ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

«Δημιουργία Παιχνιδιού σε Android παρόμοιου με το Γράμματα και Λέξεις που χρησιμοποιεί έξυπνες τεχνικές»

Του φοιτητή
ΜΠΟΝΗ ΘΕΟΔΩΡΟΥ
ΑΡ. ΜΗΤΡΩΟΥ: 134009

Επιβλέπων
ΕΥΣΤΑΘΙΟΣ ΑΝΤΩΝΙΟΥ

ΘΕΣΣΑΛΟΝΙΚΗ, ΜΑΙΟΣ 2024

Τίτλος Δ.Ε

Δημιουργία Παιχνιδιού σε Android παρόμοιου με το Γράμματα και Λέξεις που χρησιμοποιεί έξυπνες τεχνικές
Κωδικός Δ.Ε **23169**

Φοιτητής: **Μπόνης Θεόδωρος**

Εισηγητής: **Αντωνίου Ευστάθιος**

Ημερομηνία ανάληψης Δ.Ε 24/03/2023

Ημερομηνία περάτωσης Δ.Ε. 24/05/2024

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Πρόλογος

Ονομάζομαι Μπόνης Θεόδωρος και επέλεξα το θέμα "Δημιουργία Παιχνιδιού σε Android παρόμοιου με το Γράμματα και Λέξεις που χρησιμοποιεί έξυπνες τεχνικές" για την πτυχιακή μου εργασία διότι θεωρώ ότι οι σύγχρονες τεχνολογίες κινητών εφαρμογών αποτελούν τον ακρογωνιαίο λίθο της ψηφιακής εποχής μας. Η αδιάκοπη εξέλιξη των smartphones και των φορητών συσκευών έχει αλλάξει δραματικά τον τρόπο με τον οποίο οι άνθρωποι αλληλεπιδρούν με την τεχνολογία και τον κόσμο γύρω τους. Μέσω αυτής της πτυχιακής εργασίας, φιλοδοξώ να εξερευνήσω και να αναδείξω τις δυνατότητες που προσφέρει η πλατφόρμα Android για την ανάπτυξη έξυπνων και καινοτόμων εφαρμογών ενώ επιδιώκω να εξερευνήσω και να αξιοποιήσω τις δικές μου δυνατότητες στην ανάπτυξη εφαρμογών Android, εφαρμόζοντας έξυπνες τεχνικές και καινοτόμες λύσεις που θα με βοηθήσουν να εξελιχθώ ως προγραμματιστής και να αποκτήσω πολύτιμη εμπειρία στον τομέα της τεχνολογίας.

Περίληψη

Η παρακάτω πτυχιακή εργασία αφορά την δημιουργία μιας εφαρμογής Android με θέμα την δημιουργία ενός Παιχνιδιού τύπου σταυρόλεξου με έξυπνες τεχνικές, όπως την έξυπνη τοποθέτηση ελληνικών λέξεων καθώς επίσης και την αξιολόγηση επίδοσης του χρήστη. Αρχικά στο πρώτο κεφάλαιο έγινε μία εισαγωγή σχετικά με την εμφάνιση του Android στην αγορά εργασίας συγκριτικά με το iOS. Στην συνέχεια παρουσιάστηκε ο τρόπος σχεδιασμού των παιχνιδιών, οι τεχνικές απόδοσης και βελτιστοποίησης ενός παιχνιδιού Android καθώς και τα μέσα διανομής τους. Έπειτα ακολούθησε η επιλογή προτύπου, η ανάλυση της αρχιτεκτονικής και των εργαλείων που χρειάστηκαν για την διεκπεραίωση της εφαρμογής, ενώ παράλληλα ο τρόπος διεξαγωγής της εφαρμογής τους. Φυσικά δεν παραλήφθηκε η επισήμανση της υλοποίησης των παραπάνω βημάτων στο πρακτικό κομμάτι του «World of Words» με τα αντίστοιχα σχήματα. Τέλος έγινε ανασκόπηση της μεθοδολογίας αλλά και της υλοποίησης της εργασίας, των συμπερασμάτων καθώς και των μελλοντικών προκλήσεων.

Abstract

The present thesis presents the implementation of an Android application focused on developing a crossword puzzle game using intelligent techniques, such as smart placement of Greek words and user performance evaluation. Initially, in the first chapter, an introduction was provided regarding the emergence of Android in the job market compared to iOS. Subsequently, the design methods for games, performance and optimization techniques for an Android game, and their distribution methods were presented. This was followed by the selection of a template, an analysis of the architecture and tools required for the implementation of the application, and the way their application was conducted. Naturally, the implementation of the aforementioned steps in the practical part of "World of Words" with the corresponding diagrams was highlighted. Finally, there was a review of the methodology and implementation of the project, the conclusions, as well as future challenges.

Περιεχόμενα

Πίνακας Εικόνων - Πινάκων

Συντομογραφίες

Π.Ε. Πτυχιακή Εργασία

HTC High Tech Computer

IDE Integrated Development Environment

SDK Software Development Kit

VM Virtual Machine

ART Android Runtime

APK Android Package

UI User Interface

API Application Programming Interface

MVP Model View Presenter

MVC Model View Controller

MVVM Model-View-ViewModel

JSON JavaScript Object Notation

DI Dependency Injection

Εισαγωγή

Τα παιχνίδια σταυρόλεξα έχουν μακρά ιστορία στον κόσμο των παζλ, παρέχοντας διασκέδαση και γνωστική πρόκληση σε εκατομμύρια ανθρώπους. Με την εξέλιξη της τεχνολογίας και την εξάπλωση των κινητών συσκευών, τα σταυρόλεξα έχουν μεταφερθεί και στην ψηφιακή εποχή, με πλήθος εφαρμογών διαθέσιμων στις πλατφόρμες Android και iOS. Ωστόσο, παρά τη δημοτικότητά τους, πολλά από αυτά τα παιχνίδια φαίνεται να υστερούν σε έναν κρίσιμο τομέα: την έξυπνη αλληλεπίδραση με τον χρήστη. Η αλληλεπίδραση μεταξύ του χρήστη και της εφαρμογής είναι καθοριστική για την εμπειρία του παιχνιδιού. Τα παραδοσιακά σταυρόλεξα στις κινητές συσκευές συχνά περιορίζονται σε βασικές λειτουργίες, όπως η εισαγωγή λέξεων και η εύρεση επιπλέον λέξεων. Αυτή η προσέγγιση, αν και λειτουργική, δεν εκμεταλλεύεται πλήρως τις δυνατότητες των σύγχρονων τεχνολογιών για να δημιουργηθεί μια πιο δυναμική και εμπλουτισμένη εμπειρία χρήστη. Η παρούσα πτυχιακή εργασία στοχεύει να καλύψει αυτό το κενό, παρουσιάζοντας την ανάπτυξη μιας εφαρμογής Android που ενσωματώνει έξυπνες τεχνικές για την αλληλεπίδραση με τον χρήστη. Η εφαρμογή αυτή συνδυάζει τις παραδοσιακές αρχές των σταυρολέξων με τις σύγχρονες δυνατότητες βελτιστοποίησης όπως και επίσης τις έξυπνες τεχνικές προσφέροντας μια βελτιωμένη και πιο ελκυστική εμπειρία παιχνιδιού στον χρήστη. Προκειμένου λοιπόν να γίνει επίτευξη του παραπάνω στόχου είναι απαραίτητο να τεθούν οι βάσεις, οι οποίες απαιτούνται για την σωστή υλοποίηση της συγκεκριμένης εφαρμογής. Σε πρώτο στάδιο γίνεται η μελέτη σχεδιασμού των απαιτήσεων, τους τρόπους με τους οποίους μπορούμε να ενισχύσουμε την απόδοση της καθώς επίσης και την διανομή της. Σε δεύτερο στάδιο γίνεται διεξοδική μελέτη ως προς τα πρότυπα που χρησιμοποιούνται στις Android εφαρμογές, τις δυνατότητες και τα μειονεκτήματα τους. Παράλληλα αναδεικνύεται το ιδανικό πρότυπο από αυτά, ενώ παρουσιάζεται αναλυτικά η αρχιτεκτονική δομή «Clean Code» την οποία πρέπει η εκάστοτε εφαρμογή να τηρεί. Σε τρίτο στάδιο γίνεται ξεχωριστή αναφορά σε κάθε εργαλείο που πρόκειται να αξιοποιηθεί ως προς την χρήση του. Έπειτα ακολουθούν τα βήματα υλοποίησης των βασικών τμημάτων της εφαρμογής με τα αντίστοιχα σχήματα σε πρακτικό πλάνο ενώ παρουσιάζονται εικόνες κατά την διάρκεια εκτέλεσης της εφαρμογής. Τέλος, τονίζονται οι παρατηρήσεις, η διεξαγωγή των συμπερασμάτων που προκύπτουν από την συνολική διαδικασία και οι προοπτικές των μελλοντικών προκλήσεων.

Κεφάλαιο 1ο: Το Λ/Σ Android

Η δημιουργία και η εξέλιξη του λειτουργικού συστήματος Android είναι μια ιστορία που περιλαμβάνει την καινοτομία, τη συνεργασία και τη μεταμόρφωση της αγοράς κινητών συσκευών. Η εταιρεία που ανέπτυξε το Android ξεκίνησε το 2003 ως μια ανεξάρτητη εταιρεία που ιδρύθηκε από τον Andy Rubin και άλλους συνιδρυτές με στόχο την ανάπτυξη ενός λειτουργικού συστήματος για ψηφιακές κάμερες. Γρήγορα όμως η εταιρεία επαναπροσδιόρισε το στόχο της, βλέποντας τις δυνατότητες για ένα λειτουργικό σύστημα κινητών συσκευών. Το 2005, η Google εξαγόρασε το Android και συνέχισε την ανάπτυξή του ως μέρος της στρατηγικής της για την είσοδο στην αγορά κινητών. Η Google προώθησε μια προσέγγιση ανοικτού κώδικα, ενθαρρύνοντας κατασκευαστές και προγραμματιστές να συμβάλλουν στη δημιουργία ενός ευέλικτου λειτουργικού συστήματος για κινητές συσκευές.

Το 2007, η Google ανακοίνωσε το Android στο ευρύ κοινό ως μέρος της Open Handset Alliance, μιας συνεργασίας εταιρειών τεχνολογίας με στόχο την προώθηση της ανάπτυξης ενός ανοικτού λειτουργικού συστήματος για κινητά. Η πρώτη έκδοση του Android (Android 1.0) κυκλοφόρησε το 2008 μαζί με το HTC Dream, το πρώτο εμπορικό smartphone που χρησιμοποίησε το Android. Σε σχέση με άλλες τεχνολογίες, το Android έχει ορισμένα σημαντικά πλεονεκτήματα:

- **Ανοιχτός Κώδικας**: Η Google παρείχε τον πηγαίο κώδικα του Android στο κοινό, επιτρέποντας στους κατασκευαστές να προσαρμόζουν το λειτουργικό σύστημα στις δικές τους συσκευές.
- **Ευελιξία και Προσαρμοστικότητα**: Σε αντίθεση με άλλα λειτουργικά συστήματα, όπως το iOS της Apple, το Android προσφέρει μεγαλύτερη ελευθερία στους κατασκευαστές και στους προγραμματιστές να δημιουργούν προσαρμοσμένες εμπειρίες χρήστη.
- **Οικοσύστημα Εφαρμογών**: Με το Google Play Store, οι προγραμματιστές είχαν ένα κεντρικό σημείο για τη διανομή των εφαρμογών τους, επιτρέποντας τη γρήγορη ανάπτυξη ενός πλούσιου οικοσυστήματος εφαρμογών.

Σε σύγκριση με άλλες τεχνολογίες, το Android κατάφερε να κερδίσει μεγάλο μερίδιο αγοράς, κυρίως λόγω της ανοικτής και ευέλικτης προσέγγισής του. Αυτό το έκανε ελκυστικό τόσο στους κατασκευαστές όσο και στους προγραμματιστές, επιτρέποντας τη δημιουργία ενός ευρέως και ποικίλου οικοσυστήματος συσκευών και εφαρμογών. Με το πέρασμα του χρόνου, το Android εδραιώθηκε ως μια από τις κύριες πλατφόρμες κινητών συσκευών παγκοσμίως.

Το Android βασίζεται στον πυρήνα του Linux, γεγονός που του προσφέρει αξιοπιστία, σταθερότητα και λειτουργίες χαμηλού επιπέδου, όπως διαχείριση μνήμης, ασφάλεια, προγραμματισμό διεργασιών και επικοινωνία με το υλικό. Το λειτουργικό σύστημα έχει μια δομή που περιλαμβάνει πολλαπλά επίπεδα, με τα βασικά στοιχεία να περιλαμβάνουν:

- **Βιβλιοθήκες Android**: Περιλαμβάνουν βασικά στοιχεία, όπως OpenGL για γραφικά, SQLite για βάσεις δεδομένων, και άλλες βιβλιοθήκες που βοηθούν στη λειτουργικότητα του λειτουργικού συστήματος.

- **Android Runtime**: Περιλαμβάνει τη Virtual Machine (VM) Dalvik, που αργότερα αντικαταστάθηκε από την Android Runtime (ART), η οποία εκτελεί τις εφαρμογές Android.
- **Framework Εφαρμογών**: Προσφέρει τα βασικά στοιχεία που χρειάζονται οι εφαρμογές, όπως το Activity Manager, το Window Manager, και άλλες υπηρεσίες που διευκολύνουν τη διασύνδεση μεταξύ εφαρμογών και συστήματος.
- **Εφαρμογές Συστήματος**: Περιλαμβάνει βασικές εφαρμογές, όπως το τηλέφωνο, το ημερολόγιο, τα μηνύματα και άλλα, τα οποία συνοδεύουν το λειτουργικό σύστημα.

1.1 Σύγκριση Android και iOS

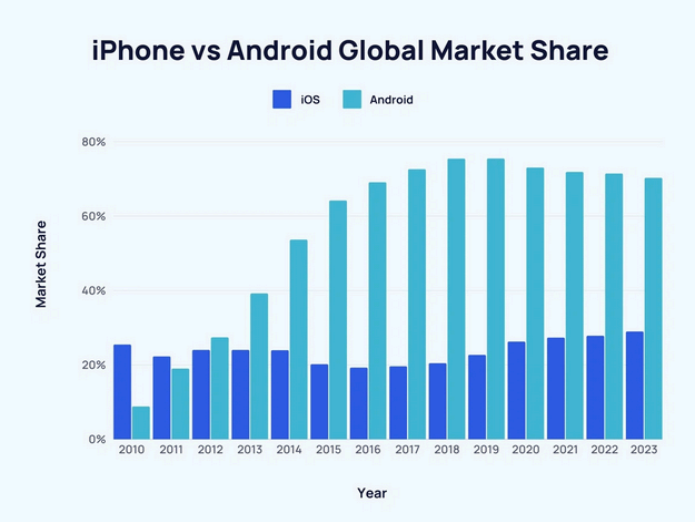
Η σύγκριση μεταξύ της δημοτικότητας των Android και iOS από την ίδρυσή τους μέχρι σήμερα αντικατοπτρίζει τον ανταγωνισμό και την εξέλιξη της αγοράς των κινητών συσκευών[5]. Το iOS, που κυκλοφόρησε αρχικά το 2007 με το πρώτο iPhone, είχε το πλεονέκτημα ότι ήταν το πρώτο λειτουργικό σύστημα που συνδυάστηκε με ένα επαναστατικό smartphone. Το iPhone άλλαξε τον τρόπο που οι άνθρωποι σκέφτονταν για τα κινητά τηλέφωνα και δημιούργησε μια νέα κατηγορία συσκευών. Αυτό έδωσε στο iOS μια ισχυρή θέση στην αγορά κατά τα πρώτα χρόνια. Αντιθέτως το Android, όπως προαναφέραμε, κυκλοφόρησε επίσημα το 2008 με το HTC Dream, μερικούς μήνες μετά το iPhone. Το Android είχε μια πιο αργή αρχή σε σύγκριση με το iOS, αλλά η στρατηγική ανοικτού κώδικα και η συνεργασία με πολλούς κατασκευαστές κινητών τηλεφώνων επέτρεψαν στο Android να κερδίσει έδαφος.

Με το πέρασμα των χρόνων, το Android έγινε πιο δημοφιλές λόγω της ευελιξίας του και της ποικιλίας συσκευών που υποστηρίζει. Με κατασκευαστές όπως η Samsung, η Huawei, η LG και πολλοί άλλοι να υιοθετούν το Android, το μερίδιο αγοράς του Android αυξήθηκε δραματικά. Το γεγονός ότι το Android είναι ανοικτού κώδικα και προσφέρει περισσότερες επιλογές προσαρμογής το έκανε ελκυστικό για ένα ευρύ κοινό.Παρά την αυξανόμενη δημοτικότητα του Android, το iOS διατήρησε ένα σταθερό και πιστό κοινό. Η Apple επικεντρώθηκε στην προσφορά μιας συνεπούς και υψηλής ποιότητας εμπειρίας χρήστη, με ένα ελεγχόμενο οικοσύστημα και αυστηρά πρότυπα για εφαρμογές. Αυτό το έκανε ελκυστικό για χρήστες που εκτιμούν την απλότητα, την ασφάλεια και τη συνέπεια.

Σήμερα, το Android έχει το μεγαλύτερο μερίδιο αγοράς παγκοσμίως. Σύμφωνα με πρόσφατες στατιστικές, το Android αντιπροσωπεύει πάνω από το 70% της παγκόσμιας αγοράς κινητών συσκευών. Η ευρεία ποικιλία συσκευών και τιμών, από χαμηλού κόστους έως premium, συμβάλλει στη μεγάλη δημοτικότητά του. Το iOS, παρότι έχει μικρότερο μερίδιο αγοράς σε σύγκριση με το Android, διατηρεί ισχυρή παρουσία, ιδιαίτερα στις ανεπτυγμένες αγορές όπως οι Ηνωμένες Πολιτείες και η Δυτική Ευρώπη. Η Apple διατηρεί υψηλή κερδοφορία μέσω της πώλησης hardware υψηλής ποιότητας και μιας ισχυρής βάσης πελατών που είναι πρόθυμη να πληρώσει για premium συσκευές και υπηρεσίες.

Στην Εικόνα 1.1 παρατηρούνται τα αποτελέσματα μιας έρευνας που πραγματοποιήθηκε το 2023 σχετικά με τη δημοτικότητα των Android και των iOS κινητών συσκευών παγκοσμίως ανά έτος. Παρατηρούμε όντως πόσο δημοφιλή ήταν κινητές συσκευές με λογισμικό iOS κατά την

πρωτοεμφάνιση των έξυπνων συσκευών ενώ με το πέρασμα των χρόνων παρατηρείται πως το Android κυριάρχησε στην αγορά εργασίας παγκοσμίως.



Εικόνα 1.1 Android vs iOS Αγορά Εργασίας

Κεφάλαιο 2ο: Ανάπτυξη Παιχνιδιών στο Android

Με την ανοδική πορεία της τεχνολογίας κινητής επικοινωνίας, τα παιχνίδια για κινητές συσκευές έχουν κατακτήσει έναν καθοριστικό ρόλο στην ψυχαγωγία των ανθρώπων. Με την ευρεία διάδοση των κινητών συσκευών Android, οι προγραμματιστές έχουν τη δυνατότητα να δημιουργήσουν εκπληκτικά παιχνίδια που μπορούν να συναρπάσουν τους χρήστες με την εμπειρία τους. Το Android Game Development δεν είναι μόνο ένας τρόπος δημιουργίας ψυχαγωγικών παιχνιδιών, αλλά και μια διαδικασία που απαιτεί τεχνική εμπειρία και χρόνο. Από τη σχεδίαση και την υλοποίηση του παιχνιδιού μέχρι τη βελτιστοποίηση της απόδοσης και την κυκλοφορία στο Google Play Store, οι προγραμματιστές πρέπει να διαθέτουν μια ολοκληρωμένη κατανόηση του οικοσυστήματος του Android και των αρχών του παιχνιδιού. Αυτή η διαδικασία περιλαμβάνει πολλαπλά στάδια, από την αρχική ιδέα έως τον σχεδιασμό, την προγραμματιστική υλοποίηση, τη δοκιμή και τελικά τη διανομή του παιχνιδιού.

