5.4: Initial UI designs and prototypes created in Figma

5.3.1 Design Process

The application's design process began with creating detailed wireframes and prototypes in Figma. The designs underwent several iterations based on user feedback and technical constraints, evolving into the final implementation. Figure 5.4 shows some of the initial design concepts and their evolution [30].

Various existing solutions and design tools inspired the design of the application, one of which stood out was the board screen design was inspired by a dice application project by Shreyansh Saurabh [31]. This repository provided a minimalistic and intuitive approach to dice roll applications, which influenced the layout and functionality of the board screen in this project.

5.3.2 Model Training

The dice detection model was developed using Roboflow's platform, which streamlined the entire process from dataset creation to deployment. The training dataset, shown in Figure 5.5, consisted of carefully annotated dice images across various conditions, ensuring robust detection performance in real-world scenarios.

The training process involved several key stages:

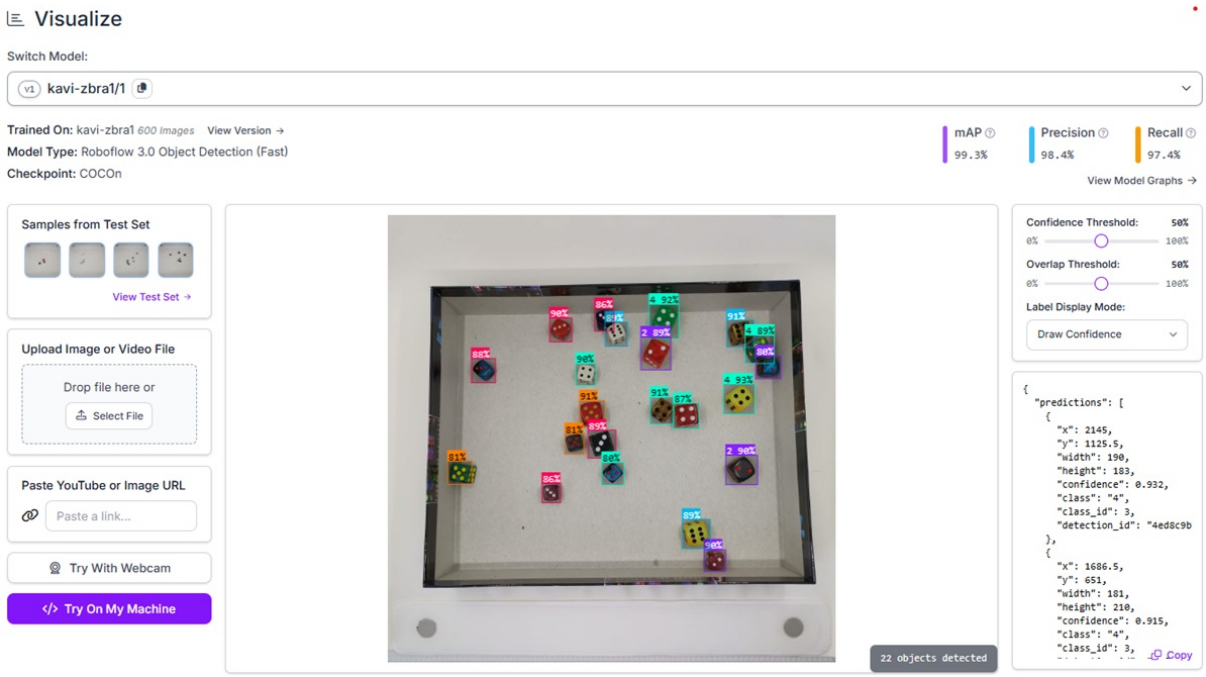


Figure 5.5: Roboflow dataset management interface showing dice image annotations

1. **Dataset Preparation**:
   - Collection of 250 dice images under various lighting conditions
   - Manual annotation of dice faces using bounding boxes
   - Dataset split: 80% training, 10% validation, 10% testing
2. **Data Augmentation**:
   - Outputs: 3 augmented versions per training example
   - Flip: Horizontal mirroring
   - Rotation: 90° clockwise and counter-clockwise, plus random rotations between -15° and +15°
   - Crop: Random zoom between 0% and 2%
   - Exposure: Adjustments between -10% and +10%
3. **Model Configuration**:
   - Architecture: YOLOv8n model
   - Input resolution: 640x640 pixels
   - Classes: 6 (representing dice faces 1-6)

The model achieved satisfactory performance metrics on the validation set, demonstrating reliable detection capabilities across different lighting conditions and angles. The

trained model was then deployed through Roboflow's Hosted API service, enabling real-time inference with consistent response times suitable for mobile applications.

The deployment process included model optimization techniques such as quantization and pruning to ensure efficient mobile performance while maintaining accuracy. This resulted in a model that could effectively detect and classify dice faces in various real-world conditions while meeting the performance requirements of a mobile application.

5.3.3 Project Timeline

The project was implemented following a structured timeline as shown in Figure 5.6. The development process was organized into major phases, including planning, design, core development, AI integration, and testing, with regular milestones to track progress.

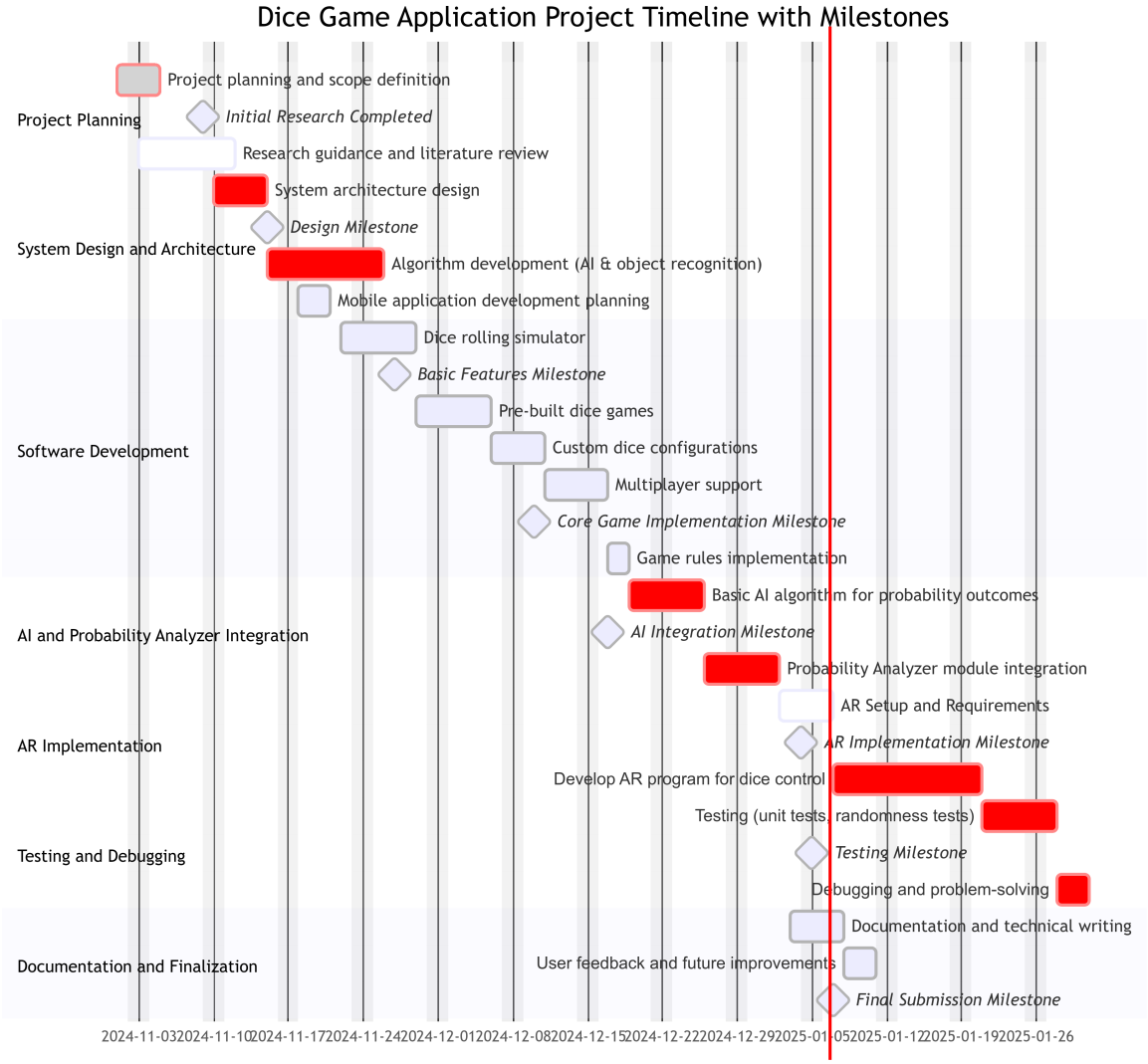


Figure 5.6: Project Gantt chart showing development phases and milestones

This structured approach allowed for continuous improvement and adaptation to changing requirements, ensuring a high-quality application. While some initially planned fea-

tures like AR implementation were identified as future enhancements, the focus remained on delivering a robust core game experience with AI capabilities.

5.4 Data Structures and Data Management

The application utilizes a variety of data structures to effectively manage game state, user profiles, and game data. These structures are designed for efficiency and scalability, supporting the diverse features of the game. Data persistence is achieved using DataStore, a modern data storage solution, and dependency injection is managed using Dagger Hilt.

5.4.1 Data Models

The application's data is represented using models located in the 'models' directory. These models encompass different aspects of the game, including game state, user data, and results from the image detection mechanisms.

Game Models

The core game models are:

- **GameStatistics**: Tracks overall game performance metrics for each user across all games. These metrics include total games played, win rates, average scores, and other relevant data.
- **GameScoreState**: Holds the current score and state of the game during an individual game session, including the score for each player and the game's current turn.
- **DecisionPatterns**: Captures the different decision-making patterns of a player in each game, including when the player decides to hold dice, or when they decide to roll or bank scores. This is used to calculate statistics about how players are playing the game.
- **WinRate**: Calculates and stores the win rate of a player. This will be used in *PlayerAnalysis*.
- **TimeMetrics**: Stores metrics about how long a player played the game, including the amount of time a player has spent playing each game, and the time spent per round. This is used in *PlayerAnalysis*.
- **PlayerAnalysis**: Provides detailed analysis of a player's behavior and performance trends, combining data from various sources like 'WinRate', 'TimeMetrics', and

'DecisionPatterns'. It tracks trends over time, identify strong and weak areas of a player, and provides an overall evaluation of a player's performance.

Detection Models

The detection models, located in the 'detection' subdirectory of 'models', are:

- **DiceDetectionResult**: Captures results from dice detection processes, including the number detected and the number on each die.
- **Detection**: Represents the state of the detection model.
- **DetectionRequest**: Represents the request to start the detection.
- **DetectionResponse**: Represents the response from the detection.
- **ImageInfo**: Holds the information of the image used for detection.
- **Prediction**: Represents a single prediction from the detection model.

5.4.2 Data Management

The application uses a modern and efficient approach to data management, leveraging DataStore for persistent storage and Dagger Hilt for dependency injection.

Persistent Storage

DataStore provides a robust, asynchronous solution for managing the application's persistent data. Unlike traditional databases, DataStore offers type safety and a reactive approach, ensuring smooth and efficient data handling for user preferences, game statistics, and other relevant application data. This approach helps the application to provide quick access to stored data and avoids some of the limitations of other persistent storage options.

Dependency Injection

Dagger Hilt simplifies the management of dependencies by providing a standardized way to inject components into the application. This improves code modularity, making the app easier to test, maintain, and scale. Dagger Hilt helps manage the dependencies between classes and is used to ensure all modules are configured correctly, and this creates a more efficient way to manage the dependencies of the application, while also making testing easier.

5.5 Components, Modules, and Classes

This section outlines the core components, modules, and classes that form the foundation of the application. It provides a summary of essential classes, detailing their roles and responsibilities.

The main classes of the application can be categorized into Main Application Classes, ViewModels, and Managers.

Main Application Classes

- **KaviApplication**: The main class that extends Android's Application class, responsible for initializing the application, and important libraries like Timber for debugging.
- **MainActivity**: The main entry point for the application, sets up the primary UI, and manages navigation.

ViewModels

- **AppViewModel**: Manages application-wide data and state, coordinating between different parts of the application.
- **GameViewModel**: Manages data and logic specific to each game, providing a bridge between the view and the data and state.
- **DetectionViewModel**: Manages the state and logic of the image detection process, and exposes this data to the UI layer.

Managers

- **MyGameManager**: Handles the state management and the logic of the custom game board.
- **PigGameManager**: Manages the game state, rules, and scoring for the Pig game.
- **GreedGameManager**: Manages the game state, rules, and scoring for the Greed game.
- **BalutGameManager**: Manages the game state, rules, and scoring for the Balut game.
- **DiceManager**: Manages the various aspects of rolling the dice, and its state.
- **DataStoreManager**: Manages the saving and retrieval of data from the DataStore.
- **StatisticsManager**: Manages the collection and processing of game statistics.

- **SettingsManager**: Manages the saving and loading of the application's settings.
- **ShakeDetectorManager**: Manages the logic of the shake gesture.

A detailed Class diagram is included in the UML diagram provided in Section 5.8.

5.6 Algorithms and Implementations

The application implements several sophisticated algorithms to provide an engaging and intelligent gaming experience.

5.6.1 Image Processing Pipeline

The dice recognition system uses a sequence of image processing steps implemented in 'RoboflowRepositoryImpl', a class responsible for handling communication with the Roboflow API:

Preprocessing

- RGB conversion using 'ensureRGBFormat()' to guarantee that the image is in the correct format for processing.
- Contrast enhancement through 'enhanceContrast()' using histogram-based normalization to make the dice pips more clear.
- Aspect ratio scaling via 'scaleWithAspectRatio()' to make sure that the images are of the correct size.
- Noise reduction with 'reduceNoise()' to reduce the noise in the image.

Detection

- API integration with Roboflow service using the 'RoboflowClient', which makes a call to the Roboflow API and gets the result.
- Confidence filtering (threshold: 40%) which removes detections with less than 40% confidence to avoid erroneous detections.
- Non-maximum suppression, performed by using a library for detection, which removes overlapping bounding boxes, by selecting the bounding boxes with the highest score and removing those that are overlapping.

5.6.2 AI Strategy System

The AI decision-making system, which is responsible for determining the optimal moves for the AI, is implemented across multiple manager classes. The strategy is designed to consider the current game state and make intelligent decisions based on the game variant. Below are the key methods used by the AI to guide its decisions:

- *'shouldAIBank()'* is responsible for determining when the AI should bank points based on the current turn's score, its total score, and the player's total score. The decision is dependent on the game variant being played, such as the "Greed" or "Pig" game mode. If the AI is playing a "Greed" or "Pig" game, it assesses whether continuing to roll is risky based on the points accumulated so far. If the risk outweighs the potential gain, the AI decides to bank.

```
1   fun shouldAIBank(currentTurnScore: Int, aiTotalScore: Int, playerTotalScore:
    Int): Boolean =
2       when (_selectedBoard.value) {
3           GameBoard.PIG.modeName -> pigGameManager.shouldAIBank(
4               currentTurnScore = currentTurnScore,
5               aiTotalScore = aiTotalScore,
6               playerTotalScore = playerTotalScore
7           )
8           GameBoard.GREED.modeName -> greedGameManager.shouldAIBank(
9               currentTurnScore = currentTurnScore,
10              aiTotalScore = aiTotalScore
11          )
12          else -> false
13      }
```

Listing 5.1: Should AI Bank Function.