Για την επιτυχημένη ανάπτυξη παιχνιδιών στο Android, οι προγραμματιστές πρέπει να διαθέτουν μια σειρά από εξής εργαλεία:

- **Γλώσσες Προγραμματισμού**: Η ανάπτυξη Android ιστορικά έγινε κυρίως με τη γλώσσα Java, η οποία παρέχει ισχυρές δυνατότητες αντικειμενοστραφούς προγραμματισμού και έχει πλούσια υποστήριξη βιβλιοθηκών. Η Kotlin, μια πιο σύγχρονη γλώσσα προγραμματισμού που υποστηρίζεται από την Google ως η επίσημη γλώσσα για το Android, έχει κερδίσει σημαντική δημοτικότητα λόγω της απλούστερης σύνταξης και των προηγμένων δυνατοτήτων, όπως οι επεκτάσεις και οι μηδενικές ασφαλισμένες μεταβλητές.Η καλή κατανόηση των γλωσσών αυτών επιτρέπει στους προγραμματιστές να δημιουργούν αξιόπιστο και επεκτάσιμο κώδικα.
- **Εργαλεία Ανάπτυξης**: Το Android Studio είναι το κύριο Integrated Development Environment (IDE) για την ανάπτυξη εφαρμογών και παιχνιδιών Android. Παρέχει μια σειρά από εργαλεία, όπως ο κώδικας επεξεργασίας με αυτο-συμπλήρωση, ο γραφικός σχεδιαστής διεπαφών, και οι ενσωματωμένοι εξομοιωτές για τη δοκιμή εφαρμογών.Το Android Studio προσφέρει επίσης εξελιγμένα εργαλεία παρακολούθησης απόδοσης, debugging, και profiling, επιτρέποντας στους προγραμματιστές να εντοπίζουν και να διορθώνουν προβλήματα στο λογισμικό τους.
- **Πλαίσια Ανάπτυξης Παιχνιδιών**: Πλαίσια όπως το Unity και το Unreal Engine έχουν καταστεί δημοφιλή για την ανάπτυξη παιχνιδιών Android, καθώς προσφέρουν ισχυρές δυνατότητες δημιουργίας γραφικών, φυσικής και ήχου. Το Unity είναι ιδιαίτερα δημοφιλές λόγω της ευκολίας χρήσης και της μεγάλης κοινότητας που το υποστηρίζει, ενώ το Unreal Engine παρέχει προηγμένες δυνατότητες και ποιότητα γραφικών.Αυτά τα πλαίσια επιτρέπουν στους προγραμματιστές να επικεντρωθούν στη δημιουργικότητα και στο gameplay, αφήνοντας τα περισσότερα τεχνικά ζητήματα, όπως η διαχείριση γραφικών και η φυσική, στο πλαίσιο.
- **Λογισμικό Τρίτων και Βιβλιοθήκες**: Υπάρχει μια μεγάλη γκάμα βιβλιοθηκών και λογισμικού τρίτων που μπορούν να χρησιμοποιηθούν για τη δημιουργία παιχνιδιών Android. Αυτές οι βιβλιοθήκες μπορούν να βοηθήσουν σε διάφορες εργασίες, όπως η διαχείριση δεδομένων, η δικτύωση, οι λειτουργίες backend, και η δημιουργία ειδικών εφέ.Παραδείγματα δημοφιλών βιβλιοθηκών περιλαμβάνουν την LibGDX για ανάπτυξη 2D παιχνιδιών, την Box2D για φυσική

προσομοίωση, και το Firebase για cloud-based λειτουργίες, όπως η αποθήκευση και η ταυτόχρονη ενημέρωση δεδομένων.

Με αυτά τα βασικά στοιχεία, οι προγραμματιστές μπορούν να ξεκινήσουν το ταξίδι τους στην ανάπτυξη παιχνιδιών Android, δημιουργώντας ποιοτικά παιχνίδια που ενθουσιάζουν και προσελκύουν παίκτες σε όλο τον κόσμο.

2.1 Σχεδιασμός και Gameplay

Ο σχεδιασμός και το gameplay αποτελούν την καρδιά ενός επιτυχημένου Android παιχνιδιού [6][7]. Ας εξετάσουμε σε βάθος τους διάφορους παράγοντες που σχετίζονται με τον σχεδιασμό και το gameplay:

2.1.1 Σχεδιασμός

Η αφήγηση αποτελεί έναν σημαντικό παράγοντα για πολλά παιχνίδια. Μια ισχυρή ιστορία μπορεί να προσελκύσει τους παίκτες και να τους κρατήσει ενδιαφέροντες.Το σενάριο πρέπει να έχει σαφή δομή, με αρχή, μέση και τέλος, καθώς και καλοσχεδιασμένους χαρακτήρες με διακριτές προσωπικότητες.

Ο σχεδιασμός επίπεδων (Level Design) αφορά τη δημιουργία των διαφόρων σταδίων του παιχνιδιού. Τα επίπεδα πρέπει να είναι ισορροπημένα, προσφέροντας μια πρόκληση χωρίς να είναι υπερβολικά δύσκολα. Ο σχεδιασμός πρέπει επίσης να λαμβάνει υπόψη την κλίση της δυσκολίας, διασφαλίζοντας ότι το παιχνίδι γίνεται πιο δύσκολο όσο προχωράει, χωρίς να γίνεται αποτρεπτικό.

Η διασύνδεση χρήστη (User Interface,UI) είναι το σημείο όπου οι παίκτες αλληλεπιδρούν με το παιχνίδι. Ένα καλά σχεδιασμένο UI πρέπει να είναι απλό, ευανάγνωστο και διαισθητικό.Το UI περιλαμβάνει στοιχεία όπως κουμπιά, μενού, και ενδείξεις παιχνιδιού (όπως ζωές, βαθμολογία, ή επίπεδο ενέργειας). Η σωστή τοποθέτηση και λειτουργικότητα αυτών των στοιχείων είναι κρίσιμη για μια καλή εμπειρία χρήστη.

2.1.2 Gameplay

Το gameplay αφορά την εμπειρία που έχουν οι παίκτες καθώς παίζουν το παιχνίδι. Καλύπτει όλες τις ενέργειες που μπορούν να εκτελέσουν οι παίκτες, καθώς και τις αντιδράσεις του παιχνιδιού σε αυτές τις ενέργειες. Τα βασικά στοιχεία του gameplay είναι:

- **Μηχανισμοί Παιχνιδιού (Game Mechanics)**: Οι μηχανισμοί παιχνιδιού είναι οι κανόνες και οι διαδικασίες που καθορίζουν πώς λειτουργεί το παιχνίδι. Περιλαμβάνουν τις κινήσεις του χαρακτήρα, τις αλληλεπιδράσεις με αντικείμενα, και τους κανόνες για την επίτευξη στόχων. Οι μηχανισμοί πρέπει να είναι σαφείς και λογικοί, επιτρέποντας στους παίκτες να μάθουν και να εφαρμόσουν στρατηγικές για να πετύχουν στο παιχνίδι.
- **Εξισορρόπηση (Balancing)**: Η εξισορρόπηση είναι κρίσιμη για να διατηρήσει το παιχνίδι δίκαιο και ενδιαφέρον. Αυτό περιλαμβάνει τη ρύθμιση της δυσκολίας, την αποφυγή υπερβολικά ισχυρών ή αδύναμων χαρακτήρων ή αντικειμένων, και τη διατήρηση μιας δίκαιης πρόκλησης. Ο σχεδιαστής πρέπει να δοκιμάσει το παιχνίδι εκτενώς για να διασφαλίσει ότι οι παίκτες δεν θα βρουν στρατηγικές που χαλάνε την ισορροπία του παιχνιδιού.
- **Προκλήσεις και Ανταμοιβές**: Οι προκλήσεις πρέπει να είναι ποικίλες και να κρατούν το ενδιαφέρον των παικτών. Οι ανταμοιβές είναι σημαντικές για να ενθαρρύνουν τους παίκτες να συνεχίσουν να παίζουν. Αυτές μπορεί να περιλαμβάνουν βαθμολογία, αναβαθμίσεις, ή και ιστορικά στοιχεία που ξεκλειδώνονται καθώς προχωράει το παιχνίδι.

2.2 Τεχνικές Απόδοσης και Βελτιστοποίησης

2.2.1 Τεχνικές Απόδοσης

- **Αποδοτική Χρήση Πόρων**: Η χρήση πόρων όπως η μνήμη και ο επεξεργαστής πρέπει να είναι όσο το δυνατόν πιο αποδοτική. Πολλαπλοί επεξεργαστές και πολυπύρηνα συστήματα επιτρέπουν την κατανομή του φορτίου εργασίας για καλύτερη απόδοση.Είναι απαραίτητο να βεβαιωθούμε ότι δεν υπάρχουν διαρροές μνήμης, καθώς αυτό μπορεί να οδηγήσει σε πτώση της απόδοσης ή ακόμα και σε κρασαρίσματα [4].
- **Διαχείριση Εικόνων και Γραφικών**: Οι εικόνες και τα γραφικά πρέπει να είναι όσο το δυνατόν πιο συμπιεσμένα χωρίς να χάνεται η ποιότητά τους. Είναι απαραίτητο να χρησιμοποιούνται κατάλληλα εργαλεία συμπίεσης για να μειωθεί το μέγεθος των αρχείων και μορφές εικόνων που υποστηρίζονται εγγενώς από το Android, όπως το PNG και το WebP, για την καλύτερη απόδοση και συμβατότητα.
- **Χρήση Παιχνιδιών με Χαμηλή Κατανάλωση Πόρων**:Η ανάπτυξη παιχνιδιών που απαιτούν χαμηλότερους πόρους επιτρέπει την καλύτερη απόδοση σε μια ευρεία γκάμα συσκευών. Είναι συνετό να αποφεύγεται η υπερβολική χρήση γραφικών και άλλων πόρων που ενδέχεται να προκαλέσουν προβλήματα απόδοσης.

2.2.2 Τεχνικές Βελτιστοποίησης

- **Προ-φόρτωση Δεδομένων**: Η προ-φόρτωση δεδομένων μπορεί να βοηθήσει στην εξομάλυνση των διακοπών κατά τη διάρκεια του παιχνιδιού όπως εικόνες, ήχους, και δεδομένα επιπέδων πριν χρειαστούν, για να εξασφαλιστεί μια ομαλή εμπειρία παιχνιδιού [4].
- **Βελτιστοποίηση Κώδικα**: Ο κώδικας πρέπει να είναι καθαρός και απλός, αποφεύγοντας περιττές λειτουργίες και βεβαιώνοντας ότι χρησιμοποιούνται αποτελεσματικοί αλγόριθμοι για τις διάφορες λειτουργίες του παιχνιδιού, ενώ χρησιμοποιούνται εργαλεία profiling για να εντοπιστούν τυχόν σημεία συμφόρησης και να τα βελτιστοποιήσουμε [4]**.**
- **Αποφυγή Υπερβολικών Επαναλήψεων**: Ο κώδικας που εκτελείται σε βρόγχους ή σε επαναλαμβανόμενες λειτουργίες μπορεί να επιβαρύνει την απόδοση. Χρειάζεται να διασφαλιστεί ότι οι επαναλήψεις εκτελούνται με αποτελεσματικό τρόπο και αποτρέψτε περιττές λειτουργίες που μπορούν να επιβραδύνουν το παιχνίδι.
- **Ορθή Διαχείριση Αντικειμένων**: Η σωστή διαχείριση των αντικειμένων στο παιχνίδι είναι κρίσιμη για την αποφυγή διαρροών μνήμης και υπερβολικής χρήσης πόρων. Είναι απαραίτητο να χρησιμοποιούνται μοτίβα σχεδιασμού όπως το object pooling για να μειώσετε τη δημιουργία νέων αντικειμένων κατά τη διάρκεια του παιχνιδιού, βεβαιώνοντας ότι τα αντικείμενα που δεν χρησιμοποιούνται διαγράφονται ή επαναχρησιμοποιούνται, αποφεύγοντας τη συσσώρευση περιττών αντικειμένων στη μνήμη.
- **Δοκιμές σε Διάφορες Συσκευές**: Για να εξασφαλιστεί ότι το παιχνίδι θα λειτουργεί σε μια ευρεία γκάμα συσκευών, είναι απαραίτητο να δοκιμαστεί σε διαφορετικά μοντέλα, διαφορετικούς κατασκευαστές, και διαφορετικές εκδόσεις Android. Ακόμη πρέπει να χρησιμοποιηθούν εργαλεία δοκιμών που προσομοιώνουν διαφορετικές συνθήκες, όπως ταχύτητα δικτύου, χρήση μνήμης, και πόρους επεξεργαστή [4].
- **Χρήση Ανάλυσης Απόδοσης (Profiling)**: Τέλος Το Android Studio παρέχει εργαλεία profiling που επιτρέπουν την παρακολούθηση της απόδοσης του παιχνιδιού σε πραγματικό χρόνο. Χρησιμοποιώντας τα μπορούν να εντοπιστούν σημεία συμφόρησης και να βελτιστοποιηθεί ο κώδικας. Μπορούν να παρακολουθεί η κατανάλωση μνήμης, η χρήση CPU, και η καθυστέρηση για να εξασφαλιστεί ότι το παιχνίδι τρέχει ομαλά και αποδίδει καλά.

2.3 Διανομή Παιχνιδιών Android

Η διανομή των παιχνιδιών Android αφορά τις μεθόδους με τις οποίες τα παιχνίδια γίνονται διαθέσιμα στους χρήστες. Το Google Play Store είναι ο κύριος τόπος διανομής, αλλά υπάρχουν και άλλες επιλογές που μπορεί να εξετάσουν οι προγραμματιστές. Οι διάφορες πτυχές της διανομής που έχουν παρατηρηθεί είναι:

1. **Google Play Store**: Είναι η κύρια πλατφόρμα διανομής για παιχνίδια Android. Για να διανείμουμε το παιχνίδι μας μέσω του Google Play Store, θα πρέπει να εγγραφτούμε ως προγραμματιστές και να πληρώσουμε ένα εφάπαξ τέλος εγγραφής.Το Google Play Store προσφέρει δυνατότητες για προώθηση και διαφήμιση παιχνιδιών, αλλά έχει επίσης αυστηρές πολιτικές και κανονισμούς που πρέπει να τηρηθούν. Οι προγραμματιστές πρέπει να εξασφαλίσουν ότι το παιχνίδι τους συμμορφώνεται με τις οδηγίες του Google Play και να περάσουν από τη διαδικασία αναθεώρησης πριν από τη δημοσίευση.
2. **Εναλλακτικές Πλατφόρμες**: Υπάρχουν εναλλακτικές πλατφόρμες διανομής παιχνιδιών Android, όπως το Amazon Appstore και το Samsung Galaxy Store. Αυτές οι πλατφόρμες μπορεί να έχουν διαφορετικά κοινά-στόχους και μπορεί να προσφέρουν διαφορετικές ευκαιρίες για προώθηση.Οι προγραμματιστές πρέπει να προσαρμόσουν το παιχνίδι τους ανάλογα με τις απαιτήσεις κάθε πλατφόρμας και να συμμορφωθούν με τις αντίστοιχες πολιτικές και όρους χρήσης.
3. **Διανομή μέσω Αρχείων APK**: Η διανομή μέσω αρχείων APK (Android Package) επιτρέπει στους προγραμματιστές να διανέμουν το παιχνίδι τους απευθείας στους χρήστες. Ωστόσο, αυτή η μέθοδος έχει κινδύνους, καθώς το APK μπορεί να είναι ευάλωτο σε αλλοιώσεις ή κακόβουλες δραστηριότητες.Οι προγραμματιστές που επιλέγουν αυτή τη μέθοδο πρέπει να λάβουν μέτρα ασφαλείας, όπως η ψηφιακή υπογραφή των αρχείων και η εξασφάλιση ασφαλούς μεταφοράς.

Κεφάλαιο 3ο: Τεχνικές και Μοτίβα του World of Words

3.1 Αρχιτεκτονική

3.1.1 Η αρχή του ¨Διαχωρισμού των Ευθυνών¨

Στη σύγχρονη ανάπτυξη εφαρμογών Android, η αρχιτεκτονική αποτελεί έναν από τους πιο κρίσιμους παράγοντες που καθορίζουν την επιτυχία ή την αποτυχία μιας εφαρμογής. Όπως ένα γερό θεμέλιο υποστηρίζει ένα κτήριο, έτσι και μια καλά σχεδιασμένη αρχιτεκτονική διασφαλίζει ότι η εφαρμογή είναι σταθερή, επεκτάσιμη και συντηρήσιμη με την πάροδο του χρόνου. Η ανάγκη για μια στιβαρή αρχιτεκτονική γίνεται εμφανής καθώς οι εφαρμογές γίνονται πιο σύνθετες και οι απαιτήσεις των χρηστών αυξάνονται. Οι προγραμματιστές πρέπει να δημιουργούν εφαρμογές που να ανταποκρίνονται γρήγορα, να είναι ανθεκτικές σε σφάλματα και να μπορούν να επεκτείνονται εύκολα με νέα χαρακτηριστικά χωρίς να θυσιάζεται η ποιότητα ή η απόδοση. Μια σωστή αρχιτεκτονική επιτρέπει στους προγραμματιστές να οργανώνουν τον κώδικά τους με τρόπο που προωθεί τον καθαρό διαχωρισμό ευθυνών, διευκολύνει την επαναχρησιμοποίηση κώδικα, και μειώνει τη σύγχυση κατά την ανάπτυξη. Επιπλέον, μια καλά σχεδιασμένη αρχιτεκτονική κάνει την εφαρμογή πιο προσβάσιμη για νέους προγραμματιστές που μπορεί να συμμετέχουν στην ανάπτυξη, επιτρέποντάς τους να κατανοήσουν γρήγορα τη δομή και τη λογική της εφαρμογής. Τα οφέλη της σωστής αρχιτεκτονικής επεκτείνονται και στη φάση των δοκιμών. Μια εφαρμογή με καλά διαχωρισμένα μέρη διευκολύνει τη διεξαγωγή μονάδων ελέγχου και ολοκληρωμένων δοκιμών, γεγονός που βελτιώνει τη συνολική ποιότητα και αξιοπιστία του προϊόντος. Επιπλέον, η αρχιτεκτονική διευκολύνει την ταχεία ανάπτυξη και τη συνεχή ολοκλήρωση, που είναι απαραίτητα στοιχεία στη σύγχρονη ανάπτυξη λογισμικού.

Η Google προτείνει μια συγκεκριμένη αρχιτεκτονική προσέγγιση για την ανάπτυξη εφαρμογών Android, που συχνά αναφέρεται ως "Android Architecture Components" ή "Android App Architecture," βασισμένη σε σύγχρονες βέλτιστες πρακτικές και αρχές της Clean Architecture. Το "Clean Code" είναι μια φιλοσοφία και πρακτική στον προγραμματισμό που επικεντρώνεται στη δημιουργία κώδικα που είναι ευανάγνωστος, εύκολα κατανοητός, και εύκολος στη συντήρηση. Ο όρος έγινε ευρύτερα γνωστός από το βιβλίο "Clean Code: A Handbook of Agile Software Craftsmanship" του Robert C. Martin, γνωστού και ως "Uncle Bob", το οποίο προτείνει ένα σύνολο αρχών και πρακτικών για τη δημιουργία καθαρού κώδικα. Το Clean Code είναι κώδικας που πληροί ορισμένα χαρακτηριστικά που τον καθιστούν εύκολο να διαβαστεί, να κατανοηθεί και να τροποποιηθεί [1] (βλ. **Εικόνα 3.1** *Clean Architecture*).