- *'chooseAICategory()'* is used to select the optimal scoring categories for the Balut AI based on the dice rolled. The AI evaluates the current dice and determines which scoring category will maximize its chances of winning, such as the best combination of dice values for the current state of the game. If the game state is invalid or not initialized, it will first initialize the Balut game before making a decision.

```
1   fun chooseAICategory(diceResults: List<Int>): String {
2       val currentState = (_gameState.value as? GameScoreState.BalutScoreState) ?:
     run {
3           Timber.e("Invalid game state for AI category selection")
4           return balutGameManager.initializeGame().let {
5               balutGameManager.chooseAICategory(diceResults, it)
6           }
7       }
8       return balutGameManager.chooseAICategory(diceResults, currentState)
9   }
10
```

Listing 5.2: Choose AI Category Function.

These methods ensure that the AI can make well-informed decisions based on its current state and the game rules, allowing for a challenging and adaptive opponent in each game variant.

5.6.3 Statistics System

The statistics tracking system, which is used for analyzing and processing user data, is centralized in ‘StatisticsManager‘:

Game Analytics

- *‘updateGameStatistics()‘* records game outcomes for the players, and saves it to the ‘GameStatistics‘ model.
- *‘updateTimeMetrics()‘* tracks timing data, including the time spent in each round, and the total time spent in a game.
- *‘updatePerformanceMetrics()‘* calculates improvement rates, by keeping track of how many games the player wins, and their high scores.

Achievement Processing

The achievement processing is responsible for monitoring and managing user progress toward unlocking achievements in the application. The following key functions are implemented:

- *‘calculateAchievements()‘*: Evaluates the unlock conditions for all achievements available in the application and updates their status based on the latest user activity and metrics.
- *‘updateProgressMetrics()‘*: Tracks progress toward achievement goals by calculating how close the user is to meeting the requirements for unlocking various achievements.

5.7 Applied Design Patterns

This application uses several design patterns to enhance code organization and maintainability:

- **Observer Pattern**: Used in the ViewModel layer with Kotlin Flow (StateFlow) to manage and notify the UI of data changes, ensuring a reactive user interface. This allows the UI to update automatically when the state of the application changes. Examples include dice state updates, game statistics updates, and detection results.

- **Singleton Pattern**: Employed for managing shared resources using Dagger Hilt's dependency injection. This avoids the need to manually create singletons. Key singletons include the 'StatisticsManager' for collecting and managing game analytics, the 'GameTracker' for monitoring gameplay sessions, and the 'DataStoreManager' for providing access to persistent storage.
- **Factory Pattern**: Implemented using Dagger Hilt's module system to dynamically provide 'GameManager' instances (such as 'PigGameManager', 'GreedGameManager', 'BalutGameManager') based on the selected game mode. This approach abstracts object creation by separating the responsibility of object creation from the main code logic and creating the instances of the game managers on the go, without having to specify which instance to create, promoting flexibility and scalability in managing different game variants.

5.8 UML Diagrams

This section presents the UML diagrams that illustrates the architecture and dynamic interactions within the application.

5.8.1 Class Diagram

The class diagram (Figure 5.7) provides a visual representation of the static structure of the application, outlining the main classes, their attributes, methods, and how they interact with one another. Key elements shown in this diagram include game states, player analysis, performance metrics, and detection-related classes that capture the core logic and data flow within the app. For instance, the `GameState` class tracks the game's progress, while the `DiceDetectionResult` class manages dice recognition results.

5.8.2 Models Diagram

Figure 5.8 presents the models diagram, which illustrates the data structure of the application. It shows the relationships between various data models, such as `GameState`, `GameStatistics`, and `Detection`. This diagram helps to understand how data is stored and manipulated across different components of the application. For example, the `Detection` class stores information about recognized objects in the dice, while the `GameStatistics` class tracks overall game data.

5.8.3 Structure Diagram

The structure diagram (Figure 5.9) outlines the modular organization of the application's packages, highlighting their dependencies and logical groupings. This diagram

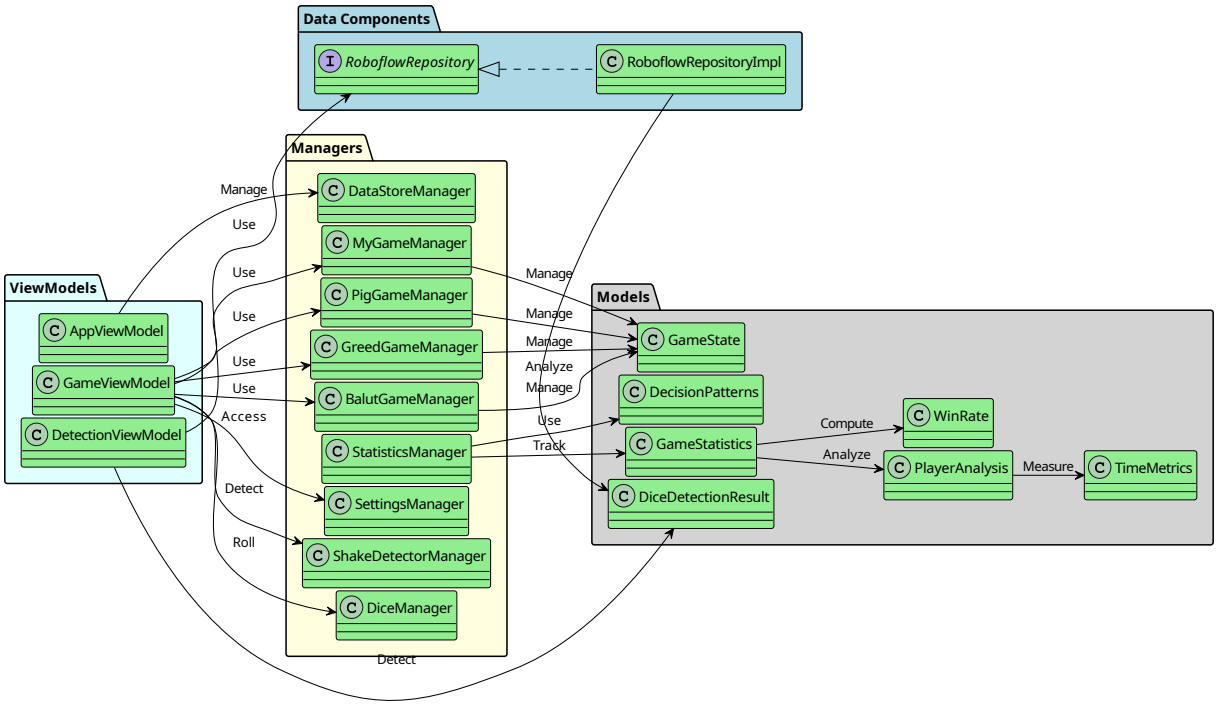


Figure 5.7: Class Diagram of the Application, showing key classes and their relationships.

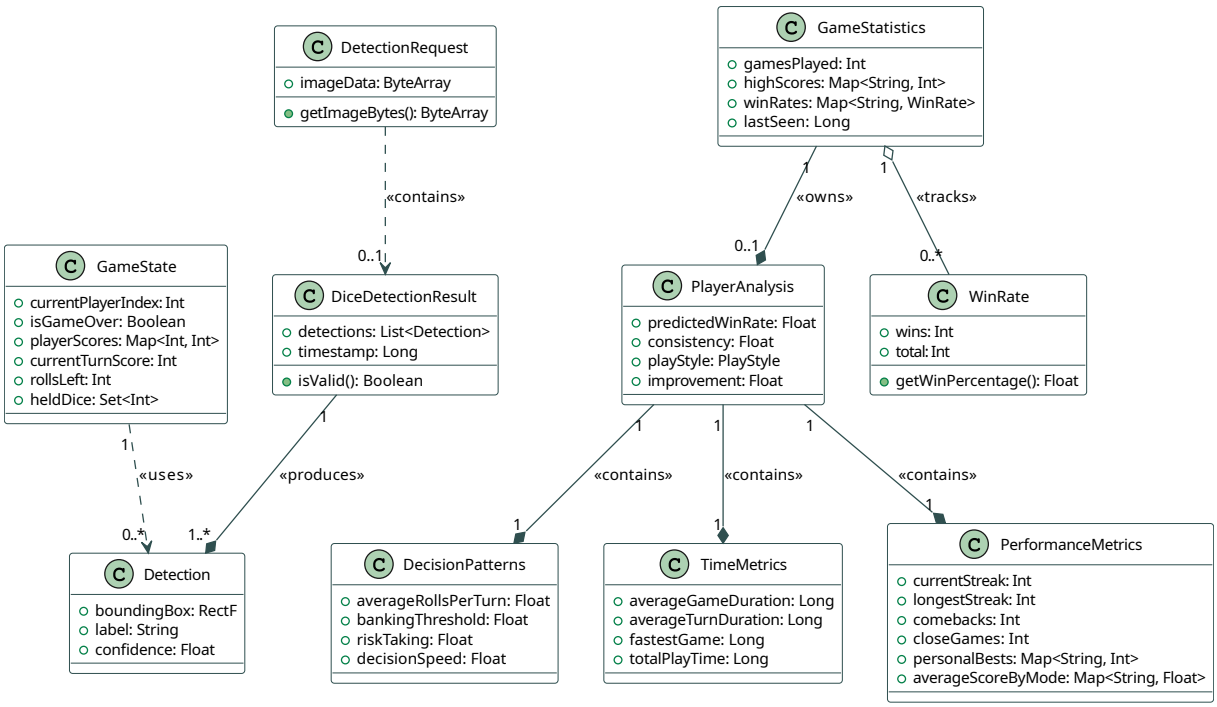


Figure 5.8: Models Diagram of the Application, showing the relationships between data models.

provides an overview of the application architecture, which is split into layers like the data layer, UI layer, and utility classes. For example, the `data` package encompasses models, managers, and repositories, while the `ui` package contains screens, components, and view models. This diagram helps to visualize the overall design and dependencies

between different components of the app.

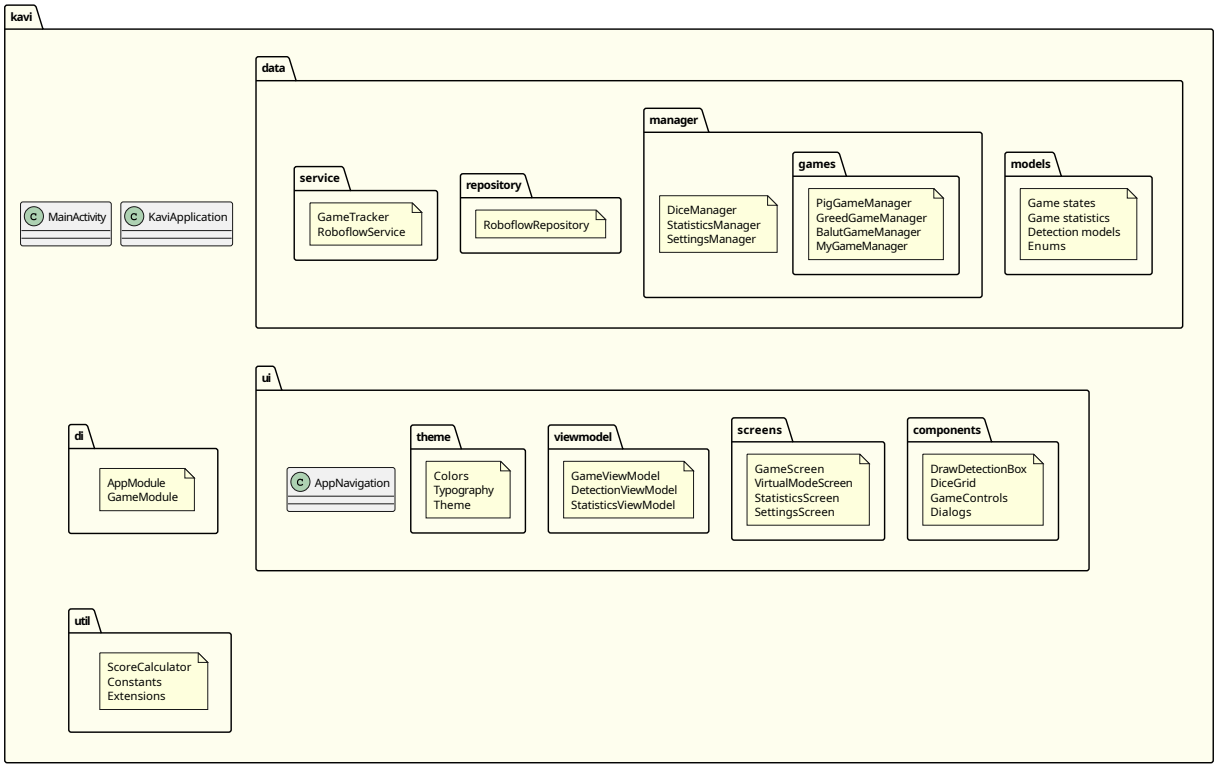


Figure 5.9: Package Structure of the Application, showing the organization of components and their dependencies.

5.9 Sequence Diagrams

Sequence diagrams are used to show the dynamic interactions between various components and objects within the game. They provide a clear visualization of the message flow and the sequence of operations in key scenarios. In this section, we present sequence diagrams that depict the game flow, virtual mode flow, and analysis flow within the application.

5.9.1 Game Flow Sequence

The game flow sequence shows how different components interact during a typical game session. Figure 5.10 shows the process starting from the player initiating a game to the end of the game.

The game starts when the player initiates it through the ‘GameScreen‘. The ‘GameScreen‘ initializes the ‘GameViewModel‘, which starts the game by calling the ‘GameManager‘ to set up the initial game state. Simultaneously, the ‘StatisticsManager‘ starts tracking game time, and the ‘ShakeDetectionManager‘ is activated to detect shake gestures for rolling dice. During gameplay, the player can roll the dice either by shaking the device or