Οι συστάσεις της Google για την αρχιτεκτονική Android επικεντρώνονται σε διάφορες αρχές που προέρχονται από την Clean Architecture, συμπεριλαμβανομένης της θεμελιώδης αρχής του "διαχωρισμού των ευθυνών" (separation of concerns), της μονοκατευθυντικής ροής δεδομένων και της αναστροφής εξαρτήσεων (dependency inversion). Για αυτό το λόγο η ύπαρξη διακριτών επιπέδων data domain και UI (User Interface) σε μια εφαρμογή Android οδηγεί σε πολλαπλά πλεονεκτήματα, συμπεριλαμβανομένης της βελτιωμένης συντήρησης, της ευελιξίας, της καλύτερης αρχιτεκτονικής και της διευκόλυνσης των δοκιμών [1].

1. Επίπεδο Δεδομένων (Data Layer):

Στο πλαίσιο της Clean Architecture, το Data Layer (Επίπεδο Δεδομένων) είναι το μέρος της εφαρμογής που είναι υπεύθυνο για την πρόσβαση και τη διαχείριση των δεδομένων. Αυτό περιλαμβάνει τη σύνδεση με εξωτερικές πηγές δεδομένων, όπως βάσεις δεδομένων, δίκτυα, ή άλλες υπηρεσίες [1] (βλ. **Εικόνα 3.1** *Clean Architecture*). Η Google προτείνει μια δομή του Data Layer που περιλαμβάνει συνήθως τα ακόλουθα συστατικά:

- **Repositories (Αποθετήρια)**: Τα Repositories λειτουργούν ως διαμεσολαβητές μεταξύ του Domain Layer (επίπεδο επιχειρηματικής λογικής) και των πηγών δεδομένων. Εξάγουν τις λεπτομέρειες της πρόσβασης στα δεδομένα από τον υπόλοιπο κώδικα, παρέχοντας μια ενιαία διασύνδεση για τη διαχείριση των δεδομένων. Περιλαμβάνουν λειτουργίες για την ανάκτηση, αποθήκευση, ενημέρωση και διαγραφή δεδομένων, καθώς και για τον συντονισμό της ροής των δεδομένων μεταξύ των πηγών και του υπόλοιπου συστήματος.

- **Data Sources (Πηγές Δεδομένων)**: Οι πηγές δεδομένων είναι υπεύθυνες για την πραγματική ανάκτηση και αποθήκευση των δεδομένων. Ο τρόπος με τον οποίο λειτουργούν μπορεί να είναι διαφορετικός ανάλογα με το είδος της πηγής. Υπάρχουν κυρίως δύο είδη πηγών, οι τοπικές πηγές που μπορεί να αφορά είτε την πρόσβαση σε μια τοπική βάση δεδομένων που μπορεί να γίνει με SQL queries μέσω του Room είτε με τοπική αποθήκευση όπως Preferences Datastore ή Shared Preferences και αρχεία, ενώ η δεύτερη κατηγορία πηγής που αφορά την πρόσβαση σε μια απομακρυσμένη πηγή όπως υπηρεσίες ιστού, δίκτυα, εξωτερικές βάσεις δεδομένων καθώς και API μπορεί να γίνει με HTTP requests μέσω βιβλιοθηκών όπως το Retrofit.

- **Μοντέλα Δεδομένων**: Τα μοντέλα δεδομένων αντιπροσωπεύουν τα δεδομένα που αποθηκεύονται ή ανακτώνται από το Data Layer. Μπορούν να είναι αντικείμενα που χρησιμοποιούνται για την αποθήκευση και μεταφορά δεδομένων μεταξύ των διαφορετικών στρωμάτων του συστήματος. Στο Room, αυτά τα μοντέλα είναι συνήθως οντότητες με annotations που καθορίζουν το σχήμα των πινάκων της βάσης δεδομένων. Για απομακρυσμένες πηγές, τα μοντέλα μπορεί να είναι DTOs (Data Transfer Objects) που αντικατοπτρίζουν τα δεδομένα που λαμβάνονται από ένα API.

- **Διαχείριση Εξάρτησης (Dependency Management)**: Το Data Layer πρέπει να είναι σχεδιασμένο με τέτοιο τρόπο ώστε να είναι εύκολα δοκιμαστέο και ευέλικτο. Σε αυτό μπορεί να γίνονται οι δηλώσεις του dependency injection (π.χ. με το Dagger ή το Hilt) για τη διαχείριση των

εξαρτήσεων, έτσι ώστε τα Repositories και οι Data Sources να μπορούν να ελέγχονται και να αλλάζουν χωρίς να επηρεάζουν τα υπόλοιπα στρώματα.Η σωστή διαχείριση εξαρτήσεων διασφαλίζει ότι οι εξαρτήσεις είναι σαφείς και διαχωρισμένες, μειώνοντας το σύζευγμα και βελτιώνοντας τη συντηρησιμότητα του κώδικα.

2. Επίπεδο Domain (Domain Layer):

Το Domain Layer (επίπεδο επιχειρηματικής λογικής) είναι η καρδιά της εφαρμογής στη Clean Architecture, όπου τοποθετείται η επιχειρηματική λογική, οι κανόνες και οι διαδικασίες που καθορίζουν τη λειτουργία της εφαρμογής. Είναι ανεξάρτητο από λεπτομέρειες υλοποίησης, όπως frameworks, UI, και συγκεκριμένες τεχνολογίες αποθήκευσης και είναι το επίπεδο όπου εφαρμόζεται η επιχειρηματική λογική, οι κανόνες, και οι βασικές διαδικασίες. Με αυτήν την ανεξαρτησία, η εφαρμογή μπορεί να είναι ευέλικτη, επαναχρησιμοποιήσιμη, και εύκολη στη δοκιμασία και συντήρηση, καθιστώντας το Domain Layer το κεντρικό σημείο της αρχιτεκτονικής [1] (βλ. **Εικόνα 3.1** *Clean Architecture*).

- **Entities (Οντότητες)**: Οι οντότητες αντιπροσωπεύουν τα βασικά επιχειρηματικά αντικείμενα και τους κανόνες που σχετίζονται με αυτά. Είναι αντικειμενοστραφή μοντέλα που ενσωματώνουν τη λογική και τη συμπεριφορά της επιχείρησης. Για παράδειγμα σε μια εφαρμογή ηλεκτρονικού εμπορίου, οι οντότητες μπορεί να είναι προϊόντα, πελάτες, παραγγελίες, κ.λπ. Ο ρόλος τους είναι να περιέχουν επιχειρηματικούς κανόνες και συνήθως δεν εξαρτώνται από συγκεκριμένα frameworks ή βιβλιοθήκες, καθώς και μπορούν να έχουν δικούς τους κανόνες εγκυρότητας, λειτουργίες, και σχέσεις με άλλες οντότητες.

- **Use Cases (Χρήσεις / Δράσεις)**: Σε αυτό το επίπεδο θα συναντήσουμε τις "Χρήσεις" (Use Cases) ή διαφορετικά τους διαμεσολαβητές (interactors) που αντιπροσωπεύουν τις κύριες επιχειρηματικές λειτουργίες ή δράσεις που μπορεί να κάνει η εφαρμογή. Κάθε χρήση αντιπροσωπεύει μια συγκεκριμένη επιχειρηματική διαδικασία. Τα Use Cases περιέχουν τη ροή των γεγονότων και τον τρόπο με τον οποίο οι οντότητες αλληλεπιδρούν μεταξύ τους και με τα άλλα επίπεδα. Είναι υπεύθυνες για την επιβολή επιχειρηματικών κανόνων και τη διασφάλιση της σωστής ροής εργασίας και πάντοτε έχουν μια συγκεκριμένη λειτουργία που εκτελούν. Για παράδειγμα σε μια εφαρμογή ηλεκτρονικού εμπορίου, οι χρήσεις μπορεί να περιλαμβάνουν "προσθήκη προϊόντος στο καλάθι", "ολοκλήρωση αγοράς", "εγγραφή νέου χρήστη", κ.λπ.

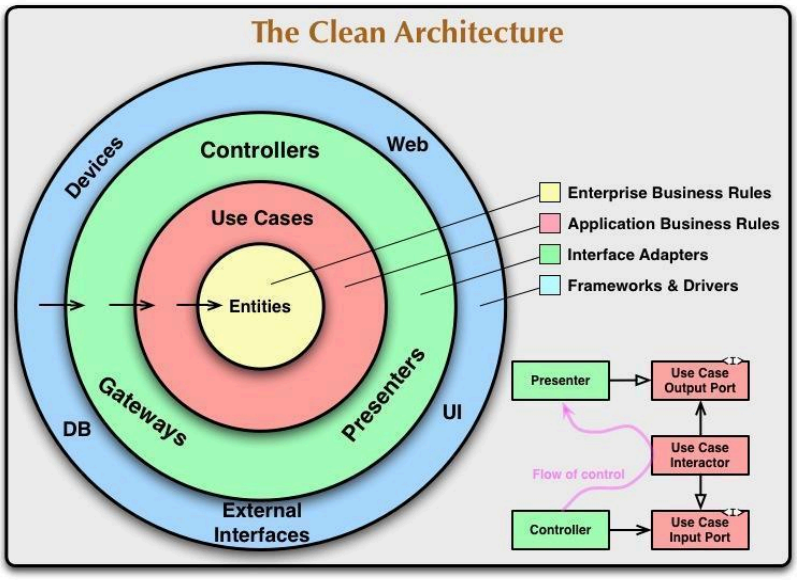
- **Ανεξαρτησία από Frameworks**: Το Domain Layer πρέπει να είναι όσο το δυνατόν πιο ανεξάρτητο από εξαρτήσεις με συγκεκριμένα frameworks ή τεχνολογίες. Αυτό το χαρακτηριστικό επιτρέπει την επαναχρησιμοποίηση, τη δοκιμαστικότητα, και την ανεξαρτησία από πλαίσια UI ή δεδομένων. Αυτή η ανεξαρτησία σημαίνει ότι το Domain Layer δεν πρέπει να εξαρτάται από συγκεκριμένες υλοποιήσεις δεδομένων ή UI. Επιτρέπει την επαναχρησιμοποίηση της επιχειρηματικής λογικής σε διαφορετικά περιβάλλοντα ή πλαίσια.

3. Επίπεδο Παρουσίασης (Presentation Layer):

Τέλος, το Presentation Layer (Επίπεδο Παρουσίασης) στη Clean Architecture είναι το τμήμα της εφαρμογής που είναι υπεύθυνο για την εμφάνιση και την αλληλεπίδραση με τον χρήστη. Αυτό το επίπεδο επικεντρώνεται στον τρόπο με τον οποίο οι χρήστες βλέπουν και αλληλεπιδρούν με την εφαρμογή, χωρίς να εξαρτάται από τις λεπτομέρειες υλοποίησης της επιχειρηματικής λογικής ή των δεδομένων [1] (βλ. **Εικόνα 3.1** *Clean Architecture*).

Τα Views και τα στοιχεία του UI (User Interface) αποτελούν τη διεπαφή που βλέπει και με την οποία αλληλεπιδρά ο χρήστης. Αυτό μπορεί να περιλαμβάνει Activities, Fragments, Widgets, και διάφορα άλλα στοιχεία του UI. Τα views είναι υπεύθυνες για την παρουσίαση των δεδομένων στον χρήστη και για την αντίδραση στις ενέργειές του. Πρέπει να είναι καθαρά από την επιχειρηματική λογική και να επικεντρώνονται στη διάταξη και την εμπειρία του χρήστη. Μπορούν να περιλαμβάνουν οθόνες με λίστες προϊόντων, φόρμες εισαγωγής δεδομένων, κουμπιά, γραμμές εργαλείων, κ.λπ.

Αυτό το επίπεδο περιλαμβάνει τον τρόπο με τον οποίο οι χρήστες αλληλεπιδρούν με την εφαρμογή και πώς τα συμβάντα αυτής της αλληλεπίδρασης μεταφέρονται στα ViewModels και περαιτέρω στο Domain Layer. Τα Views συχνά συλλέγουν συμβάντα από τους χρήστες, όπως κλικ, εισαγωγή δεδομένων, ή άλλες ενέργειες, και τα διαβιβάζουν στα ViewModels, τα οποία στη συνέχεια τα κατευθύνουν στα κατάλληλα Use Cases. Ακόμα είναι σημαντικό να διατηρείται ένας καθαρός διαχωρισμός μεταξύ των Views και των ViewModels, διασφαλίζοντας ότι οι λογικές επιχειρηματικές διαδικασίες παραμένουν εκτός του Presentation Layer.



Εικόνα 3.1 Clean Architecture

3.1.2 Βασικές Αρχές του Clean Code

Οι αρχές του Clean Code περιλαμβάνουν διάφορες πρακτικές και προσεγγίσεις για την επίτευξη αυτών των χαρακτηριστικών [1]:

- **Κατάλληλα Ονόματα**: Τα ονόματα των μεταβλητών, των συναρτήσεων, των κλάσεων, κ.λπ., πρέπει να είναι περιγραφικά και κατανοητά, αντικατοπτρίζοντας τον σκοπό τους. Αυτό βοηθά στη διευκόλυνση της ανάγνωσης και της κατανόησης του κώδικα.
- **Μικρές και Κατανοητές Συναρτήσεις**: Οι συναρτήσεις πρέπει να είναι μικρές, εκτελώντας μία και μόνο μία λειτουργία, και να έχουν σαφείς εισόδους και εξόδους. Αυτό διευκολύνει την επαναχρησιμοποίηση και τον έλεγχο.
- **Ελάχιστες Εξαρτήσεις**: Ο κώδικας θα πρέπει να έχει όσο το δυνατόν λιγότερες εξαρτήσεις από εξωτερικούς παράγοντες, όπως άλλες συναρτήσεις ή κλάσεις. Αυτό μειώνει τη σύγχυση και τις πιθανότητες σφαλμάτων.
- **Ενιαία Ευθύνη (Single Responsibility Principle - SRP)**: Κάθε κλάση ή συνάρτηση πρέπει να έχει μία και μόνο μία ευθύνη. Αυτό μειώνει την πολυπλοκότητα και διευκολύνει τη συντήρηση.
- **Σχόλια με Σύνεση**: Τα σχόλια πρέπει να χρησιμοποιούνται με φειδώ και να επικεντρώνονται σε εξηγήσεις που δεν είναι προφανείς από τον κώδικα. Ο καθαρός κώδικας συχνά δεν χρειάζεται πολλά σχόλια, καθώς ο ίδιος ο κώδικας είναι επαρκώς κατανοητός.
- **Αποφυγή Διπλότυπου Κώδικα**: Ο διπλότυπος κώδικας πρέπει να αποφεύγεται. Αν παρατηρείτε επαναλαμβανόμενες ενότητες κώδικα, είναι καλύτερο να δημιουργήσετε συναρτήσεις ή κλάσεις που εξυπηρετούν τον σκοπό αυτό, ώστε να μειωθεί η πολυπλοκότητα και να διευκολυνθεί η συντήρηση.
- **Συνεχής Βελτίωση**: Ο καθαρός κώδικας δεν είναι στατικός. Οι προγραμματιστές θα πρέπει να αναθεωρούν και να βελτιώνουν συνεχώς τον κώδικα, αφαιρώντας τα περιττά μέρη, βελτιστοποιώντας τον σχεδιασμό, και εφαρμόζοντας νέες βέλτιστες πρακτικές.

3.1.3 Οφέλη του Clean Code

Ο διαχωρισμός αυτών των επιπέδων σε μια εφαρμογή Android προσφέρει πολλά πλεονεκτήματα [1]:

- **Διαχωρισμός των Ευθυνών**: Κάθε επίπεδο έχει μια συγκεκριμένη ευθύνη. Το data domain ασχολείται με τα δεδομένα και την επιχειρησιακή λογική, ενώ το UI με την παρουσίαση και τις αλληλεπιδράσεις με τους χρήστες. Αυτό οδηγεί σε λιγότερη πολυπλοκότητα και καλύτερη οργάνωση του κώδικα.
- **Ευκολία Συντήρησης**: Όταν ο κώδικας είναι καθαρά διαχωρισμένος, οι αλλαγές σε ένα επίπεδο δεν επηρεάζουν τα υπόλοιπα. Αυτό κάνει πιο εύκολη τη συντήρηση και την επέκταση της εφαρμογής.
- **Ευκολία Δοκιμών**: Με τον ξεκάθαρο διαχωρισμό, είναι ευκολότερο να δημιουργηθούν μονάδες ελέγχου (unit tests) και δοκιμές ολοκλήρωσης (integration tests), αφού η λογική της επιχείρησης μπορεί να ελεγχθεί ανεξάρτητα από το UI.
- **Ευελιξία**: Η διακριτή οργάνωση επιτρέπει την ευκολότερη προσαρμογή του κώδικα σε αλλαγές στις απαιτήσεις του επιχειρησιακού τομέα ή του UI. Επίσης, επιτρέπει την επαναχρησιμοποίηση κώδικα σε διαφορετικά περιβάλλοντα, όπως web, Android, iOS κ.λπ.
- **Βελτιωμένη Κλιμάκωση**: Μια αρχιτεκτονική με σαφή διαχωρισμό επιπέδων μπορεί να κλιμακωθεί πιο εύκολα, διότι είναι απλούστερο να εντοπιστούν και να επεκταθούν οι περιοχές που χρειάζονται βελτίωση ή επιπλέον πόρους.
- **Καθαρή Αρχιτεκτονική**: Ο διαχωρισμός των επιπέδων προάγει μια πιο καθαρή και ευανάγνωστη αρχιτεκτονική, η οποία είναι εύκολη να κατανοηθεί από νέους προγραμματιστές και να αναπτυχθεί περαιτέρω.

Αυτό έχει ως αποτέλεσμα ο διαχωρισμός του data domain και του UI layer σε μια εφαρμογή Android είναι απαραίτητος για μια καθαρή, ευέλικτη, και εύκολα συντηρήσιμη αρχιτεκτονική. Προωθεί τον καθαρό διαχωρισμό των ευθυνών, διευκολύνει τη συντήρηση και τις δοκιμές, και επιτρέπει την ευελιξία και την κλιμάκωση της εφαρμογής. Αυτή η αρχιτεκτονική προσέγγιση είναι θεμελιώδης για την ανάπτυξη σύγχρονων, υψηλής ποιότητας εφαρμογών Android.

3.2 Πρότυπα Σχεδίασης

Όταν αναπτύσσουμε μια εφαρμογή Android, η χρήση ενός αρχιτεκτονικού προτύπου θεωρείται μια από τις καλύτερες πρακτικές από τους προγραμματιστές. Ένα αρχιτεκτονικό πρότυπο βοηθά να οργανώσουμε τη δομή του έργου με τρόπο που καθιστά τα αρχεία πιο ευέλικτα και διαχειρίσιμα. Επιπλέον, επιτρέπει τη δημιουργία μονάδων δοκιμής (Unit Testing), που είναι κώδικας ο οποίος ελέγχει και επιβεβαιώνει ότι άλλα τμήματα του λογισμικού λειτουργούν σωστά με βάση τα αναμενόμενα αποτελέσματα. Με αυτόν τον τρόπο, οι προγραμματιστές μπορούν ευκολότερα να συντηρούν το λογισμικό, ενώ έχουν τη δυνατότητα να προσθέτουν νέες λειτουργίες στην εφαρμογή στο μέλλον [8].