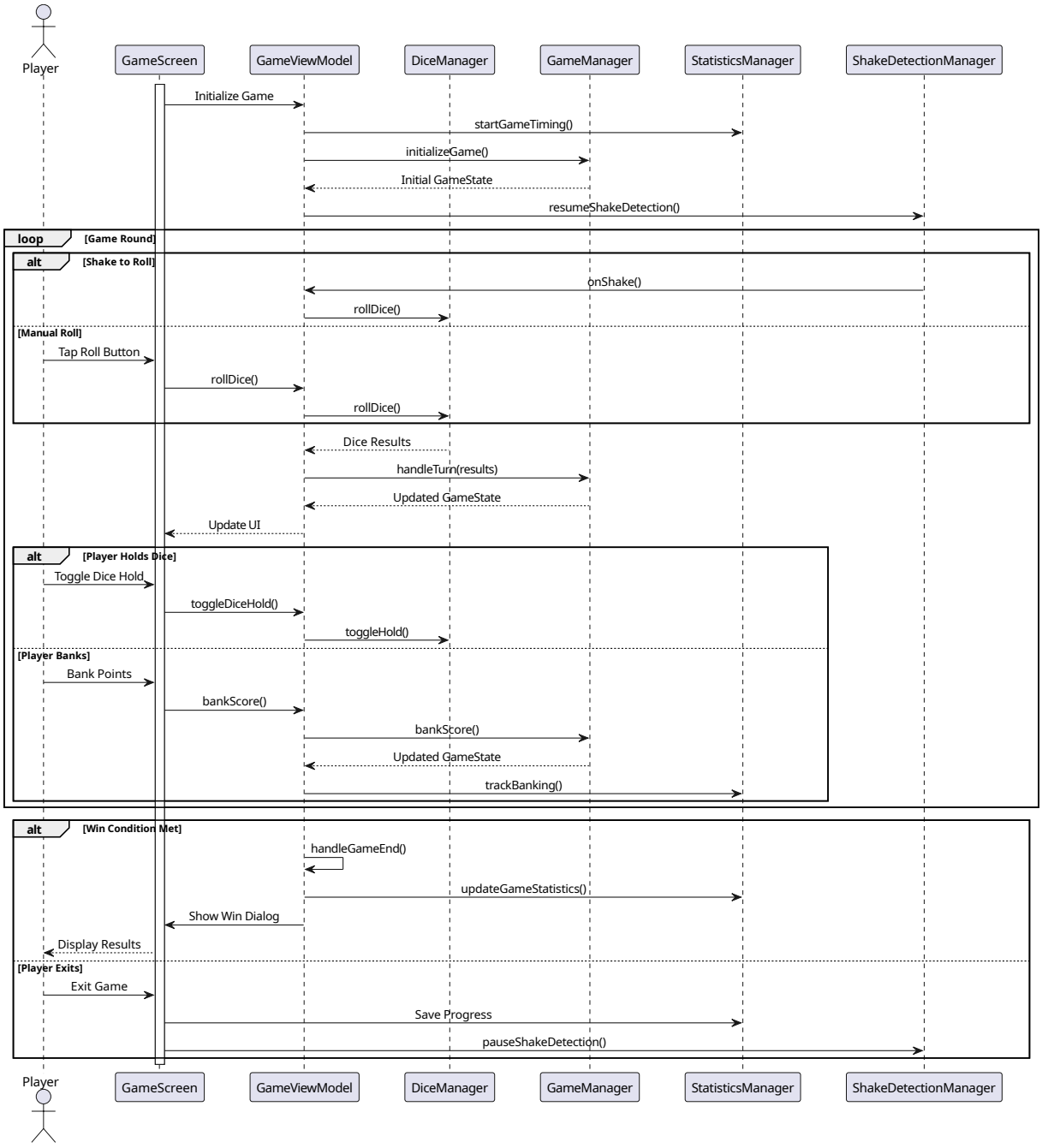


Figure 5.10: Game Flow Sequence in the Application

manually tapping the roll button. These actions are handled by the ‘GameViewModel‘, which instructs the ‘DiceManager‘ to roll the dice and updates the game state through the ‘GameManager‘. The player can also toggle dice holds or bank their score, triggering updates in the ‘GameManager‘ and ‘StatisticsManager‘ as needed. The game ends when a win or loose condition is met, prompting the ‘GameViewModel‘ to update statistics and display the results to the player. Alternatively, if the player exits before completing the game, progress is saved, and the shake detection is paused.

5.9.2 Virtual Mode Sequence

The virtual mode sequence, shown in Figure 5.11, demonstrates how the application manages the image capture and dice detection within the virtual mode.

In the virtual mode, the ‘VirtualModeScreen‘ initializes a camera preview using a ‘CameraPreview‘ class. The player then taps the capture button, which initiates the image capture process. The captured image is passed to the ‘DetectionViewModel‘, which sets its internal state to processing. The ‘DetectionViewModel‘ then calls the ‘RoboflowRepository‘ which handles all the detection requests, by preprocessing the images, calling the ‘RoboflowService‘, and mapping the result to the ‘DetectionViewModel‘. The ‘DetectionViewModel‘ sets its internal state based on the received information, and updates the ‘VirtualModeScreen‘ to draw bounding boxes on the screen. The process is repeated if the player retakes the image.

5.9.3 Analytics Flow Sequence

The analytics flow, shown in Figure 5.12, illustrates the steps involved in retrieving and displaying user statistics and analytics.

When the player opens the ‘StatisticsScreen‘, the ‘AnalyticsDashboard‘ is initialized. The ‘AnalyticsDashboard‘ then collects the ‘gameStatistics‘ from the ‘StatisticsManager‘, which retrieves the stored statistics from the ‘DataStore‘. The ‘StatisticsManager‘ calculates various metrics, such as win rates, play style, performance metrics, achievements, decision patterns, and time metrics. Afterward, it updates the ‘AnalyticsDashboard‘ with the latest information. The player can view detailed metrics on the ‘AnalyticsDashboard‘, including risk-taking levels, average rolls per turn, banking thresholds, and achievement progress. If the player chooses to clear the statistics, the ‘StatisticsManager‘ will use the ‘DataStore‘ to reset the stored data and clear the local state.