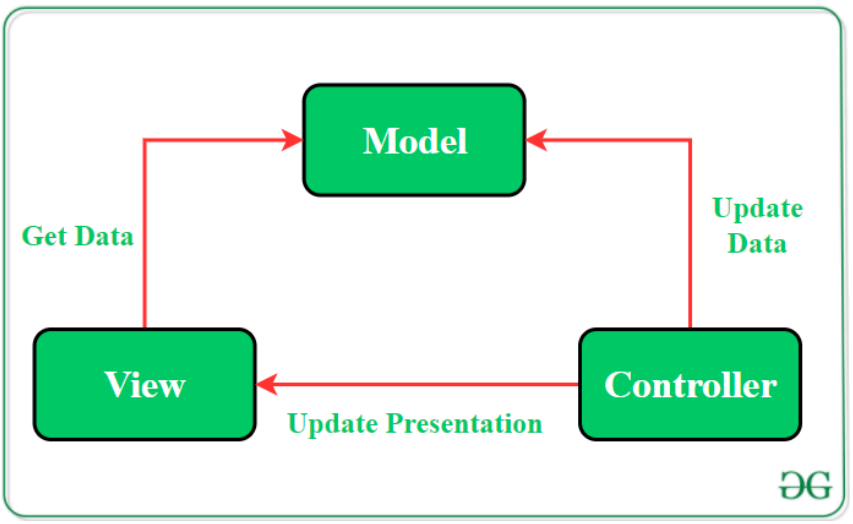
3.2.1 Πρότυπο MVC (Model-View-Controller)

Σε αυτό το πλαίσιο, οι αρχιτεκτονικές προσεγγίσεις όπως το MVC, το MVP και το MVVM έχουν αναδειχθεί ως δημοφιλείς μέθοδοι για την οργάνωση εφαρμογών Android. Καθεμία από αυτές προσφέρει πλεονεκτήματα και μειονεκτήματα, και η επιλογή της κατάλληλης αρχιτεκτονικής εξαρτάται από τις απαιτήσεις του έργου, το μέγεθος της ομάδας ανάπτυξης, και την πολυπλοκότητα της εφαρμογής.

Στη συνέχεια, θα εξετάσουμε τις διαφορές μεταξύ αυτών των αρχιτεκτονικών, τον τρόπο εφαρμογής τους σε εφαρμογές Android, και γιατί το MVVM θεωρείται συχνά ως η πιο αποτελεσματική προσέγγιση για την ανάπτυξη σύγχρονων Android εφαρμογών. Η αρχιτεκτονική MVC χωρίζει την εφαρμογή σε τρία κύρια μέρη [3][19] (βλ. **Εικόνα 3.1** *Πρότυπο MVC*):

1. **Model (Μοντέλο)**: Περιέχει τη λογική των δεδομένων, συμπεριλαμβανομένων των δομών δεδομένων και των λειτουργιών διαχείρισης δεδομένων. Στο Android, αυτό μπορεί να περιλαμβάνει βάσεις δεδομένων, REST APIs, και άλλα στοιχεία διαχείρισης δεδομένων.
2. **View (Προβολή)**: Είναι το επίπεδο παρουσίασης της εφαρμογής, δηλαδή το πώς εμφανίζεται στον χρήστη. Στο Android, οι προβολές είναι τα στοιχεία της διεπαφής χρήστη (UI) που αλληλεπιδρούν με τον χρήστη.
3. **Controller (Ελεγκτής)**: Ο συνδετικός κρίκος μεταξύ του μοντέλου και της προβολής. Είναι υπεύθυνος για τη διαχείριση της αλληλεπίδρασης του χρήστη με την προβολή και για την ενημέρωση του μοντέλου βάσει των ενεργειών του χρήστη.

Η κύρια αδυναμία του MVC είναι ότι ο ελεγκτής μπορεί να γίνει υπερβολικά πολύπλοκος, οδηγώντας σε αυτό που ονομάζεται "Massive View Controller". Αυτό συμβαίνει επειδή ο ελεγκτής αναλαμβάνει συχνά πάρα πολλές ευθύνες, καθιστώντας δύσκολη τη συντήρηση και τον έλεγχο του κώδικα.



Εικόνα 3.2 Πρότυπο MVC

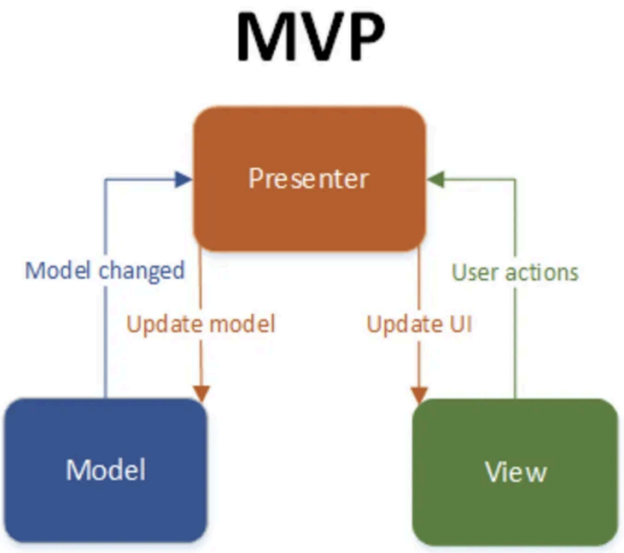
3.2.2 Πρότυπο MVP (Model-View-Presenter)

Η αρχιτεκτονική MVP εισάγει τον "Presenter" ως μεσάζοντα μεταξύ του μοντέλου και της προβολής [18]:

1. **Model (Μοντέλο)**: Όπως και στην αρχιτεκτονική MVC, το μοντέλο περιλαμβάνει τη λογική των δεδομένων.
2. **View (Προβολή)**: Αντί να αλληλεπιδρά άμεσα με το μοντέλο, η προβολή επικοινωνεί με τον παρουσιαστή.
3. **Presenter (Παρουσιαστής)**: Ο παρουσιαστής αναλαμβάνει τη λογική της παρουσίασης και αλληλεπιδρά με το μοντέλο για να ενημερώσει την προβολή. Σε αντίθεση με τον ελεγκτή, ο παρουσιαστής δεν γνωρίζει τις λεπτομέρειες της διεπαφής χρήστη, καθιστώντας τον πιο εύκολο στη διαχείριση.

Η αρχιτεκτονική MVP μειώνει την πολυπλοκότητα της προβολής, καθιστώντας την πιο ελαφριά και διατηρώντας το διαχωρισμό των ευθυνών μεταξύ του παρουσιαστή και του μοντέλου.

Ωστόσο, η κύρια αδυναμία του MVP είναι η δυσκολία του στη συντήρηση, ειδικά αν οι παρουσιαστές αρχίσουν να αναλαμβάνουν περισσότερη λογική επιχειρησιακής διαχείρισης.



Εικόνα 3.3 Πρότυπο MVP

3.2.3 Πρότυπο MVVM (Model-View-ViewModel)

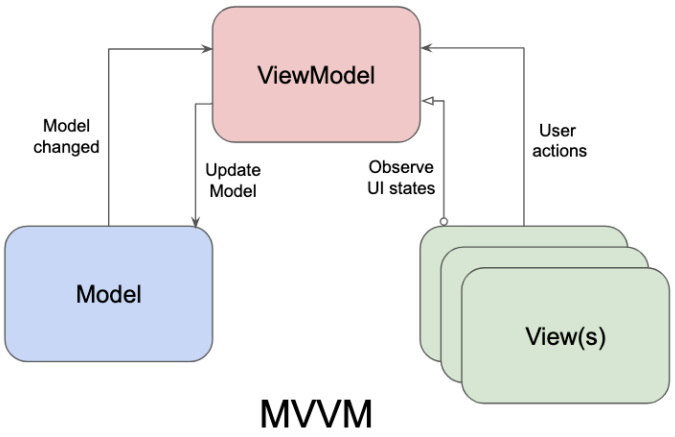
Το πρότυπο MVVM πρόκεται για το πιο εξελιγμένο και το πιο δημοφιλή στις μέρες μας όσο αφορά την ανάπτυξη android εφαρμογών. Η αρχιτεκτονική MVVM είναι μια εξέλιξη των προηγούμενων μοντέλων, εισάγοντας το "ViewModel" ως τον κύριο μεσάζοντα μεταξύ του μοντέλου και της προβολής [2][11][20] (βλ. **Εικόνα 3.1** *Πρότυπο MVVM*):

1. **Model (Μοντέλο)**: Όπως και στις προηγούμενες αρχιτεκτονικές, το μοντέλο περιλαμβάνει τα δεδομένα και τη λογική επιχειρησιακής διαχείρισης.
2. **View (Προβολή)**: Είναι υπεύθυνη μόνο για την εμφάνιση των δεδομένων στον χρήστη και για τη λήψη των εισόδων του χρήστη.
3. **ViewModel**: Ο ρόλος του ViewModel είναι να παρέχει τα δεδομένα στην προβολή με τρόπο που να διευκολύνει τη σύνδεση δεδομένων (data binding). Το ViewModel διατηρεί μια καθαρή αποσύνδεση από την προβολή, επιτρέποντας πιο ευέλικτη και αποδοτική διαχείριση των δεδομένων.

Η αρχιτεκτονική MVVM προσφέρει μια σειρά από πλεονεκτήματα:

1. **Καθαρός Διαχωρισμός Ευθυνών**: Το MVVM διατηρεί ένα καθαρό διαχωρισμό μεταξύ των επιπέδων της εφαρμογής, καθιστώντας τον κώδικα πιο συντηρήσιμο και επεκτάσιμο.
2. **Data Binding**: Η χρήση μηχανισμών data binding, όπως το Android Data Binding Library, επιτρέπει τη σύνδεση δεδομένων μεταξύ του ViewModel και της προβολής, μειώνοντας την ανάγκη για γραπτή κώδικα χειροκίνητης ενημέρωσης.
3. **Ευκολία Δοκιμών**: Το MVVM επιτρέπει τη διενέργεια μονάδων ελέγχου (unit tests) και δοκιμών ολοκλήρωσης (integration tests) χωρίς να εξαρτάται από το UI, καθιστώντας το πιο αξιόπιστο και εύκολο στον έλεγχο.
4. **Υποστήριξη Σύγχρονων Εργαλείων και Βιβλιοθηκών**: Το MVVM υποστηρίζεται από σύγχρονες βιβλιοθήκες και εργαλεία της Google, όπως το LiveData και το ViewModel στο Android Architecture Components, που διευκολύνουν την ανάπτυξη και βελτιώνουν την απόδοση.

Αυτό έχει ως συμπέρασμα η αρχιτεκτονική MVVM να θεωρείται η καλύτερη επιλογή για την ανάπτυξη εφαρμογών Android λόγω του καθαρού διαχωρισμού των ευθυνών, της ευκολίας στη σύνδεση δεδομένων, και της ευελιξίας στην ανάπτυξη και τον έλεγχο του κώδικα. Καθώς οι εφαρμογές Android γίνονται πιο σύνθετες, το MVVM παρέχει ένα ισχυρό πλαίσιο για την ανάπτυξη εφαρμογών που είναι εύκολες στη συντήρηση, δοκιμασμένες και αποδοτικές.



Εικόνα 3.4 Πρότυπο MVVM

3.3 Εργαλεία Ανάπτυξης

3.3.1 Android Studio

Το βασικό εργαλείο ανάπτυξης κάθε Android εφαρμογής είναι το Android Studio που είναι το επίσημο Integrated Development Environment (IDE) της Google για την ανάπτυξη εφαρμογών Android. Ανακοινώθηκε το 2013, καθώς πριν από αυτό γινόταν χρήση του Eclipse για την ανάπτυξη εφαρμογών Android με την χρήση της προγραμματιστικής γλώσσας Java, και από τότε αποτελεί το κύριο εργαλείο που χρησιμοποιούν οι προγραμματιστές για τη δημιουργία εφαρμογών για το λειτουργικό σύστημα Android. Είναι ένα ισχυρό και ολοκληρωμένο εργαλείο για την ανάπτυξη εφαρμογών Android, από το σχεδιασμό διεπαφών χρήστη μέχρι την κατασκευή και διανομή εφαρμογών, παρέχει όλα τα απαραίτητα εργαλεία για μια αποτελεσματική διαδικασία ανάπτυξης. Είτε πρόκειται για μικρά προσωπικά έργα είτε για μεγάλες επαγγελματικές εφαρμογές, το Android Studio προσφέρει την ευελιξία και τη δύναμη που χρειάζονται οι προγραμματιστές Android.

Το Android Studio διαθέτει πολλές προηγμένες λειτουργίες, όπως έξυπνη ολοκλήρωση κώδικα, εύκολη πλοήγηση, και ισχυρά εργαλεία αναζήτησης. Διαθέτει εργαλεία σχεδιασμού για διεπαφές χρήστη, όπως τον Layout Editor, που επιτρέπει στους προγραμματιστές να δημιουργούν γραφικά διεπαφές χωρίς να γράφουν XML με το χέρι. Έχει το εργαλείο Android Virtual Device (AVD) Manager, που επιτρέπει τη δημιουργία και τη διαχείριση εικονικών συσκευών Android για δοκιμές. Οι προγραμματιστές μπορούν να δοκιμάζουν τις εφαρμογές τους σε διάφορες εκδόσεις του Android και σε διαφορετικές συσκευές χωρίς να χρειάζονται πραγματικές συσκευές [9].

Build Tools και Gradle: Το Android Studio χρησιμοποιεί το Gradle ως σύστημα κατασκευής (build system), επιτρέποντας αυτοματισμούς και προσαρμογές στη διαδικασία κατασκευής της εφαρμογής . Σε αυτό το αρχείο ***build.grandle***, που συνήθως βρίσκεται στον ριζικό φάκελο του έργου ή σε υποέργα (modules), μπορούμε να ορίσουμε τις εκδόσεις των πακέτων, βιβλιοθηκών και εξαρτήσεων που χρησιμοποιούμε στο έργο μας. Αυτό είναι σημαντικό για να διασφαλιστεί η σταθερότητα του έργου

μας και να αποφεύγονται προβλήματα συμβατότητας που μπορεί να προκύψουν λόγω διαφορετικών εκδόσεων.

Εντοπισμός Σφαλμάτων (Debugging) και Προφίλ (Profiling): Περιλαμβάνει ισχυρά εργαλεία εντοπισμού σφαλμάτων και προφίλ που βοηθούν τους προγραμματιστές να εντοπίζουν προβλήματα και να βελτιστοποιούν την απόδοση της εφαρμογής τους.

Εργαλεία για την Κατασκευή APKs και AABs: Το Android Studio επιτρέπει τη δημιουργία αρχείων APK (Android Package) για εγκατάσταση εφαρμογών σε συσκευές και αρχείων AAB (Android App Bundle), το οποίο είναι το προτεινόμενο format για διανομή εφαρμογών μέσω του Google Play.

3.3.2 AndroidManifest

Το Android Manifest File, γνωστό ως AndroidManifest.xml, είναι ένα κρίσιμο αρχείο σε κάθε εφαρμογή Android. Το Manifest είναι ο σύνδεσμος μεταξύ της εφαρμογής και του συστήματος Android, το σύστημα χρησιμοποιεί το Manifest για να κατανοήσει τι περιλαμβάνει η εφαρμογή και πώς θα συμπεριφερθεί. Πρόκειται για ένα αρχείο διαμόρφωσης σε μορφή XML που βρίσκεται στον ριζικό κατάλογο μιας εφαρμογής Android και περιέχει μια περιγραφή των κύριων συστατικών της εφαρμογής, όπως Activities, Services, Broadcast Receivers, και Content Providers.

Παρέχει μεταδεδομένα που περιγράφουν την εφαρμογή, όπως το όνομά της, το εικονίδιο της, την έκδοση της, καθώς και τις άδειες (permissions) που χρειάζεται η εφαρμογή για να λειτουργήσει, όπως πρόσβαση στο Διαδίκτυο, στη θέση (location), ή σε άλλους πόρους του συστήματος. Κάθε Activity και Service που αποτελεί μέρος της εφαρμογής πρέπει να δηλωθεί στο Manifest. Εμπεριέχει Intent Filters, που καθορίζουν πώς ένα Activity, Service, ή Broadcast Receiver μπορεί να ανταποκριθεί σε συγκεκριμένες προθέσεις (intents), δηλαδή τι μπορεί να ξεκινήσει ένα Activity, ποιες ενέργειες μπορεί να αναλάβει ένα Service, και ποια μηνύματα μπορεί να λάβει ένας Broadcast Receiver [26] (βλ. **Εικόνα 3.5** *Android Manifest File*).

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <uses-permission android:name="android.permission.INTERNET"/>
    <application
        android:name=".ui.WorldOfWordsApplication"
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/wow_launcher_round"
        android:label="World of Words"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.AppCompat.Light"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:screenOrientation="portrait"
            android:label="World of Words"
            android:theme="@style/Theme.AppCompat.Light">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Εικόνα 3.5 Android Manifest File

3.3.3 Android Gradle

Το Gradle είναι ένα εργαλείο που χρησιμοποιείται ευρέως στην ανάπτυξη εφαρμογών Android για την αυτοματοποίηση της διαδικασίας build όπως συλλογή κώδικα, σύνδεση εξαρτήσεων, δημιουργία εκτελέσιμων αρχείων, κ.α. [13]. Ένα τυπικό έργο Android με Gradle έχει δύο βασικά αρχεία *build.gradle*:

1. **Project-level Build File**: Αυτό το αρχείο βρίσκεται στη ρίζα του έργου και καθορίζει τις γενικές ρυθμίσεις που επηρεάζουν ολόκληρο το έργο. Περιλαμβάνει *gradle Plugins* όπου ορίζονται τα *plugins* που χρειάζονται για το έργο, όπως το 'com.android.application' για εφαρμογές Android ή το 'kotlin-android' για έργα Kotlin. Ακόμα εμπεριέχει *Repositories* που καθορίζουν από πού το Gradle θα τραβάει τις εξαρτήσεις, όπως το jcenter ή το mavenCentral και *Classpaths* που αναφέρουν τις εξαρτήσεις που απαιτούνται σε επίπεδο project, όπως το 'com.android.tools.build:gradle', που είναι ο σύνδεσμος με το εργαλείο build του Android.

```
plugins { this: PluginDependenciesSpecScope
    id("com.android.application") version "8.3.2" apply false
    id("org.jetbrains.kotlin.android") version "1.9.0" apply false
    id("com.google.dagger.hilt.android") version "2.48.1" apply false
    id("org.jetbrains.kotlin.jvm") version "1.9.22"
    id("org.jetbrains.kotlin.plugin.serialization") version "1.9.22"
}
```

Εικόνα 3.6 *Project-Level Build File*

2. **Module-level Build File**: Αυτό το αρχείο βρίσκεται μέσα σε κάθε module και περιέχει ρυθμίσεις που σχετίζονται με το συγκεκριμένο module. Κάθε έργο Android έχει τουλάχιστον ένα module που αντιπροσωπεύει την εφαρμογή. Περιέχει το *Android Configuration* που αφορά τις ρυθμίσεις για το Android SDK, τις εκδόσεις στόχου (target) και ελάχιστες εκδόσεις (minSdk), και άλλες παραμέτρους όπως τα buildTypes (π.χ., debug, release) . Επίσης συναντάμε τα dependencies με τα οποία καθορίζουμε τις βιβλιοθήκες και τα frameworks που χρησιμοποιεί το module, όπως και μπορεί να περιλαμβάνει εξωτερικές βιβλιοθήκες, όπως το Retrofit, ή εσωτερικές εξαρτήσεις, όπως άλλα modules. Τέλος αποτελείται και από *Flavors και Build Variants* αν η εφαρμογή έχει πολλαπλές εκδόσεις (π.χ., δωρεάν και πληρωμένη), που μπορούν να καθοριστούν διαφορετικά 'product flavors' με συγκεκριμένες παραμέτρους για κάθε μία.

```
plugins { this: PluginDependenciesSpecScope
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
    kotlin("kapt")
    id("dagger.hilt.android.plugin")
}

android { this: BaseAppModuleExtension
    namespace = "com.example.worldofwords"
    compileSdk = 34

    defaultConfig { this: ApplicationDefaultConfig
        applicationId = "com.example.worldofwords"
        minSdk = 24
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
        vectorDrawables { this: VectorDrawables
            useSupportLibrary = true
        }
    }

    buildTypes { this: NamedDomainObjectContainer<ApplicationBuildType>
        release { this: ApplicationBuildType
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile( name: "proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }
```

Εικόνα 3.7 *Module-level Build File*

Δυνατότητες του Gradle αποτελούν :

- **Διαχείριση Build**: Με το Gradle, μπορούμε να ορίσουμε πώς θα συντίθεται και θα κατασκευάζεται το λογισμικό. Αυτό περιλαμβάνει τη δημιουργία του APK (το πακέτο εφαρμογής για Android), την υποστήριξη πολλαπλών build types, και την οργάνωση των αρχείων.
- **Αυτοματισμός**: Το Gradle επιτρέπει τη δημιουργία προσαρμοσμένων εργασιών (tasks) για αυτοματοποίηση επαναλαμβανόμενων εργασιών, όπως η εκτέλεση τεστ ή η παραγωγή νυχτερινών build.