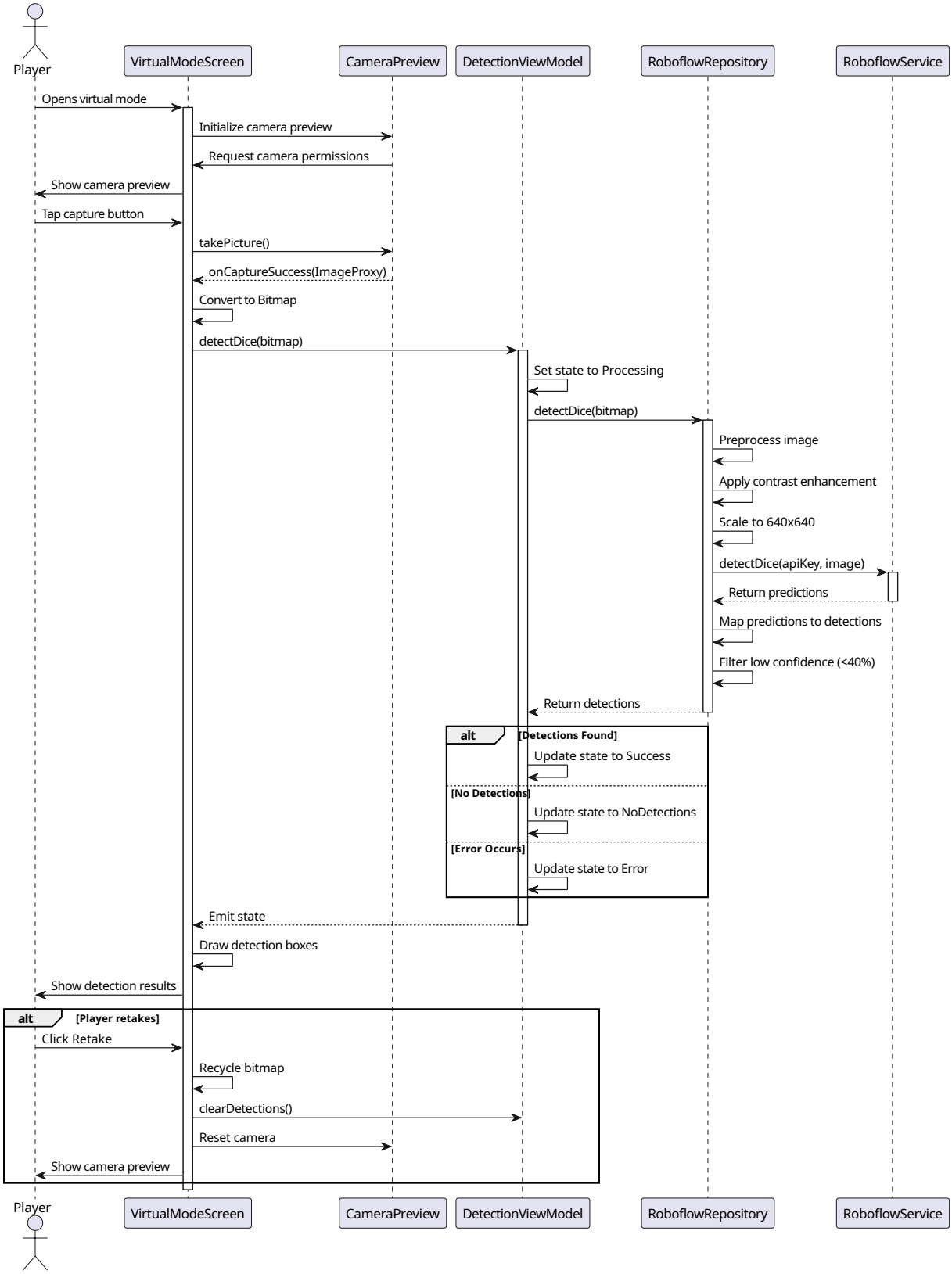


Figure 5.11: Virtual mode Sequence in the Application

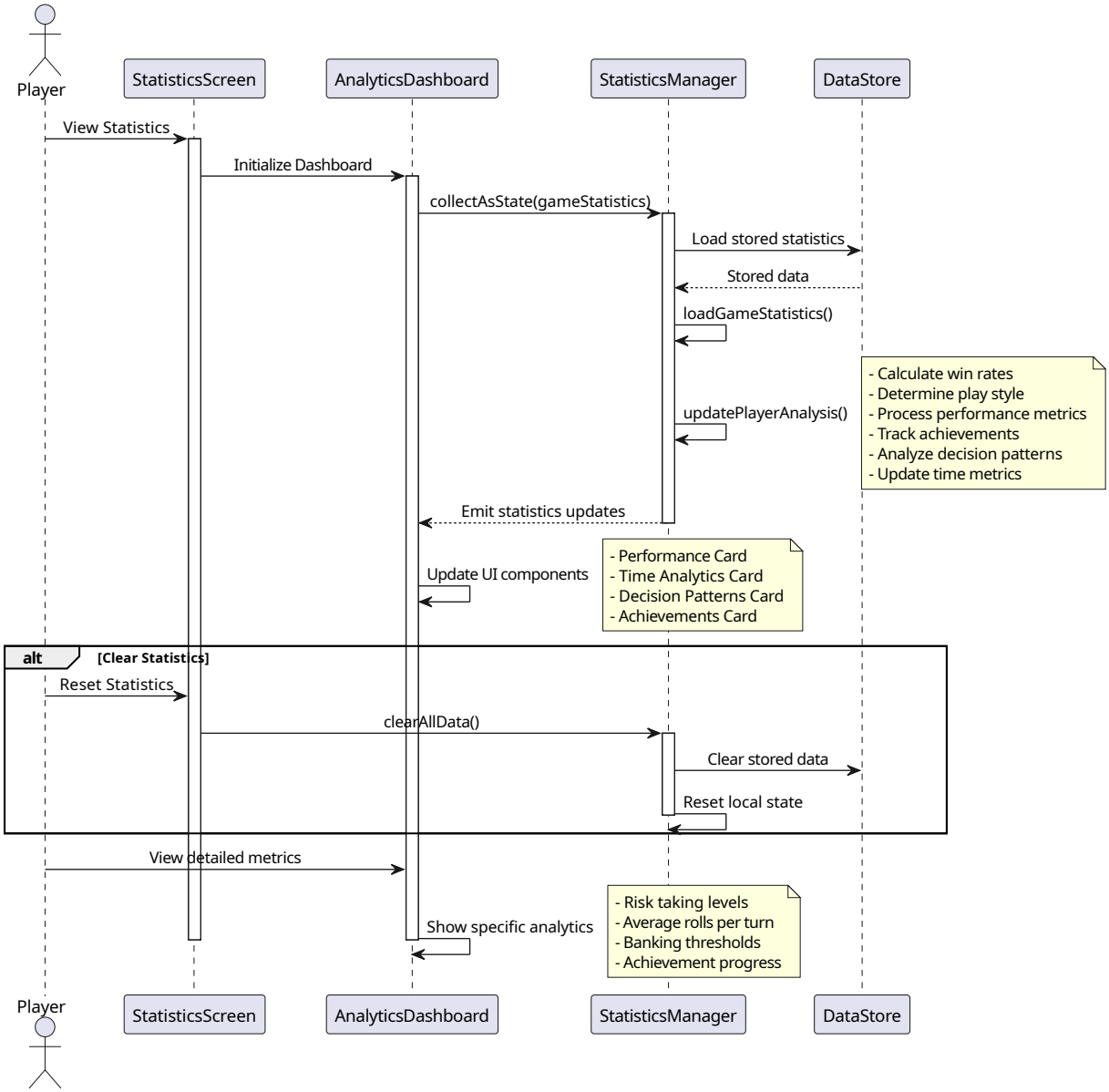


Figure 5.12: Analysis Flow Sequence in the Application

Chapter 6

Verification and Validation

Ensuring the reliability, functionality, and usability of software applications necessitates rigorous verification and validation processes. Verification confirms that the application is built according to design specifications, while validation ensures it fulfils user needs.

This chapter provides an overview of our quality assurance methods during application development. Specifically, it outlines the testing paradigm, test case designs, testing scope, detected and resolved bugs, and experimental results.

6.1 Testing

The Table 6.1 summarizes the different testing types, their purposes, and scopes for this application.

Table 6.1: Summary of Testing Types and Scope

Test Type	**Purpose**	**Scope**
Unit Testing	Validate individual components or functions.	Core game logic, utility functions.
Integration Testing	Ensure correct interaction between modules.	Game logic and UI, and image recognition.
System Testing	Verify the complete application works as intended.	End-to-end gameplay and Image recognition.
Regression Testing	Identify defects after changes to the codebase.	Post-fix validation of all modules.
Performance Testing	Measure responsiveness and stability under load.	Frame rate, AI performance with multiple players.

The V-Model testing paradigm guided the verification and validation of this application. A structured approach, particularly suitable where high-quality assurance is critical [32], the V-Model expands on the traditional Waterfall Model by emphasizing a parallel relationship between development and testing phases (Figure 6.1).

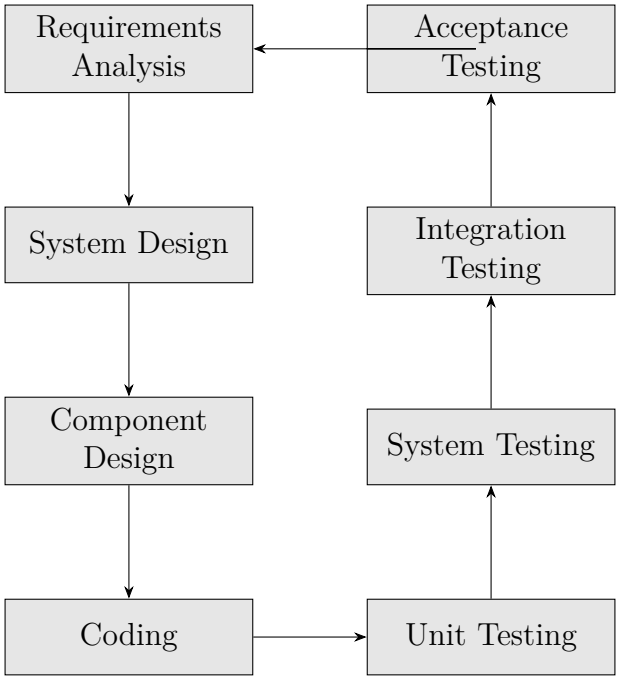


Figure 6.1: Simplified V-Model Diagram

Each development activity was paired with a corresponding testing phase to ensure systematic validation at every stage; for example, requirements analysis was linked to acceptance testing. The V-Model's clarity and emphasis on accountability made it an ideal paradigm for this project, with its aim to deliver a reliable, user-friendly application.