- **Unit Testing και Code Coverage**: Με το Gradle, μπορούμε να κάνουμε δοκιμές (tests) και να διασφαλίσουμε την κάλυψη του κώδικα (code coverage), δίνοντας τη δυνατότητα για αυτοματοποίηση των δοκιμών σε κάθε build.
- **Build Variants**: Που επιτρέπει τη δημιουργία πολλαπλών εκδόσεων της εφαρμογής από την ίδια βάση κώδικα, με διαφορετικές παραμέτρους και εξαρτήσεις για κάθε έκδοση.

3.3.4 Activities

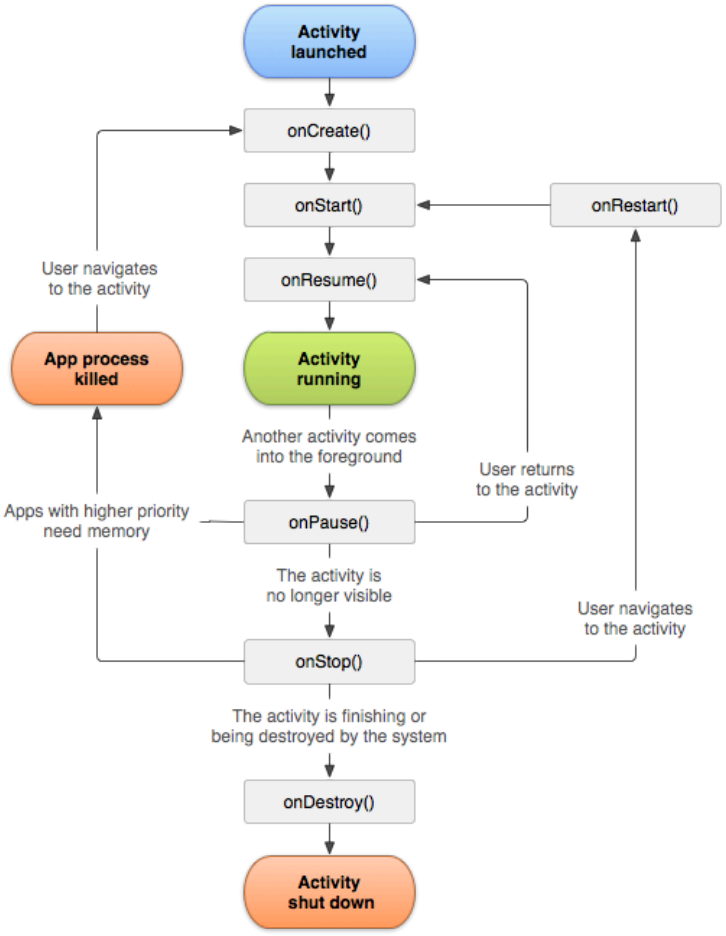
Τα Activities και τα Fragments είναι θεμελιώδη συστατικά του Android για τη δημιουργία και τη διαχείριση διεπαφών χρήστη (UI). Ενώ μοιράζονται πολλές ομοιότητες, έχουν διαφορετικούς ρόλους και χρήσεις. Ας αναλύσουμε τα Activities και τα Fragments, εξετάζοντας τη φύση τους, τον τρόπο λειτουργίας τους, και τις βασικές διαφορές τους.

Τα Activities είναι τα βασικά στοιχεία της διεπαφής χρήστη στο Android. Αντιπροσωπεύουν μεμονωμένες οθόνες ή λειτουργίες στην εφαρμογή. Συνήθως χρησιμοποιούνται για βασικές λειτουργίες της εφαρμογής, όπως η κύρια οθόνη, η οθόνη ρυθμίσεων ή η οθόνη εισόδου. Τα Activities είναι αυτά που συνδέονται με το AndroidManifest.xml, το οποίο καθορίζει ποια Activity ξεκινά όταν εκκινείται η εφαρμογή, ενώ για να επικοινωνούν μεταξύ τους, τα Activities χρησιμοποιούν Intents, τα οποία επιτρέπουν τη μετάδοση δεδομένων και την έναρξη νέων Activities.

Κάθε Activity έχει έναν καλά καθορισμένο κύκλο ζωής με συγκεκριμένα στάδια όπως [23] (βλ. **Εικόνα 3.8** *Κύκλος Ζωής ενός Activity*). :

- **onCreate()**: Αυτό είναι το πρώτο στάδιο του κύκλου ζωής, όπου το Activity δημιουργείται. Συνήθως, εδώ πραγματοποιούνται οι αρχικές ρυθμίσεις, όπως η σύνδεση με τα στοιχεία διεπαφής χρήστη, η διαμόρφωση των εξαρτήσεων, η ανάκτηση δεδομένων και η ανάκτηση και διαχείριση δεδομένων που σχετίζονται με την κατάσταση της εφαρμογής.
- **onStart()**: Το Activity γίνεται ορατό ωστόσο δεν είναι ακόμα έτοιμο για αλληλεπίδραση με τον χρήστη, αλλά μπορούν να πραγματοποιηθούν κάποιες προετοιμασίες για αυτό
- **onResume()**: Το Activity είναι έτοιμο για αλληλεπίδραση με τον χρήστη και αποτελεί το ενεργό Activity στο προσκήνιο. Αυτό είναι το σημείο στο οποίο εκτελείται το μεγαλύτερο μέρος της λειτουργικότητας της εφαρμογής καθώς οι αλλαγές που γίνονται εδώ έχουν άμεσο αντίκτυπο στη διεπαφή του χρήστη.
- **onPause()**: Το Activity χάνει την εστίαση αλλά παραμένει ορατό. Συνήθως, εδώ σταματούν ή αποθηκεύονται οι εργασίες που δεν πρέπει να συνεχίσουν όταν το Activity δεν είναι στο προσκήνιο. Αυτό το στάδιο είναι σύντομο και το Activity μπορεί να επιστρέψει στο προσκήνιο ή να προχωρήσει στο επόμενο στάδιο (onStop()).

- **onStop()**: Το Activity δεν είναι πλέον ορατό. Αυτό το στάδιο χρησιμοποιείται για την απελευθέρωση πόρων που δεν χρειάζονται όταν το Activity δεν είναι ορατό ενώ μπορούμε να αποθηκεύσουμε δεδομένα για να τα ανακτήσουμε αργότερα
- **onDestroy()**: Τέλος στο στάδιο OnDestroy() το Activity καταστρέφεται και αυτό μπορεί να συμβεί λόγω της ολοκλήρωσης του κύκλου ζωής του Activity ή αν το σύστημα χρειάζεται να ελευθερώσει πόρους.Είναι σημαντικό να "καθαρίζονται" οι πόροι και να διακόπτονται οι συνδέσεις για να αποφεύγονται οι διαρροές μνήμης.



Εικόνα 3.8. *Κύκλος Ζωής ενός Activity*

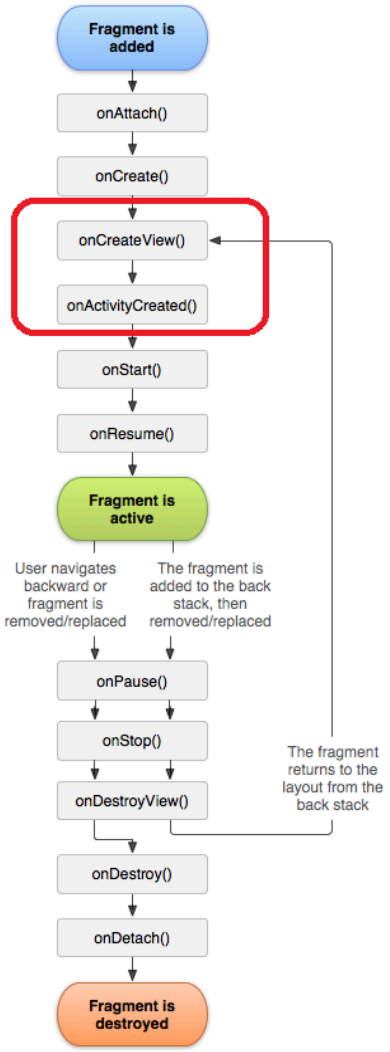
3.3.5 Fragments

Τα Fragments είναι μικρότερα, επαναχρησιμοποιήσιμα κομμάτια διεπαφής χρήστη. Συνήθως αποτελούν μέρος ενός Activity και μπορούν να συνδυαστούν για να δημιουργήσουν σύνθετες

διεπαφές. Χρησιμοποιούνται για τη δημιουργία επαναχρησιμοποιήσιμων και δυναμικών τμημάτων της διεπαφής χρήστη, καθώς επιτρέπουν στις εφαρμογές να έχουν ευέλικτη διάταξη, καθιστώντας τα ιδανικά για εφαρμογές που πρέπει να προσαρμόζονται σε διαφορετικά μεγέθη οθόνης, όπως σε tablets ή τηλέφωνα. Μπορούν να επικοινωνούν μεταξύ τους μέσω του Activity που τα φιλοξενεί, ενώ στην ανταλλαγή δεδομένων συμβάλλουν τα interfaces ή ViewModels.

Κάθε Fragment έχει τον δικό του κύκλο ζωής και ο κύκλος ζωής τους είναι παρόμοιος με αυτόν ενός Activity αλλά με μερικά επιπλέον στάδια [24] (βλ. **Εικόνα 3.9** *Κύκλος Ζωής ενός Fragment*).:

- **onAttach()**:Το Fragment συνδέεται με το Activity. Σε αυτό το στάδιο μπορούμε να αποκτήσουμε πρόσβαση σε πόρους και να κάνουμε αρχικές ρυθμίσεις.
- **onCreateView()**:Εδώ δημιουργείται και επιστρέφεται η οθόνη του Fragment. Χρησιμοποιείται για να δημιουργηθούν τα στοιχεία διεπαφής χρήστη που αποτελούν μέρος του Fragment.
- **onActivityCreated()**: Σε αυτή τη κατάσταση το Fragment έχει δημιουργηθεί πλήρως και συνδέεται με το Activity. Μπορούν να γίνουν οι εργασίες που απαιτούν την πλήρη δημιουργία του Activity.
- **onDestroyView()**: Η οθόνη του Fragment καταστρέφεται και διαγράφονται οι πόροι που σχετίζονται με τη αυτήν.
- **onDetach()**: Το Fragment αποσυνδέεται από το Activity



Εικόνα 3.9. *Κύκλος Ζωής ενός Fragment*

3.3.6 ViewModel

Το ViewModel είναι ένα βασικό στοιχείο της αρχιτεκτονικής MVVM (Model-View-ViewModel) και αποτελεί μέρος του Android Jetpack, που στοχεύει στην απλοποίηση και την βελτίωση της δομής των εφαρμογών Android. Το ViewModel εξυπηρετεί ως ενδιάμεσο μεταξύ του Model (δεδομένα και επιχειρηματική λογική) και του View (διεπαφή χρήστη), παρέχοντας μια σειρά από πλεονεκτήματα και δυνατότητες που κάνουν την ανάπτυξη εφαρμογών πιο οργανωμένη και ευέλικτη [27].

Τα ViewModels λειτουργούν ως ενδιάμεσος μεταξύ των Views και του Domain Layer. Διαχειρίζονται την κατάσταση της διεπαφής χρήστη και επικοινωνούν με τα Use Cases για την εκτέλεση επιχειρηματικής λογικής. Ο ρόλος των ViewModels είναι να διατηρούν την κατάσταση που σχετίζεται με μια οθόνη και παρέχουν δεδομένα στα Views, ώστε να μπορούν να παρουσιαστούν στον χρήστη. Επίσης, χειρίζονται συμβάντα από τα Views και κατευθύνουν τις ενέργειες στο Domain Layer. Η χρήση τους βοηθά στη διαχωριστικότητα των ανησυχιών, καθιστώντας το Presentation

Layer πιο ευέλικτο και δοκιμάσιμο. Επίσης, υποστηρίζει την επιβίωση της κατάστασης κατά την αναδημιουργία των δραστηριοτήτων (activities) και άλλων στοιχείων του UI.

Βασικές Λειτουργίες του αποτελούν:

- **Διατήρηση Κατάστασης**: Το ViewModel διατηρεί την κατάσταση της εφαρμογής σε περιπτώσεις αλλαγών του κύκλου ζωής, όπως η αλλαγή προσανατολισμού της οθόνης ή η αναδημιουργία ενός Activity/Fragment. Αυτό σημαίνει ότι τα δεδομένα παραμένουν διαθέσιμα χωρίς να χρειάζεται να αποθηκεύονται ή να ανακτώνται συνεχώς.
- **Αποσύνδεση του View από το Model**: Το ViewModel επιτρέπει τη διαχωριστικότητα μεταξύ του View και του Model, μειώνοντας τη σύζευξη και καθιστώντας τον κώδικα πιο δοκιμάσιμο και ευέλικτο. Αυτό διευκολύνει την αντικατάσταση του View ή του Model χωρίς να επηρεάζονται άλλες περιοχές της εφαρμογής.
- **Επικοινωνία με τα View Components**: Το ViewModel χρησιμοποιεί συνήθως παρατηρήσιμα δεδομένα (LiveData) για να διατηρεί ενημερωμένα τα View Components (Activities, Fragments). Αυτή η προσέγγιση επιτρέπει την αντιδραστική ενημέρωση των Views χωρίς να χρειάζεται άμεση παρέμβαση από το ViewModel.
- **Διαχείριση Δεδομένων**: Το ViewModel συχνά λειτουργεί ως διαμεσολαβητής μεταξύ του View και των πηγών δεδομένων (όπως βάσεις δεδομένων ή δικτυακές κλήσεις). Μπορεί να ανακτά δεδομένα, να τα επεξεργάζεται και να τα παρέχει στα Views.
- **Λογική Εφαρμογής**: Αν και δεν περιέχει την επιχειρηματική λογική (η οποία ανήκει στο Domain Layer στη Clean Architecture), το ViewModel μπορεί να χειρίζεται την επιμέρους λογική της διεπαφής χρήστη, όπως την επικύρωση δεδομένων, τη λήψη αποφάσεων βάσει χρήστη, κ.λπ.
- **Παρακολούθηση Κατάστασης**: Το ViewModel μπορεί να παρακολουθεί και να διατηρεί την κατάσταση που σχετίζεται με το View, όπως ποιο στοιχείο είναι επιλεγμένο, ποια δεδομένα είναι διαθέσιμα, κ.λπ.
- **LiveData**: Το ViewModel συχνά χρησιμοποιεί LiveData, έναν τύπο παρατηρήσιμων δεδομένων που επιτρέπει την αντιδραστική ενημέρωση των Views. Με το LiveData, μπορείτε να παρακολουθείτε τις αλλαγές στα δεδομένα και να ενημερώνεται αυτόματα το UI όταν υπάρχουν αλλαγές.
- **Επιβίωση κατά την αναδημιουργία (Recreation Survival)**: Το ViewModel παραμένει ζωντανό κατά την αναδημιουργία του Activity ή του Fragment, επιτρέποντας στη διεπαφή χρήστη να διατηρεί την κατάστασή της χωρίς απώλεια δεδομένων.
- **Ενσωμάτωση με άλλες βιβλιοθήκες Jetpack**: Το ViewModel λειτουργεί καλά με άλλες βιβλιοθήκες του Android Jetpack, όπως το Room (για βάσεις δεδομένων) και το WorkManager (για εργασίες στο παρασκήνιο), καθιστώντας το μέρος μιας ολοκληρωμένης αρχιτεκτονικής.

3.3.7 Jetpack Compose

Η ανάπτυξη εφαρμογών Android έχει εξελιχθεί σημαντικά με την πάροδο του χρόνου, ειδικά με την εισαγωγή του Jetpack Compose. Πριν από την ύπαρξη του Jetpack Compose, οι προγραμματιστές Android χρησιμοποιούσαν παραδοσιακές μεθόδους για τη δημιουργία διεπαφών χρήστη και την ανάπτυξη εφαρμογών. Κάθε οθόνη ή στοιχείο της διεπαφής χρήστη είχε τη δική του διάταξη, η οποία οριζόταν μέσω XML αρχείων, όπως *activity_main.xml*, ενώ για τη δημιουργία δυναμικών διεπαφών, οι προγραμματιστές χρησιμοποιούσαν προγραμματισμό Java ή Kotlin για να τροποποιούν τα στοιχεία της διάταξης κατά τη διάρκεια εκτέλεσης. Όμως η χρήση XML για τον καθορισμό διατάξεων είχε ορισμένα μειονεκτήματα. Για παράδειγμα, οι δυναμικές αλλαγές στη διεπαφή χρήστη ήταν πιο περίπλοκες, και η διαχείριση αλληλεπιδράσεων μεταξύ διαφόρων στοιχείων απαιτούσε περισσότερο κώδικα.Η επικοινωνία μεταξύ των στοιχείων της διάταξης και της λογικής της εφαρμογής (Activities και Fragments) συχνά περιλάμβανε πολλές αναφορές και μηχανισμούς όπως το "findViewById()", που μπορούσαν να καταστούν περίπλοκοι σε μεγάλες εφαρμογές.

Για να βοηθήσει τους προγραμματιστές να αντιμετωπίσουν μερικά από τα προβλήματα που σχετίζονται με τις περίπλοκες διατάξεις και την αρχιτεκτονική εφαρμογών, η Google παρουσίασε τα Android Architecture Components. Αυτά περιλάμβαναν βιβλιοθήκες όπως ViewModel, LiveData, και Room, οι οποίες βοήθησαν στη βελτίωση της δοκιμαστικότητας και της συντηρησιμότητας των εφαρμογών. Βέβαια με την εισαγωγή του Jetpack Compose, η ανάπτυξη εφαρμογών Android γνώρισε μια σημαντική αλλαγή προς μια πιο δηλωτική και δυναμική προσέγγιση.