The V-Model incorporates two fundamental types of testing:

- **Verification**: Involves activities like reviews, inspections, and static testing to confirm the application meets specified requirements. For example, unit tests confirm that individual components behave as designed.

```
1    @Test
2    fun `test game initialization`() {
3        val gameState = balutGameManager.initializeGame()
4        // Verify initial state
5        assertEquals(2, gameState.playerScores.size)
6        assertTrue(gameState.playerScores[0]?.isEmpty() == true)
7        assertEquals(1, gameState.currentRound)
8        assertFalse(gameState.isGameOver)
9    }
10
```

Listing 6.1: Unit Test for Game Initialization

- **Validation**: Consists of dynamic testing where the final product is evaluated to confirm it meets user requirements and needs. For example, integration tests are done to confirm correct interactions between software modules.

```
    @Test
    fun `detectDice should update state correctly`() = runTest {
        val detections = listOf(mockk<Detection>())
        coEvery { repository.detectDice(any()) } returns detections
        viewModel.detectDice(mockBitmap)
        assert(viewModel.detectionState.value is DetectionState.Processing)
        testDispatcher.scheduler.advanceUntilIdle()
        assert(viewModel.detectionState.value is DetectionState.Success)
    }

```

Listing 6.2: Integration Test for Dice Detection

6.2 Test Cases and Testing Scope

The application's testing strategy utilized full and partial approaches, as outlined below.

6.2.1 Full Testing

Full testing focused on critical components of the application:

- **Game Logic**: Included comprehensive tests of all game rules and scoring mechanics, targeting game manager components:

```
    @Test
    fun `test player rolls 1 and loses turn`() {
        val initialState = pigGameManager.initializeGame()
            .copy(currentPlayerIndex = 0)
        val updatedState = pigGameManager.handleTurn(initialState, 1)
        assertEquals(AI_PLAYER_ID.hashCode(), updatedState.currentPlayerIndex)
        assertEquals(0, updatedState.currentTurnScore)
        assertEquals(0, updatedState.playerScores[0])
    }

```

Listing 6.3: Unit Test for Game Logic

- **Integration Testing**: Validated the interaction of various application modules and their integration within a single system.

```
    @Test
    fun `integration test gameViewModel`() = runTest {
        val gameState = viewModel.handleTurn(listOf(1))
       assert(gameState.currentPlayerIndex == AI_PLAYER_ID.hashCode())
    }

```

Listing 6.4: Integration Test for Game View Model

- **User Interface**: Usability and responsiveness of the UI were assessed across different devices and screen sizes.

6.2.2 Partial Testing

Partial testing was applied to less critical components, with an emphasis on functionality isolation:

- **Unit Testing**: Individual methods were assessed using JUnit tests focused on isolated code.

```
1     @Test
2     fun `test roll dice for Greed game`() = runTest {
3         val results = diceManager.rollDiceForBoard(GameBoard.GREED.modeName)
4         assertEquals(6, results.size)
5         results.forEach { dice ->assertTrue(dice in 1..6)}
6     }
7
```

Listing 6.5: Unit Test for Dice Rolling

- **Regression Testing**: Existing functionality was assessed post-fixes and/or new additions to the code base, specifically to check and confirm errors fixed do not manifest and do not effect further components.

```
1     @Test
2     fun `test game completion`() {
3         // Create a state where all categories are filled except one
4         val allCategories = BalutGameManager.CATEGORIES
5             .associateWith { 10 }
6             .toMutableMap()
7         allCategories.remove("Choice")
8         val initialState = balutGameManager.initializeGame()
9             .copy(playerScores = mapOf(0 to allCategories))
10        val updatedState = balutGameManager.scoreCategory(
11            initialState, listOf(1,2,3,4,5), "Choice")
12        assertTrue(updatedState.isGameOver)
13        assertEquals(15, updatedState.playerScores[0]?.get("Choice"))
14    }
15
```

Listing 6.6: Regression Test for Game Completion

6.3 Detected and Fixed Bugs

During testing, several issues were identified and subsequently fixed. Notable bugs included:

- **Shake Detection Sensitivity**: Rapid shaking triggered multiple dice rolls, fixed with a debounce mechanism.

```
    fun resumeShakeDetection() {
        shakeDetectionManager.setOnShakeListener {
            viewModelScope.launch {
                shakeFlow.emit(Unit)
            }
        }
        viewModelScope.launch {
            shakeFlow
                .debounce(300) // Added 300ms debounce
                .collect {
                    if (!isRolling.value && isRollAllowed.value)
                        rollDice()
                }
        }
    }

```

Listing 6.7: Fix for Shake Detection Sensitivity

- **Settings Navigation State Loss**: Navigation to the settings screen reset the game state; addressed by using launched effects for navigation to ensure game state retention.

```
    LaunchedEffect(selectedBoard) {
        // Only reset game if board type changes
        if (selectedBoard != currentBoardType) {
            viewModel.setSelectedBoard(boardType)
            viewModel.resetGame()
        }
    }

```

Listing 6.8: Fix for Settings Navigation State Loss

- **Game State Initialization**: Game state not correctly initialized leading to incorrect player scores. Was fixed by ensuring the game state was reset correctly when starting the game.

```
    fun resetGame() = viewModelScope.launch {
        _showWinDialog.value = false
        _heldDice.value = emptySet()
        diceManager.resetGame()
        statisticsManager.startGameTiming()
    }

```

Listing 6.9: Fix for Game State Initialization

- **UI Responsiveness**: Some UI elements were unresponsive on certain devices. This was resolved with optimization in UI rendering to render responsive on diverse display dimensions.
- **Dice Recognition Accuracy**: Dice value misidentification; fixed by refining the image processing techniques with a min-max normalization of pixel brightness values to enhance pips from backgrounds.

6.4 Results of Experiments

Throughout the development of the application, several experiments were conducted to assess the performance and usability of the system. The following results were observed:

- **Performance Metrics**: The application maintained a consistent frame rate of 60 FPS during gameplay, regardless of whether the player was engaged with a single opponent or multiple opponents. This demonstrates the application's smoothness and efficiency in rendering, which is crucial for delivering a responsive and enjoyable user experience.
- **User Feedback**: User testing was conducted with a diverse group of participants. The feedback gathered indicated high levels of satisfaction, with an average rating of 4.5 out of 5 for both the user interface and overall functionality. This suggests that the design is intuitive, easy to use, and meets the expectations of the target audience.
- **Code Quality and Maintainability**: During development, efforts were made to maintain a high standard of code quality. This was reflected in the following practices:
  - **Clear Documentation**: Each method and class was thoroughly documented using consistent header comments, making it easier for future developers to understand and extend the codebase.
  - **Modular Design**: The code was structured to follow principles of modularity, with components designed to be reusable and maintainable. This approach facilitates future improvements and reduces the risk of introducing bugs.
  - **Efficient Code Practices**: The code was optimized for performance, and redundant logic was removed to ensure that the application runs efficiently, even as new features are added.
  - **Consistent Coding Standards**: A consistent coding style was enforced throughout the project, making the codebase easier to read and reducing cognitive load when navigating different parts of the application.

Chapter 7

Conclusions

This chapter summarizes the project's key findings and achievements, reflecting on the objectives outlined in the thesis, the challenges encountered during development, and potential paths for future enhancements.

The game project successfully met its primary objectives, resulting in a modern Android application that implements multiple classic and custom dice game variants. The application features three distinct game variants: Pig, Greed, and Balut each with unique rules and gameplay mechanics, which were designed to enhance user engagement and provide a comprehensive gaming experience. An adaptive AI opponent was also developed to challenge players, adjusting its difficulty based on their performance. This AI provides a more engaging experience, and encourages strategic thinking, making the game more dynamic. The application also boasts a user-friendly interface designed with a modern Material Design 3 UI, ensuring an intuitive user experience. The implementation of customizable themes and touch controls enhances accessibility and user satisfaction. The application follows MVVM and Clean Architecture principles, which promote maintainability and scalability. Dependency injection using Hilt and reactive programming with Kotlin Coroutines and Flow were effectively utilized. Finally, the application was validated with a thorough testing strategy, including unit tests and integration tests, which ensured the reliability and stability of the application.