Το Jetpack Compose είναι ένα μοντέρνο πλαίσιο (framework) για την ανάπτυξη διεπαφών χρήστη (UI) για εφαρμογές Android [12]. Αντικαθιστά το παραδοσιακό σύστημα βασισμένο σε XML με μια δηλωτική προσέγγιση, επιτρέποντας στους προγραμματιστές να δημιουργούν δυναμικές και ευέλικτες διεπαφές με λιγότερο και πιο κατανοητό κώδικα. Αποτελεί ένα μέρος της οικογένειας εργαλείων Jetpack από την Google για την ανάπτυξη εφαρμογών Android. Πρόκειται για ένα δηλωτικό πλαίσιο UI που επιτρέπει στους προγραμματιστές να δημιουργούν UI με βάση κώδικα Kotlin, αντί για XML. Αυτό σημαίνει ότι οι προγραμματιστές μπορούν να καθορίσουν το περιεχόμενο και την εμφάνιση του UI μέσω κώδικα, χωρίς να χρειάζεται να χρησιμοποιούν ξεχωριστά αρχεία XML.

Το Jetpack Compose βασίζεται σε μερικές βασικές αρχές που το καθιστούν ισχυρό και ευέλικτο:

1. **Δηλωτική Προσέγγιση**: Σε αντίθεση με την παραδοσιακή προσέγγιση όπου ο κώδικας αλλάζει το UI μέσω χειριστών (manipulators), το Jetpack Compose επιτρέπει στους προγραμματιστές να καθορίσουν την εμφάνιση του UI με τρόπο που να αντιδρά αυτόματα στις αλλαγές των δεδομένων.
2. **Εργαλεία και Εξαρτήματα**: Το Jetpack Compose παρέχει έναν πλούσιο κατάλογο στοιχείων UI (widgets), όπως κουμπιά, λίστες, διατάξεις (layouts), καθώς και προηγμένες λειτουργίες όπως κινούμενα γραφικά (animations) και μεταβάσεις (transitions).
3. **Συμβατότητα με Υπάρχουσες Τεχνολογίες**: Το Jetpack Compose είναι σχεδιασμένο ώστε να είναι συμβατό με το υπάρχον οικοσύστημα Android, επιτρέποντας την ενσωμάτωση με

XML-based UIs, την πρόσβαση σε Android APIs και την υποστήριξη της αρχιτεκτονικής ViewModel.

3.3.8 Dependency Injection

Η **Dependency Injection (DI)** είναι μια αρχιτεκτονική πρακτική στον σχεδιασμό λογισμικού που προάγει τη χαλαρή σύζευξη (loose coupling) μεταξύ των εξαρτημένων συστατικών (components) ενός συστήματος. Το DI επιτρέπει στα συστατικά να εξαρτώνται από άλλες υπηρεσίες ή αντικείμενα χωρίς να δημιουργούν άμεσες εξαρτήσεις. Αυτό γίνεται συνήθως παρέχοντας τις εξαρτήσεις από εξωτερικούς παράγοντες, αντί τα ίδια τα συστατικά να δημιουργούν ή να διαχειρίζονται τις εξαρτήσεις τους [21].

Σε ένα σύστημα χωρίς Dependency Injection, τα συστατικά δημιουργούν τις δικές τους εξαρτήσεις απευθείας μέσα στον κώδικα. Αυτό οδηγεί σε σκληρή σύζευξη, καθιστώντας δύσκολη τη συντήρηση, τον έλεγχο, ή την επαναχρησιμοποίηση του κώδικα. Με τη χρήση της Dependency Injection, οι εξαρτήσεις εισάγονται (injected) στα συστατικά από εξωτερικές πηγές, όπως ένας "injector" ή ένα "container". Αυτός ο εξωτερικός παράγοντας αναλαμβάνει την ευθύνη της διαχείρισης των εξαρτήσεων και της παροχής τους στα συστατικά που τις χρειάζονται. Υπάρχουν τρεις κύριοι τρόποι DI:

1. **Εισαγωγή μέσω Κατασκευαστή (Constructor Injection)**: Οι εξαρτήσεις παρέχονται στο αντικείμενο κατά τη δημιουργία του, μέσω του κατασκευαστή. Αυτός ο τύπος DI είναι συχνά προτιμότερος καθώς εξασφαλίζει ότι το αντικείμενο έχει όλες τις εξαρτήσεις του όταν δημιουργείται.
2. **Εισαγωγή μέσω Μεθόδου (Method Injection)**: Οι εξαρτήσεις παρέχονται μέσω μιας ειδικής μεθόδου. Αυτός ο τρόπος χρησιμοποιείται συχνά για εξαρτήσεις που μπορούν να αλλάξουν κατά τη διάρκεια ζωής του αντικειμένου.
3. **Εισαγωγή μέσω Πεδίου (Field Injection)**: Οι εξαρτήσεις εισάγονται απευθείας στα πεδία του αντικειμένου, συνήθως μέσω ανακλαστικότητας (reflection). Ενώ είναι γρήγορη μέθοδος, δεν είναι πάντα τόσο ασφαλής όσο η εισαγωγή μέσω κατασκευαστή.

Οφέλη Dependency Injection

Τα οφέλη της Dependency Injection (DI) είναι πολλά και επηρεάζουν θετικά διάφορες πτυχές της ανάπτυξης λογισμικού. Από τη βελτίωση της ευελιξίας μέχρι την αύξηση της επαναχρησιμοποίησης του κώδικα, η χρήση της DI έχει αποδειχθεί εξαιρετικά χρήσιμη σε πολλούς τομείς. Ας δούμε αναλυτικά τα οφέλη της DI:

1. Χαλαρή Σύζευξη (Loose Coupling)

Η DI μειώνει τις εξαρτήσεις μεταξύ των διαφόρων συστατικών του λογισμικού. Αντί να δημιουργούν τα ίδια τα συστατικά τις εξαρτήσεις τους, αυτές παρέχονται από εξωτερικές πηγές, όπως ένας "injector" ή ένα "container". Αυτός ο χαλαρός δεσμός επιτρέπει στα συστατικά να αλλάζουν, να αντικαθίστανται ή να επαναχρησιμοποιούνται χωρίς να επηρεάζουν άλλα μέρη της εφαρμογής.

2. Ευκολία Δοκιμών (Testability)

Με τη DI, οι εξαρτήσεις μπορούν να αντικατασταθούν με mock αντικείμενα ή stubs κατά τη διάρκεια των δοκιμών. Αυτό επιτρέπει τη δημιουργία πιο απομονωμένων δοκιμών (unit tests) για τα διάφορα συστατικά, διευκολύνοντας τη διαδικασία ελέγχου και εντοπισμού σφαλμάτων. Η δυνατότητα αυτή είναι ιδιαίτερα σημαντική για την εξασφάλιση της ποιότητας και της αξιοπιστίας του λογισμικού.

3. Καλύτερη Συντήρηση (Maintainability)

Η DI προάγει μια πιο καθαρή και ευέλικτη αρχιτεκτονική, καθιστώντας τον κώδικα πιο εύκολο στη συντήρηση. Καθώς τα συστατικά έχουν ξεκάθαρες εξαρτήσεις που παρέχονται εξωτερικά, είναι πιο εύκολο να εντοπίσετε και να διορθώσετε προβλήματα, να κάνετε αναβαθμίσεις ή να προσθέσετε νέα χαρακτηριστικά χωρίς να χρειάζεται να ανασχεδιάσετε ολόκληρο το σύστημα.

4. Επαναχρησιμοποίηση Κώδικα (Reusability)

Η DI επιτρέπει την επαναχρησιμοποίηση συστατικών σε διαφορετικά περιβάλλοντα ή περιπτώσεις χρήσης. Καθώς τα συστατικά δεν εξαρτώνται άμεσα από άλλα μέρη της εφαρμογής, μπορούν να χρησιμοποιηθούν σε διάφορα σενάρια, προωθώντας την επαναχρησιμοποίηση και μειώνοντας την πολυπλοκότητα.

5. Βελτιωμένη Επεκτασιμότητα (Scalability)

Η αρχιτεκτονική που προάγει η DI διευκολύνει την επέκταση της εφαρμογής. Οι εξωτερικές εξαρτήσεις μπορούν να προσαρμοστούν ή να επεκταθούν χωρίς να επηρεαστούν τα κύρια συστατικά του λογισμικού. Αυτό διευκολύνει την προσθήκη νέων χαρακτηριστικών ή υπηρεσιών με λιγότερη πολυπλοκότητα.

6. Αυξημένη Ευελιξία (Flexibility)

Η DI επιτρέπει την ευελιξία στο σχεδιασμό και την αρχιτεκτονική του λογισμικού. Μας δίνει την δυνατότητα να αλλάξουμε την υλοποίηση μιας εξάρτησης χωρίς να επηρεάσουμε τα συστατικά που τη χρησιμοποιούν. Αυτό διευκολύνει την υιοθέτηση νέων τεχνολογιών ή προσεγγίσεων καθώς η εφαρμογή εξελίσσεται.

7. Προώθηση Καλών Πρακτικών (Promotion of Best Practices)

Η χρήση της DI συχνά συμβαδίζει με άλλες αρχιτεκτονικές πρακτικές όπως το SOLID και το Clean Architecture, προωθώντας τον καθαρό κώδικα και τον διαχωρισμό ευθυνών. Αυτό δημιουργεί μια κουλτούρα ανάπτυξης που ενθαρρύνει την ποιότητα και τη συνέπεια στον κώδικα.

Δημοφιλής Βιβλιοθήκες DI

Dagger: Είναι μια από τις πιο γνωστές βιβλιοθήκες Dependency Injection για Android. Αναπτύχθηκε αρχικά από την Square και τώρα συντηρείται από την Google. Είναι ιδιαίτερα δημοφιλής λόγω της απόδοσης και της ευελιξίας του.Χρησιμοποιεί γεννήτριες κώδικα κατά τη διαδικασία μεταγλώττισης (compile-time code generation), που οδηγεί σε γρηγορότερη εκτέλεση και αποφυγή προβλημάτων

χρόνου εκτέλεσης. Χρησιμοποιεί annotations για τον ορισμό εξαρτήσεων, μονάδων (modules), και συστατικών (components). Αυτό το κάνει ισχυρό καθώς προσφέρει ισχυρή απόδοση και σταθερότητα λόγω της γεννήτριας κώδικα. Είναι συμβατό με τις πρακτικές της Google και χρησιμοποιείται ευρέως σε μεγάλες εφαρμογές ωστόσο μπορεί να καθιστά περίπλοκο κατά την εκμάθηση του από αρχάριους.

Hilt: Είναι μια επέκταση του Dagger που αναπτύχθηκε από την Google για να απλοποιήσει τη χρήση της DI στο Android. Έχει σχεδιαστεί για να κάνει τη διαδικασία πιο προσιτή στους προγραμματιστές και να μειώσει τον κώδικα "boilerplate". Το Hilt αυτοματοποιεί πολλά από τα σημεία που απαιτούσαν χειροκίνητη ρύθμιση στο Dagger, καθιστώντας την εισαγωγή εξαρτήσεων πιο απλή και ευκολότερη. Παρέχει έτοιμα scopes για συστατικά όπως Activities και Fragments, απλοποιώντας τη δημιουργία εξαρτήσεων σε Android εφαρμογές. Χρησιμοποιεί annotations για να ορίσει τα σημεία όπου θα γίνει η εισαγωγή των εξαρτήσεων. Προσφέρει απλοποιημένη εμπειρία, ειδικά για προγραμματιστές που δεν έχουν μεγάλη εμπειρία με το Dagger. Παρέχει ολοκληρωμένη υποστήριξη για τη δομή μιας Android εφαρμογής.

Koin: Είναι μια δημοφιλής βιβλιοθήκη για Dependency Injection (DI) στο Android, που βασίζεται στη γλώσσα Kotlin. Το όνομα "Koin" προέρχεται από το "Kotlin + Injection" και αντικατοπτρίζει τον στόχο της να παρέχει μια διαφανή και απλή λύση για DI στη γλώσσα Kotlin. Έχει σχεδιαστεί για να είναι απλή στη χρήση και να προσφέρει μια εύκολη λύση για την εισαγωγή εξαρτήσεων σε Android εφαρμογές. Η Koin χρησιμοποιεί μια προσέγγιση με "Service Locator", όπου οι εξαρτήσεις ορίζονται σε ένα κεντρικό σημείο και στη συνέχεια μπορούν να εισαχθούν σε διάφορα μέρη της εφαρμογής. Σε αντίθεση με το Dagger, που χρησιμοποιεί μεταγλωττιστή για την επίλυση των εξαρτήσεων κατά τη μεταγλώττιση, η Koin είναι δυναμική και χρησιμοποιεί DSL (Domain Specific Language) της Kotlin για να καθορίσει πώς εισάγονται οι εξαρτήσεις.

3.3.9 Coroutines

Οι "coroutines" είναι ένα προγραμματιστικό μοτίβο που επιτρέπει τη δημιουργία λειτουργιών ή μεθόδων που μπορούν να διακόπτονται και να επαναλαμβάνονται αργότερα, επιτρέποντας τη διακοπή της εκτέλεσης σε συγκεκριμένα σημεία χωρίς να χάνεται το πλαίσιο της συνάρτησης. Αυτό το χαρακτηριστικό τις καθιστά ιδανικές για διάφορα σενάρια, όπως ασύγχρονος προγραμματισμός, καταμερισμός χρόνου σε διεργασίες, και δημιουργία γεννητριών [22].

Οι coroutines έχουν την ικανότητα να "διατηρούν" την κατάστασή τους όταν διακόπτονται, επιτρέποντας την επανάληψη από το σημείο διακοπής χωρίς να επανα-δημιουργούνται. Σε αντίθεση με τις παραδοσιακές συναρτήσεις, που αρχίζουν και τελειώνουν με μια ενιαία ροή εκτέλεσης, οι coroutines μπορούν να "ανασταλούν" (suspend) και να "συνεχιστούν" (resume) ανάλογα με τις ανάγκες του προγράμματος. Χρησιμοποιούνται για τη διαχείριση ασύγχρονων εργασιών χωρίς τη χρήση πολλαπλών νημάτων [14]. Αυτό είναι χρήσιμο σε εφαρμογές που χρειάζονται γρήγορη ανταπόκριση ή πρέπει να χειριστούν μεγάλο όγκο δεδομένων χωρίς να μπλοκάρουν το κύριο νήμα εκτέλεσης. Επιτρέπουν τη δημιουργία γεννητριών, οι οποίες μπορούν να παράγουν δεδομένα κομμάτι-κομμάτι, αντί να επιστρέφουν όλα τα δεδομένα ταυτόχρονα. Ενώ μπορούν να χρησιμοποιηθούν για καταμερισμό διεργασιών, επιτρέποντας σε πολλαπλά κομμάτια κώδικα να λειτουργούν σε "χρονικές φέτες" μέσα στο ίδιο νήμα.

Σχετικά παραδείγματα γλωσσών προγραμματισμού που αξιοποιούν τις coroutines:

1. **Python**: Οι coroutines είναι ενσωματωμένες στη γλώσσα μέσω της χρήσης του *async* και *await*, επιτρέποντας ασύγχρονο προγραμματισμό με ευκολία. Αυτό διευκολύνει τη διαχείριση ασύγχρονων εργασιών, όπως κλήσεις δικτύου ή I/O.
2. **Kotlin**: Υποστηρίζει coroutines ως τρόπο διαχείρισης ασύγχρονων εργασιών σε Android εφαρμογές χωρίς την ανάγκη πολυπλοκότητας που σχετίζεται με τα νήματα.
3. **C#**: Εισήγαγε τη χρήση του *async* και *await* για ασύγχρονες λειτουργίες, κάνοντας την ανάπτυξη εφαρμογών πιο αποτελεσματική και εύκολη στην ανάγνωση.

Στην Kotlin, οι coroutines είναι ένας τρόπος εκτέλεσης κώδικα με μη μπλοκαριστικό τρόπο, επιτρέποντας τον καταμερισμό εργασιών στο ίδιο νήμα ή σε διαφορετικά νήματα χωρίς την πολυπλοκότητα που σχετίζεται με την παραδοσιακή διαχείριση νημάτων. Μπορείτε να φανταστείτε τις coroutines ως μια βελτιωμένη έκδοση των παραδοσιακών γεννητριών (generators), όπου μπορείτε να διακόψετε και να συνεχίσετε την εκτέλεση όπως απαιτείται. Μερικά από τα πλεονεκτήματα χρήσης coroutines είναι :

- **Ευκολία**: Με τη χρήση των λέξεων-κλειδιών *suspend* και *launch*, οι coroutines γίνονται εύκολες στην υλοποίηση και στη συντήρηση.
- **Μη μπλοκαριστική συμπεριφορά**: Οι coroutines επιτρέπουν τον ασύγχρονο προγραμματισμό χωρίς να μπλοκάρουν το κύριο νήμα, καθιστώντας τες ιδανικές για εργασίες που περιλαμβάνουν εισόδους/εξόδους, δικτύωση, ή άλλες ενέργειες που απαιτούν χρόνο.
- **Διαχείριση πόρων**: Οι coroutines είναι πιο αποδοτικές από τα νήματα σε επίπεδο πόρων, καθώς μπορούν να τρέχουν στο ίδιο νήμα χωρίς να απαιτούν το γενικό κόστος δημιουργίας και συντήρησης πολλαπλών νημάτων.
- **Ευελιξία**: Επιτρέπουν εύκολη μετάβαση μεταξύ διαφόρων περιβαλλόντων (contexts), όπως στο πλαίσιο μιας Android εφαρμογής, από το κύριο νήμα σε νήμα για εργασίες στο παρασκήνιο.

Βασικές Λέξεις-κλειδιά και Δομές

- **suspend**: Δηλώνει ότι μια συνάρτηση μπορεί να ανασταλεί (suspend) και να συνεχιστεί αργότερα. Αυτό επιτρέπει τη μη μπλοκαριστική συμπεριφορά.
- **delay**: Με αυτή την δήλωση μπορούμε να εισάγουμε καθυστέρηση στην εκτέλεση κάποιων λειτουργιών εφόσον αυτό επιθυμούμε
- **launch**: Ξεκινά μια νέα coroutine χωρίς να μπλοκάρει το καλούν νήμα. Είναι χρήσιμο για εργασίες που μπορούν να τρέξουν στο παρασκήνιο.
- **async**: Ξεκινά μια coroutine που επιστρέφει ένα *Deferred* αντικείμενο, το οποίο μπορεί να χρησιμοποιηθεί για να ανακτηθεί το αποτέλεσμα όταν ολοκληρωθεί η coroutine.
- **withContext**: Μεταβαίνει σε ένα διαφορετικό περιβάλλον εκτέλεσης, όπως από το κύριο νήμα σε ένα νήμα εργασίας.

- **runBlocking**: Μας δίνει την δυνατότητα να μπλοκάρουμε το main thread, προκειμένου να εκτελεστεί πρώτα το τμήμα κώδικα που εμπεριέχεται μέσα στα brackets ({})

3.3.10 Coroutine Scopes

Οι "coroutine scopes" στην Kotlin είναι ένας μηχανισμός που καθορίζει το πλαίσιο ή το εύρος όπου "ζουν" και εκτελούνται οι coroutines. Ένα coroutine scope καθορίζει τη διάρκεια ζωής των coroutines που ανήκουν σε αυτό, και αν καταργηθεί το scope, όλες οι coroutines που περιέχει ακυρώνονται. Αυτό επιτρέπει τη διαχείριση του κύκλου ζωής των coroutines και εμποδίζει διαρροές μνήμης ή ανεπιθύμητη συμπεριφορά λόγω coroutines που τρέχουν στο παρασκήνιο χωρίς έλεγχο. Τα coroutine scopes είναι σημαντικά γιατί μας επιτρέπουν να ελέγχουμε τη διάρκεια ζωής των coroutines σε σχέση με το περιβάλλον όπου εκτελούνται. Αυτός ο έλεγχος βοηθά στη διαχείριση πόρων και στην αποφυγή σφαλμάτων ή διαρροών λόγω coroutines που συνεχίζουν να τρέχουν όταν δεν χρειάζεται.

Οι τέσσερις βασικοί και πιο σημαντικοί τύποι coroutine scopes στην Kotlin είναι:

GlobalScope

Το GlobalScope στην Kotlin είναι ένα προεπιλεγμένο coroutine scope που προορίζεται για coroutines οι οποίες χρειάζονται διάρκεια ζωής ανεξάρτητη από άλλα αντικείμενα ή τοπικά context. Δεν σχετίζεται με κανέναν συγκεκριμένο κύκλο ζωής, όπως συμβαίνει με τα scope που συνδέονται με Activity ή Fragment σε Android. Αυτό έχει σαν αποτέλεσμα οι coroutines που εκτελούνται στο GlobalScope συνεχίζουν να τρέχουν έως ότου ολοκληρωθούν ή ακυρωθούν με μη αυτόματο τρόπο. Αυτό μπορεί να οδηγήσει σε coroutines που συνεχίζουν να τρέχουν ακόμα και όταν το υπόλοιπο μέρος της εφαρμογής έχει τελειώσει. Συνήθως εκτελούνται σε κοινόχρηστα threads, χρησιμοποιώντας τους προεπιλεγμένους dispatchers της Kotlin, όπως το Dispatchers.Default ή το Dispatchers.IO.

Ενώ το GlobalScope είναι εύκολο στη χρήση, μπορεί να οδηγήσει σε προβλήματα όπως:

1. **Διαρροές Μνήμης**: Αν χρησιμοποιηθεί σε περιβάλλοντα όπου ο κύκλος ζωής είναι σημαντικός, όπως σε Android εφαρμογές, υπάρχει κίνδυνος οι coroutines να συνεχίσουν να τρέχουν αφού η δραστηριότητα ή το πλαίσιο έχει καταστραφεί.
2. **Χειρισμός Σφαλμάτων**: Με το GlobalScope, οι coroutines δεν συνδέονται με συγκεκριμένο πλαίσιο, καθιστώντας τον χειρισμό σφαλμάτων λιγότερο προβλέψιμο. Αν μια coroutine στο GlobalScope αποτύχει, πρέπει να διαχειριστούμε τα σφάλματα με μη αυτόματο τρόπο.
3. **Ανεξέλεγκτη Εκτέλεση**: Καθώς δεν υπάρχει αυτόματη διαχείριση διάρκειας ζωής, υπάρχει κίνδυνος coroutines που εκτελούνται στο GlobalScope να συνεχίσουν να τρέχουν ακόμα και όταν δεν είναι πλέον απαραίτητες.

Φυσικά υπάρχουν ορισμένες περιπτώσεις που καθιστούν το GlobalScope ιδανικό αν όχι απαραίτητο όπως σε περιπτώσεις που χρειάζεται να εκτελέσουμε μακροχρόνιες εργασίες που πρέπει να συνεχίσουν να τρέχουν για όσο τρέχει η εφαρμογή, για εργασίες που δεν συνδέονται με συγκεκριμένο Activity ή Fragment και πρέπει να συνεχίσουν ακόμα και μετά το κλείσιμο του γραφικού περιβάλλοντος και για εργασίες που αφορούν παγκόσμια δεδομένα ή άλλες λειτουργίες που πρέπει να τρέχουν ανεξάρτητα από το υπόλοιπο της εφαρμογής.

ViewModelScope

Το "ViewModel scope" είναι ένα coroutine scope στην Kotlin που συνδέεται με τη διάρκεια ζωής ενός ViewModel στην ανάπτυξη εφαρμογών Android. Όταν το ViewModel καταστραφεί (συνήθως όταν το σχετικό Activity ή Fragment καταστραφεί ή αλλάξει διάταξη), όλες οι coroutines που εκτελούνται στο ViewModel scope ακυρώνονται αυτόματα. Αυτός ο έλεγχος εισάγει μια σημαντική δυνατότητα για την εκτέλεση ασύγχρονων εργασιών σε ViewModels, διασφαλίζοντας ότι οι εργασίες αυτές ακυρώνονται όταν καταστραφεί το ViewModel. Αυτό βοηθά στην αποφυγή διαρροών μνήμης και εξασφαλίζει ότι οι ασύγχρονες εργασίες δε συνεχίζουν να εκτελούνται μετά τον κύκλο ζωής του ViewModel.

Το ViewModel scope αξιοποιείται σε διάφορα σενάρια στην ανάπτυξη εφαρμογών Android, όπως στη λήψη δεδομένων από δίκτυο κατά την εκτέλεση ασύγχρονων εργασιών στη λήψη δεδομένων χωρίς να μπλοκάρετε το κύριο νήμα, στις εργασίες επεξεργασίας δεδομένων που μπορεί να διαρκέσουν αρκετό χρόνο αλλά και στην διαχείριση βάσεων δεδομένων κατά την αλληλεπίδραση με αυτές ή και άλλες τοπικές λειτουργίες εισόδου/εξόδου.

Ωστόσο προκειμένου να έχουμε το μέγιστο επιθυμητό αποτέλεσμα με την χρήση του είναι απαραίτητο να διαχειριζόμαστε κατάλληλα τα σφάλματα μέσα στις coroutines που εκτελούνται σε αυτό. Να γίνεται περιορισμός των εργασιών διότι καθώς το ViewModel Scope ακυρώνεται όταν το ViewModel καταστραφεί, είναι σημαντικό να μην τρέχουν παράλληλα υπερβολικά πολλές coroutines για να αποφεύγεται η περιττή πολυπλοκότητα. Τέλος και αρκετά σημαντικός παράγοντας είναι η χρήση του κατάλληλου Dispatcher που αναλύονται αργότερα σε αυτό το κεφάλαιο προκειμένου να εκτελούνται οι coroutines στα σωστά νήματα (threads).

LifecycleScope

Το "LifecycleScope" είναι ένα coroutine scope στην Kotlin που σχετίζεται με το κύκλο ζωής των Android component, όπως τα Activity, Fragment, ή άλλα LifecycleOwner. Αυτό το scope μας επιτρέπει να εκτελούμε coroutines με διάρκεια ζωής συνδεδεμένη με τον κύκλο ζωής του Android component, διασφαλίζοντας ότι οι coroutines ακυρώνονται αυτόματα όταν το component καταστραφεί ή αλλάξει κατάσταση.

Το LifecycleScope προσφέρει πολλά πλεονεκτήματα, κυρίως σε σχέση με τη διαχείριση κύκλου ζωής και την ασφάλεια των ασύγχρονων εργασιών όπως στη διαχείριση κύκλου ζωής όπου οι coroutines που εκτελούνται στο LifecycleScope ακυρώνονται αυτόματα όταν ο κύκλος ζωής του component φτάσει σε μια τελική κατάσταση, όπως όταν ένα Activity ή ένα Fragment καταστρέφεται. Στην ασφάλεια από διαρροές μνήμης δεδομένου ότι οι coroutines ακυρώνονται όταν το component καταστραφεί, μειώνεται ο κίνδυνος διαρροών μνήμης που σχετίζονται με coroutines που τρέχουν στο παρασκήνιο μετά την καταστροφή του component. Και στην ευκολία χρήσης καθώς το LifecycleScope κάνει εύκολη την εκκίνηση coroutines μέσα στα Android component χωρίς να χρειάζεται να ανησυχούμε για τη διαχείριση κύκλου ζωής.

CoroutineScope

Το "CoroutineScope" στην Kotlin είναι ένα βασικό στοιχείο του συστήματος coroutines, παρέχοντας ένα πλαίσιο για τη δημιουργία, εκτέλεση και διαχείριση των coroutines. Είναι ένα σημείο αναφοράς για τις coroutines και καθορίζει τη διάρκεια ζωής τους, εξασφαλίζοντας ότι ακυρώνονται όταν το scope δεν είναι πλέον ενεργό. Το CoroutineScope μπορεί να θεωρηθεί ως μια συλλογή coroutines που μοιράζονται έναν κοινό κύκλο ζωής. Είναι ίσως η πιο βασική coroutine που χρησιμοποιείται σε επίπεδο UI μέσα στα composables προκειμένου να εκτελεστεί μια σειρά από εργασίες που επηρεάζουν το UI.

Κάθε CoroutineScope έχει κάποιες βασικές λειτουργίες όπως:

1. **Job**: Το οποίο καθορίζει τη διάρκεια ζωής και την κατάσταση της coroutine. Μας δίνει την δυνατότητα να το χρησιμοποιήσουμε για να ακυρώσουμε το scope, διακόπτοντας όλες τις coroutines που συνδέονται με αυτό. Ακόμα μπορούμε να το χρησιμοποιήσουμε για να ομαδοποιήσετε τις coroutines και να τις ακυρώσουμε όλες μαζί είτε να διαχειριστούμε τους πόρους των coroutines ακυρώνοντας εργασίες που δεν χρειάζονται πλέον.
2. **Dispatcher**: Που όπως προαναφέραμε και αναλύεται παρακάτω είναι πολύ σημαντικό για τον έλεγχο του νήματος πάνω στο οποίο θα εκτελεστεί.
3. **CoroutineName**: Την προσθήκη ενός ονόματος στις coroutines για ευκολότερη παρακολούθηση και διαχείριση.
4. **CoroutineExceptionHandler**: Που επιτρέπει τον χειρισμό εξαιρέσεων που μπορεί να προκύψουν κατά την εκτέλεση των coroutines.

3.3.11 Dispatcher

Στην Kotlin, ο όρος "Dispatcher" αναφέρεται στον μηχανισμό που καθορίζει σε ποιο νήμα ή ομάδα νημάτων εκτελείται μια coroutine. Οι Dispatchers είναι βασικό μέρος του coroutine context και παρέχουν την ευελιξία να εκτελούμαι coroutines σε διάφορα περιβάλλοντα, ανάλογα με τις ανάγκες της εργασίας. Στην ανάπτυξη εφαρμογών Android, η επιλογή του σωστού Dispatcher είναι κρίσιμη για τη διασφάλιση της ομαλής λειτουργίας της εφαρμογής και την αποφυγή προβλημάτων απόδοσης ή αστάθειας. Η δήλωση τους γίνεται αμέσως μετά την επιλογή του Scope που θέλουμε να χρησιμοποιήσουμε έτσι ολοκληρώνεται και η επιλογή της διάρκειας ζωής του scope καθώς και το νήμα πάνω στο οποίο θα εκτελεστεί το τμήμα κώδικα που εμπεριέχεται.

Οι πιο συνηθισμένοι Dispatchers που χρησιμοποιούνται σε εφαρμογές Android είναι οι εξής:

1. **Dispatchers.Main**: Εκτελεί coroutines στο κύριο νήμα (main thread) της εφαρμογής Android. Αυτός ο Dispatcher χρησιμοποιείται για εργασίες που σχετίζονται με το UI, όπως ενημέρωση των στοιχείων του UI, διαχείριση συμβάντων χρηστών, ή κλήσεις σε λειτουργίες που πρέπει να εκτελούνται στο κύριο νήμα.
2. **Dispatchers.IO**: Σχεδιασμένος για εργασίες εισόδου/εξόδου (I/O) και ιδανικός για εργασίες I/O που απαιτούν αναμονή ή καθυστερήσεις, όπως δικτυακές κλήσεις, πρόσβαση σε βάσεις

δεδομένων, ή ανάγνωση/γραφή αρχείων. Αυτός ο Dispatcher διασφαλίζει ότι αυτές οι εργασίες εκτελούνται σε background νήματα, αποτρέποντας το μπλοκάρισμα του κύριου νήματος.

3. **Dispatchers.Default**: Χρησιμοποιείται για εργασίες υψηλής υπολογιστικής έντασης, όπως επεξεργασία δεδομένων ή υπολογισμοί. Αυτός ο Dispatcher χρησιμοποιεί μια ομάδα νημάτων που κλιμακώνεται δυναμικά ανάλογα με το φόρτο εργασίας, καθιστώντας τον ιδανικό για εργασίες που απαιτούν σημαντικούς πόρους CPU. Στην περίπτωση που δεν δηλωθεί κανένας Dispatcher κατά την χρήση οποιουδήποτε από τα παραπάνω Scopes, θα χρησιμοποιηθεί αυτός όπως ορίζει και το όνομα του.

4. **Dispatchers.Unconfined**: Εκτελεί coroutines στο ίδιο νήμα όπου ξεκινούν. Είναι ένας ειδικός τύπος Dispatcher που δεν συνιστάται για τις περισσότερες εργασίες, καθώς μπορεί να οδηγήσει σε αστάθεια ή μη αναμενόμενη συμπεριφορά.

3.3.12 Βιβλιοθήκη Lottie

Η δημιουργία του Lottie συνδέεται με την Airbnb, μια γνωστή εταιρεία τεχνολογίας που ειδικεύεται στις ενοικιάσεις κατοικιών και εμπειριών διακοπών. Η ομάδα της Airbnb αναζήτησε τρόπους για να μεταφέρει animations που είχαν δημιουργηθεί με το Adobe After Effects, ένα ισχυρό εργαλείο για τη δημιουργία κινούμενων γραφικών και εφέ, σε εφαρμογές iOS και Android. Αντί να εξάγουν τα animations σε μορφή βίντεο ή εικόνας, που είναι λιγότερο αποδοτικά σε όρους απόδοσης και μνήμης, η ομάδα εστίασε στην εξαγωγή των animations σε μορφή δεδομένων JSON.

Η Airbnb ανέπτυξε μια βιβλιοθήκη που μπορούσε να πάρει αυτά τα αρχεία JSON και να τα αναπαράγει ως κινούμενα σχέδια σε εφαρμογές. Αυτή η βιβλιοθήκη ονομάστηκε Lottie, εμπνευσμένη από τη Lotte Reiniger, μια πρωτοπόρο σκηνοθέτη κινούμενων σχεδίων που ειδικεύεται στη χρήση σιλουέτας. Η βιβλιοθήκη Lottie παρέχει έναν τρόπο να παίρνει animations από το After Effects και να τα αναπαράγει σε εφαρμογές με τρόπο αποδοτικό και ευέλικτο. Η Lottie κυκλοφόρησε ως έργο ανοικτού κώδικα, επιτρέποντας σε άλλους προγραμματιστές να χρησιμοποιήσουν και να συνεισφέρουν στη βιβλιοθήκη, ενώ παράλληλα δίνεται η δυνατότητα δημιουργίας και premium animations τα οποία προσφέρονται στους χρήστες που επιθυμούν να αγοράσουν συνδρομή.Λόγω της απλότητας και της ευελιξίας της, η Lottie κέρδισε γρήγορα δημοτικότητα και έγινε το πρότυπο για τη χρήση animations από το After Effects σε εφαρμογές iOS, Android, και ακόμα και σε εφαρμογές web μέσω της Lottie Web.

Η Lottie είναι ιδανική για την ενσωμάτωση animations σε κινητές συσκευές καθώς τα αρχεία Json αρχεία είναι μικρότερα και πιο αποδοτικά από βίντεο ή σειρές εικόνων. Ακόμα με την χρήση του composable ”LottieAnimation()” μας δίνεται η δυνατότητα να ελέγξουμε κάθε φάσμα του animation είτε αυτό αφορά το μέγεθος, την διάρκεια, το σημείο εκκίνησης και τερματισμού, την ταχύτητα κ.α.

3.3.13 Gson

Gson είναι μια βιβλιοθήκη για την ανάλυση και τη σειριοποίηση δεδομένων σε μορφή JSON (JavaScript Object Notation) στο Android. Ενώ το JSON αρχείο είναι ένα ελαφρύ, ανθρώπινα αναγνώσιμο μορφότυπο ανταλλαγής δεδομένων, και είναι δημοφιλές στο web και σε εφαρμογές λογισμικού για τη μεταφορά δεδομένων ανάμεσα σε διακομιστές και πελάτες. Η βιβλιοθήκη Gson προσφέρει μια εύχρηστη διεπαφή για την μετατροπή αντικειμένων Java σε JSON και αντίστροφα. Μπορεί να χρησιμοποιηθεί για τη σειριοποίηση (serialization) αντικειμένων σε JSON κατά την αποστολή δεδομένων σε έναν διακομιστή, καθώς και για την ανάλυση (deserialization) του JSON από αιτήσεις απόκρισης από το διακομιστή πίσω σε αντικείμενα που μπορούν να χρησιμοποιηθούν στον κώδικα της εφαρμογής. Η Gson είναι δημοφιλής λόγω της ευκολίας χρήσης και της αξιοπιστίας της στη μετατροπή μεταξύ αντικειμένων Java και JSON, καθιστώντας την καίριο εργαλείο για τη διαχείριση των αλληλεπιδράσεων με τις υπηρεσίες του διαδικτύου σε Android εφαρμογές.

Για την υλοποίηση της εφαρμογής έγινε χρήση της Gson βιβλιοθήκης προκειμένου να γίνει σειριοποίησης δεδομένων σε ένα αρχείο Json με 360.000 γραμμές όπου κάθε γραμμή αποτελούνταν από 1 ελληνική λέξη [28]. Μέσω αυτής της βιβλιοθήκης μπόρεσα να ανακτήσω όλες αυτές τις λέξεις και να τις εισάγω σε μια λίστα προκειμένου αυτή να διαχωριστεί σε περισσότερες και να δημιουργήσω επίπεδα και υποεπίπεδα (βλ. **Εικόνα 3.10** *Gson File Parsing*).

```
tbonis
override fun getDictionary(): List<DictionaryRemoteItem> {
    val jsonString = context.assets.open( fileName: "dictionary.json").bufferedReader().use { it.readText() }
    val dictionary = parseJsonToModel(jsonString)
    return dictionary
}
//Serializing and Adding Json Values to Model
TheodorosB
private fun parseJsonToModel(jsonString: String): List<DictionaryRemoteItem> {
    val gson = Gson()
    return gson.fromJson(jsonString, object : TypeToken<List<DictionaryRemoteItem>>() {}.type)
}
```

Εικόνα 3.10 *Gson File Parsing*

3.3.14 GitHub

Το Git είναι ένα διανεμημένο σύστημα ελέγχου έκδοσης (DVCS) που σχεδιάστηκε για να παρακολουθεί αλλαγές στον κώδικα και να επιτρέπει συνεργασία σε έργα λογισμικού. Δημιουργήθηκε από τον Linus Torvalds, τον δημιουργό του πυρήνα του Linux, το 2005.

Το Git αναπτύχθηκε ως απάντηση στην ανάγκη για ένα ευέλικτο και ισχυρό εργαλείο ελέγχου έκδοσης για την ανάπτυξη του Linux. Πριν το Git, το έργο Linux χρησιμοποιούσε το BitKeeper, ένα κλειστό σύστημα ελέγχου έκδοσης, για τη διαχείριση των αλλαγών στον κώδικα. Ωστόσο, το 2005, η συμφωνία χρήσης του BitKeeper έληξε, γεγονός που οδήγησε την κοινότητα του Linux στην αναζήτηση μιας εναλλακτικής λύσης.

Ο Torvalds αποφάσισε να αναπτύξει ένα νέο σύστημα ελέγχου έκδοσης που θα ήταν:

- **Διανεμημένο**: Να επιτρέπει σε κάθε προγραμματιστή να έχει ένα πλήρες αντίγραφο του αποθετηρίου και να εργάζεται τοπικά.
- **Γρήγορο και Αποδοτικό**: Να διαχειρίζεται μεγάλα έργα με πολλές αλλαγές και πολλούς συνεργάτες χωρίς καθυστερήσεις.
- **Αξιόπιστο και Ασφαλές**: Να προστατεύει από απώλεια δεδομένων και να διασφαλίζει την ακεραιότητα του κώδικα.