Throughout the development, several challenges were encountered. Implementing the various game rules and ensuring accurate scoring mechanisms proved complex, requiring extensive testing and debugging. Creating an adaptive AI that could effectively challenge players was also a significant hurdle, requiring much trial and error to balance the AI's difficulty level. The design of a user-friendly interface that accommodates various screen sizes also required careful consideration and multiple iterations to achieve the desired outcome. Furthermore, implementing user authentication and data synchronization presented complexities. Setting up and managing user authentication with Firebase, navigating its documentation, and handling different authentication flows was challenging. Similarly, ensuring seamless data synchronization between Firebase and Android's DataStore required

careful management of data consistency and conflict resolution. While TensorFlow Lite was initially considered for image recognition, Roboflow was chosen for its streamlined approach to computer vision tasks. Roboflow's platform provided ready-to-use tools for dataset management, model training, and mobile deployment, allowing for faster development while maintaining high accuracy in dice detection.

While the project has achieved significant milestones, several avenues for future development remain. Future updates could include additional game variants or modes, such as multiplayer options or online leaderboard. This would be useful to enhance competitiveness and social interaction among players. Further development could also focus on improving the AI's decision-making algorithms, and providing the option to select the difficulty of the AI. Integrating augmented reality (AR) elements could also provide a more immersive gaming experience, allowing players to interact with the game in new and innovative ways. Expanding the application to support other platforms, such as iOS or web-based versions, could also broaden the user base and increase accessibility. Finally, the implementation of a feedback mechanism within the app could help gather user insights, guide future enhancements, and ensure that the application continuously meets user expectations.

Bibliography

[1] App Annie. *The State of Mobile 2021*. 2021. URL: https://www.slideshare.net/slideshow/app-annie-the-state-of-mobile-2021/241508622/ (visited on 05/01/2025).

[2] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Switzerland: Springer, 2018. ISBN: 978-3-319-63519-4.

[3] Kotlin. *Kotlin Documentation*. URL: https://kotlinlang.org/docs/android-overview.html (visited on 17/12/2024).

[4] Roboflow. *Roboflow*. URL: https://docs.roboflow.com/ (visited on 15/01/2025).

[5] Dice Detection. *Kavi Dataset*. https://universe.roboflow.com/dice-detection-fj0am/kavi-zbra1. Available on Roboflow Universe. 2025. URL: https://universe.roboflow.com/dice-detection-fj0am/kavi-zbra1.

[6] Figma. *Figma Learn*. 2024. URL: https://help.figma.com/hc/en-us/articles/14563969806359-What-is-Figma (visited on 15/01/2025).

[7] Git. *Git Documentation*. URL: https://git-scm.com/doc (visited on 17/12/2024).

[8] Android Developer. *Android Studio*. URL: https://developer.android.com/studio/ (visited on 05/10/2024).

[9] S. Olofsson and S. Juhlin. *Performance Evaluation of Jetpack Compose: A Comparison with XML-Based UI Development*. 2023. URL: https://www.diva-portal.org/smash/get/diva2%3A1763502/FULLTEXT01.pdf (visited on 15/01/2025).

[10] Android Developer. *Jetpack Compose*. URL: https://developer.android.com/jetpack/compose (visited on 05/10/2024).

[11] Android Developer. *Dependency Injection with Hilt*. URL: https://developer.android.com/training/dependency-injection/hilt-android/ (visited on 10/10/2024).

[12] Lottie. *Lottie Animation*. URL: https://airbnb.io/lottie/#/android (visited on 05/01/2024).

[13] Vico Charts. *Vico Charts Documentation*. URL: https://vicotools.com/charts/documentation (visited on 05/01/2025).

[14] Ravi Tamada. *Android Better Logging using Timber Library*. 2024. URL: https://www.androidhive.info/2024/10/android-better-logging-using-timber.html (visited on 06/01/2025).

[15] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt and Christian Stein. *JUnit 5 User Guide*. 2024. URL: https://junit.org/junit5/docs/current/user-guide/#writing-tests (visited on 06/01/2024).

[16] Richard Szeliski. *Computer Vision: Algorithms and Applications*. New York: Springer Science & Business Media, 2010. ISBN: 978-1-84882-941-1.

[17] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Advances in Neural Information Processing Systems 25 (NIPS 2012)*. 2012, pp. 1097–1105. URL: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[18] Joseph Redmon and Ali Farhadi. 'YOLOv3: An Incremental Improvement'. In: *arXiv preprint arXiv:1804.02767* (2018). URL: https://arxiv.org/abs/1804.02767.

[19] Salman Khan, Muhammad Saleem and Muhammad Azam. 'A Survey on Image Classification with Deep Learning: Algorithms, Designs, and Applications'. In: *IEEE Access* 8 (2020), pp. 122368–122396. DOI: 10.1109/ACCESS.2020.2990135. URL: https://ieeexplore.ieee.org/document/9104017.

[20] Harsh Munshi. *D3 Deep Dice Detector*. https://github.com/harshmunshi/D3-Deep-Dice-Detector. 2018.

[21] Ignacio Ordovás Pascual. *Dice Scores Recognition*. https://github.com/ordovas/dice-scores-recognition. 2024.

[22] Pandula Péter. *Zilch-Dice*. https://github.com/pandulapeter/zilch-dice. 2024.

[23] Akmal Muhaimin. *Flutzy*. https://github.com/amuhaimin02/flutzy. 2024.

[24] Jonathan Kuhl. *Python Dice*. https://github.com/jckuhl/Python-Dice. 2024.

[25] Nell Byler. *Dice Detection Project*. https://github.com/nell-byler/dice_detection. 2018.

[26] *Android Data Backup Documentation*. URL: https://developer.android.com/guide/topics/data/backup (visited on 07/01/2025).

[27] *OWASP Top 10, Security Misconfiguration*. URL: https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A06-Security_Misconfiguration (visited on 07/01/2025).

[28] *Android code obfuscation documentation*. URL: https://developer.android.com/build/shrink-code (visited on 07/01/2025).

[29] Sr.IOS Developer. *How not to get desperate with MVVM implementation*. 2018. URL: https://aruniphoneapplication.blogspot.com/2018/07/how-not-to-get-desperate-with-mvvm.html (visited on 05/01/2025).

[30] Mayokun Sofowora. *Kavi thesis project*. https://www.figma.com/design/LGri6Wuhf4xouWkr1is Kavi-thesis-project. 2024.

[31] Shreyansh Saurabh. *Dice*. https://github.com/binaryshrey/Dice. 2021.

[32] Kevin Forsberg and Harold Mooz. *The V-Model: Application and Implementation in Software Engineering*. 2nd. New York: Springer, 2009.

Appendices

Index of abbreviations and symbols

AI	Artificial Intelligence.
API	Application Programming Interface.
APK	Android Package Kit.
AR	Augmented Reality.
ARM	Advanced RISC(Reduced Instruction Set Computing) Machine.
CNN	Convolutional Neural Network.
FPS	Frames Per Second.
GB	Gigabyte.
HTTPS	Hypertext Transfer Protocol Secure.
KTX	Kotlin Extensions.
MB	Megabyte.
ML	Machine Learning.
MVVM	Model View ViewModel.
OS	Operating System.
R8	R8 Architecture.
RAM	Random Access Memory.
RGB	Red Green Blue.
SDK	Software Development Kit.
SSL	Secure Sockets Layer.
TLS	Transport Layer Security.

UI User Interface.

UML Unified Modelling Language.

USB Universal Serial Bus.

UX User Experience.

XML Extensible Markup Language.

Listings

List of additional files in electronic submission

Additional files uploaded to the system include:

- source code of the application,
- test data,
- a video file showing how the application developed for the thesis is used.

List of Figures

List of Tables