Το Git είναι ένα εργαλείο που επιτρέπει στους προγραμματιστές να παρακολουθούν αλλαγές στον κώδικα, να δημιουργούν αντίγραφα ασφαλείας, να ανακτούν προηγούμενες εκδόσεις, και να συνεργάζονται με άλλους προγραμματιστές. Βασίζεται στην ιδέα του "branching" και "merging", επιτρέποντας την ταυτόχρονη ανάπτυξη πολλών διαφορετικών εκδόσεων ενός έργου και την εύκολη συγχώνευση τους, ενώ χρησιμοποιούνταν κυρίως σε τοπικό περιβάλλον ή μέσω ιδιωτικών εξυπηρετητών (servers). Οι προγραμματιστές μπορούσαν να χρησιμοποιούν το Git σε τοπικούς υπολογιστές, ανταλλάσσοντας αρχεία μέσω πρωτοκόλλων όπως το SSH, το HTTP, ή το FTP, για να μοιράζονται κώδικα και να συνεργάζονται.

Με το Git, οι προγραμματιστές μπορούν να:

1. **Δημιουργούν Branches**: Να δημιουργούν διαφορετικά κλαδιά για ανάπτυξη, δοκιμές, ή νέα χαρακτηριστικά.
2. **Συγχωνεύουν (Merge) ή Εφαρμόζουν (Rebase) Κλαδιά**: Να ενώνουν κλαδιά και να διαχειρίζονται αλλαγές.
3. **Δημιουργούν Commits**: Να καταγράφουν αλλαγές με σχόλια και να δημιουργούν ιστορικό του έργου.
4. **Συνεργάζονται**: Να μοιράζονται κώδικα και να δουλεύουν σε ομάδες.

Το GitHub ιδρύθηκε το 2008 από τους Tom Preston-Werner, Chris Wanstrath, και PJ Hyett. Η κύρια ιδέα πίσω από το GitHub ήταν να δημιουργηθεί μια πλατφόρμα που θα παρέχει εύκολη πρόσβαση σε αποθετήρια Git μέσω του διαδικτύου, καθιστώντας πιο απλή τη συνεργασία σε έργα λογισμικού. Το GitHub βασίζεται στο Git ως την κύρια τεχνολογία ελέγχου έκδοσης, αλλά προσφέρει επιπλέον λειτουργίες που διευκολύνουν τη συνεργασία και τη διαχείριση έργων λογισμικού.

Με το GitHub, οι προγραμματιστές μπορούν να συνεργάζονται σε πραγματικό χρόνο, να μοιράζονται κώδικα δημόσια ή ιδιωτικά, και να εκμεταλλεύονται μια μεγάλη κοινότητα προγραμματιστών που ανταλλάσσουν γνώσεις και πόρους. Το GitHub έχει εξελιχθεί σε μια από τις μεγαλύτερες πλατφόρμες φιλοξενίας κώδικα και συνεργασίας, επηρεάζοντας τον τρόπο με τον οποίο οι προγραμματιστές εργάζονται και συνεργάζονται σε έργα λογισμικού. Προσφέρει διάφορα εργαλεία και λειτουργίες που διευκολύνουν τη συνεργασία και τη διαχείριση έργων λογισμικού:

- **Φιλοξενία Αποθετηρίων**: Παρέχει αποθετήρια Git για αποθήκευση και διαχείριση κώδικα, είτε σε δημόσια είτε σε ιδιωτικά αποθετήρια.
- **Συνεργασία και Έλεγχος Έκδοσης**: Επιτρέπει σε πολλούς προγραμματιστές να συνεργάζονται σε ένα έργο, να υποβάλλουν αλλαγές, και να διαχειρίζονται συγχωνεύσεις και αναστροφές (rebases).
- **Pull Requests και Code Reviews**: Δίνει τη δυνατότητα στους προγραμματιστές να προτείνουν αλλαγές και να ζητούν αναθεώρηση από άλλους, προάγοντας την ομαδική συνεργασία και την ποιότητα του κώδικα.
- **Issues και Project Management**: Επιτρέπει την παρακολούθηση σφαλμάτων (bugs), την ανάθεση εργασιών, και τη διαχείριση έργων μέσω εργαλείων όπως τα Issues και τα Projects.

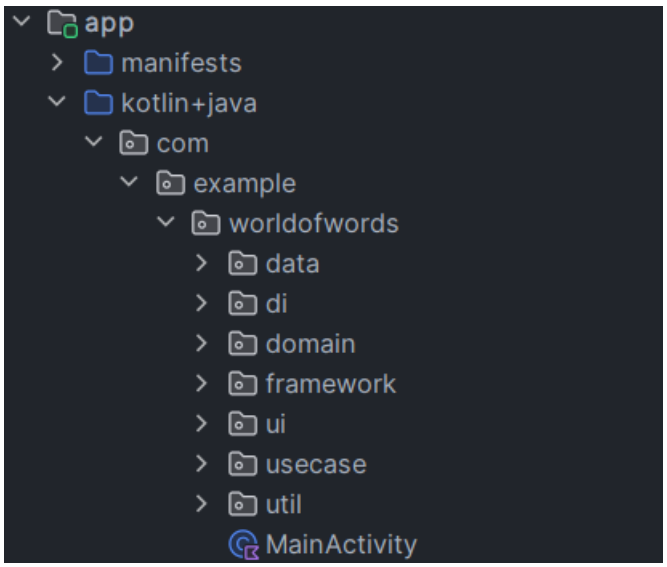
- **Διαδικτυακή Παρουσία και Κοινότητα**: Οι προγραμματιστές μπορούν να δημιουργήσουν προφίλ, να ακολουθούν άλλα αποθετήρια, και να συμμετέχουν σε μια ευρύτερη κοινότητα προγραμματιστών.

Κεφάλαιο 4ο: Υλοποίηση Εφαρμογής World Of Words

Το "World of Words" έχει σχεδιαστεί για να είναι ένα εκπαιδευτικό διαδραστικό παιχνίδι λέξεων όπου πέρα από το διασκεδαστικό κομμάτι που αφορά την εύρεση λέξεων, την αποκομιδή δώρων και την προσπέλαση επιπέδων, οι χρήστες μπορούν να εκπαιδευτούν και να διευρύνουν τις γνώσεις τους πάνω στο λεξικό. Σε σχέση με τις υπόλοιπες αντίστοιχες εφαρμογές αυτό το καθιστά το World of Words ξεχωριστό, είναι το γεγονός ότι υπάρχει αξιολόγηση του χρήστη μετά το πέρασμα κάθε επιπέδου που ανάλογα με το πόσο γρήγορα βρίσκει τις λέξεις του τρέχων γύρου καθώς και κάθε έξτρα λέξη που βρίσκει το ανταμείβει με το να προσπερνάει επίπεδα και κατά συνέπεια να λαμβάνει τα νομίσματα που θα έπαιρνε από αυτά. Ακόμα αυτό δίνει την δυνατότητα στον χρήστη να προχωρήσει πιο γρήγορα σε επίπεδα που θα είναι πιο κοντά στις ικανότητες του χωρίς όμως να χάνει τα οφέλη που θα έπαιρνε ένας χρήστης παίζοντας κανονικά. Σε αυτό το σημείο βέβαια να συμπληρώσουμε όπως αναφέρεται και στις οδηγίες εντός παιχνιδιού δεν είναι απαραίτητη προυπόθεση ένας χρήστης να παίζει με σκοπό να προσπερνάει επίπεδα, μπορεί να παίρνει τον χρόνο του να βρίσκει τις λέξεις κάθε γύρου όπως θα έπαιζε οποιοδήποτε άλλο παιχνίδι αντίστοιχης κατηγορίας. Επίσης κάθε επιπλέον λέξη που βρίσκει τον ανταμείβει με επιπλέον χρόνο κατά την αξιολόγηση ενθαρύνοντας τον χρήστη να προσπαθήσει να βρει όσες επιπλέον λέξεις μπορεί χωρίς να ανησυχεί ότι θα χάσει την δυνατότητα του να προσπεράσει επίπεδα και ανταμείβωντας τον με επιπλέον νομίσματα για κάθε επιπλέον λέξη.

4.1 Clean Architecture

Φυσικά όπως και σε κάθε εφαρμογή, είναι απαραίτητο ο προγραμματιστής ο οποίος έχει αναλάβει ένα project, να φροντίζει να είναι «καθαρό» δηλαδή να φροντίσει ότι ο κώδικας του είναι κατανοητός, επαναχρησιμοποιήσιμος, με συγκεκριμένη λειτουργία και να ακολουθεί μια δομή, μια αρχιτεκτονική που να συμβάλλει στο project να παραμένει σταθερό όσο αυτό θα αναπτύσσεται και θα αλλάζει. Ένας εύκολος τρόπος να καταννοήσουμε αυτή την δομή που έχει να κάνει με το Data Layer – Domain Layer – Ui Layer, είναι να παρατηρήσουμε τον διαχωρισμό των αρχείων μέσα στο project [1] (βλ. **Εικόνα 4.1** *Δομή Clean Architecture*) .



Εικόνα 4.1 *Δομή Clean Architecture*

Τα αρχεία στον παραπάνω πίνακα που αφορούν την αρχιτεκτονική είναι:

- Data, που αφορά το Data Layer το οποίο εμπεριέχει το Interface του DictDataSource και Datastore από το οποίο αντλούμε το ιστορικό του χρήστη, τα στοιχεία του παιχνιδιού καθώς και τις αποστολές. Ακόμα εμπεριέχονται το μοντέλο που έχω δημιουγήσει ανάλογα με το πως είναι δομημένο το Json αρχείο και έχω φτιάξει mappers που χρησιμοποιώ με σκοπό να μετατρέψω τα παραπάνω δεδομένα σε μορφή που θέλω να τα χρησιμοποιήσω, στην προκειμένη περίπτωση εισάγω όλες τις λέξεις σε μια λίστα (παρακάτω γίνεται πιο αναλυτική αναφορά ως προς την υλοποίηση τους).
- DI, που αφορά το Dependency Injection, προκειμένου να απλοποιήσει την δημιουργία εξαρτήσεων για πιο εύκολη αξιοποίηση κλάσεων από τις υπόλοιπες κλάσεις.
- Domain, στο οποίο υπάρχει τον μοντέλο Entity, το Interface της κλάσης DictRepository και ο αντίστοιχος mapper με τον οποίο διαχωρίζεται η λίστα σε μικρότερες λίστες ανάλογα με το μέγεθος τον λέξεων και τα κοινά γράμματα προκειμένου να δημιουργηθούν τα επίπεδα(levels).
- Framework, υπάρχει το implementation της κλάσης DictDataSource (DictDataSourceImpl) στο οποίο περιγράφονται οι μέθοδοι που υλοποιεί. Ακόμα σε αυτό το επίπεδο θα μπορούσαν να υπάρχουν οι σχετικές «κλήσεις» προς API.
- Ui, που φυσικά πρόκειται για το επίπεδο που αφορά οτιδήποτε έχει να κάνει με την εμφάνιση όπως Activities, Fragments, Composables καθώς και το Ui μοντέλο που σε συνδυασμό με τον UiMapper τροποποιείται έτσι ώστε να εμφανίζονται οι λέξεις σωστά στην οθόνη του χρήστη και με τα χαρακτηριστικά εκείνα προκειμένου να υλοποιείται και να διαχειρίζεται σωστά.
- UseCase, στο οποίο υπάρχει ένα UseCase για κάθε στοιχείο του χρήστη, κάθε αποστολή και κάθε στοιχείο που θέλουμε είτε να ανακτήσουμε είτε να ανανεώσουμε αλλά και για κάθε άλλη τιμή που προέρχεται από Repositories.

- ❖ Τέλος το util, αφορά λοιπά γενικά αρχεία που χρησιμοποιούνται σε πολλά διαφορετικά σημεία μέσα στα υπόλοιπα directories.

4.2 Data Layer

4.2.1 UserRepository – DataStore

Όπως προαναφέραμε το Preferences Datastore πρόκειται για μια αναβάθμιση του παλιού SharedPreferences που χρησιμοποιεί συνδυασμό Value-Pairs. Πρόκειται για μια τοπική αποθήκευση δεδομένων από τις βιβλιοθήκες του Jetpack Compose με όλα τα θετικά που προαναφέραμε παραπάνω [15][25] (βλ. **Εικόνα 4.1** *UserRepository - Datastore*).

Με την χρήση του Preferences Datastore, αποθηκεύω τα στοιχεία του χρήστη τοπικά, στοιχεία όπως :

- ❖ Username (String), με το όνομα του χρήστη το οποίο μπορεί να τροποποιήσει στην καρτέλα του
- ❖ Score (Int), στο οποίο κρατάω ένα συντελεστή για αξιολόγηση του.
- ❖ Coins (Int), τα νομίσματα που έχει συλλέξει ο χρήστης.
- ❖ Hints (Int), τον αριθμό των ενδείξεων που μπορεί να χρησιμοποιήσει για να εμφανίσει τυχαία γράμματα στο ταμπλό.
- ❖ Level (Int), που αφορά το επίπεδο στο οποίο βρίσκεται.
- ❖ SubLevel (Int), που αφορά το υποεπίπεδο στο οποίο βρίσκεται.
- ❖ TotalSubLevels (Int), αναφέρεται στα υποεπίπεδα που έχει καταφέρει ως τώρα ολοκληρώσει.
- ❖ TotalSkippedSubLevels (Int), αναφέρεται στα υποεπίπεδα που έχει καταφέρει να προσπεράσει ως τώρα με την διεκπαιρέωση καλύτερου χρόνου.
- ❖ BestTime (Long), πρόκειται για τον καλύτερο χρόνο μέσα στο οποίο ολοκλήρωσε τον γύρο.
- ❖ TotalCoins (Int), το σύνολο των νομισμάτων που έχει συλλέξει είτε που έχει συλλέξει μέσα από τους γύρους είτε με την ολοκλήρωση αποστολών.
- ❖ TotalHints (Int), το σύνολο των Ενδείξεων που έχει χρησιμοποιήσει στην προσπάθεια του να εμφανίσει τυχαία γράμματα (ακόμα και τις ενδείξεις που έχει ξοδέψει νομίσματα για να τις εμφανίσει).
- ❖ TotalExtraWords (Int), που αφορά το σύνολο των επιπλέων λέξεων που έχει βρει ως τώρα παίζοντας τα επίπεδα.

```
val userStats: Flow<UserStats> = dataStore.data
    .catch { this: FlowCollector<Preferences> exception ->
        if (exception is IOException) {
            emit(emptyPreferences())
        } else {
            throw exception
        }
    }.map { preferences ->
        val username = preferences[PreferencesKeys.USERNAME] ?: "ΝΕΟΣ ΧΡΗΣΤΗΣ"
        val points = preferences[PreferencesKeys.COINS] ?: 0
        val hints = preferences[PreferencesKeys.HINTS] ?: 0
        val level = preferences[PreferencesKeys.LEVEL] ?: 0
        val subLevel = preferences[PreferencesKeys.SUBLEVEL] ?: 0
        val score = preferences[PreferencesKeys.SCORE] ?: 0
        val totalSubLevels = preferences[PreferencesKeys.TOTALSUBLEVELS] ?: 0
        val totalPassedLevels = preferences[PreferencesKeys.TOTALSKIPPEDLEVELS] ?: 0
        val totalCoins = preferences[PreferencesKeys.TOTALCOINS] ?: 0
        val totalExtraWords = preferences[PreferencesKeys.TOTALEXTRAWORDS] ?: 0
        val bestTime = preferences[PreferencesKeys.BESTTIME] ?: 0
        val totalHints = preferences[PreferencesKeys.TOTALHINTS] ?: 0
        UserStats(
            username = username,
            hints = hints,
            coins = points,
            score = score,
            level = level,
            subLevel = subLevel,
            bestTime = bestTime,
            totalCoins = totalCoins,
            totalExtraWords = totalExtraWords,
            totalHints = totalHints,
            totalSubLevels = totalSubLevels,
            totalPassedLevels = totalPassedLevels
        ) ^map
    }
```

Εικόνα 4.2 *UserRepository - Datastore*

Στο παραπάνω σχήμα (βλ. **Εικόνα 4.2** *UserRepository - Datastore*), δημιουργείται ένα Flow που αποτελεί μια ακολουθία συνεχών ασύγχρονων τιμών. Μπορεί να θεωρηθεί σαν μια ακολουθία εκδόσεων, όπου κάθε μια από τις οποίες παρέχεται στον καταναλωτή καθώς είναι διαθέσιμη. Αυτές οι τιμές παράγονται κατά τη διάρκεια του χρόνου και μπορούν να περνούν σε χρήστες χωρίς να πρέπει να περιμένουν την ολοκλήρωση της παραγωγής τους. Με αυτό τον τρόπο υπάρχει η δυνατότητα ασύγχρονα να λαμβάνονται τα δεδομένα του χρήστη χωρίς να εμποδίζεται το κύριο νήμα της εφαρμογής και να λειτουργεί ομαλα χωρίς καθυστερήσεις. Παράλληλα έχει εισαχθεί ένα αντικείμενο UserStats στο οποίο αποθηκεύονται οι παραπάνω τιμές.

Εικόνα 4.3 *Ανανέωση Μεταβλητών Χρήστη Datastore*

Στο παραπάνω σχήμα (βλ. Εικόνα 4.3 ***Ανανέωση Μεταβλητών Χρήστη Datastore***) χρησιμοποιούνται οι μέθοδοι updateUserCoins και updateUserHints, που είναι δηλωμένες με το όρο "suspend" προκειμένου να χρησιμοποιηθούν ασύγχρονα, με τις οποίες όταν ο χρήστης ολοκληρώσει το επίπεδο αποθηκεύεται το νέο σύνολο νομισμάτων και ενδείξεων στο DataStore καθώς και ανανεώνεται το σύνολο αυτών. Με τον ίδιο τρόπο αντίστοιχα ενημερώνονται τα υπόλοιπα στοιχεία του χρήστη με τις δικές τους ιδιαιτερότητες. Ενώ με απλές μεθόδους get() λαμβάνουμε τα Flows με τις τιμές των στοιχείων του χρήστη για την απεικόνιση σε επίπεδο Ui (βλ. **Εικόνα 4.4**. *Ανάκτηση Στοιχείων Χρήστη (DataStore)*)

Εικόνα 4.4 Ανάκτηση Στοιχείων Χρήστη (DataStore)

4.2.2 QuestRepository - DataStore

Ομοίως με το UserRepository που προαναφέρθηκε παραπάνω, οι λειτουργικότητα και ο τρόπος ανανέωσης και ανάκτησης είναι παρόμοιως, το μόνο που διαφοροποιεί τα δυο Repositories αυτά, είναι η δομή του μοντέλου και οι τιμές που αποθηκεύονται σε αυτό (βλ. Εικόνα 4.5 *QuestRepository – DataStore*). Σε αυτό αποθηκεύονται τα id των πέντε αυτών αποστολών που μπορεί να έχει ο χρήστης (καθώς μπορεί να έχει μέχρι 5 αποστολές την φορά) με τις αντίστοιχες τιμές τους ώστε να αποθηκεύεται η πρόοδος του χρήστη σε κάθε αποστολή ξεχωριστά. Ακόμα αποθηκεύεται η previousDay η οποία αναφέρεται στην μέρα που ο χρήστης έλαβε τελευταία φορά νέες αποστολές, το questChanged που παίρνει τιμή true όταν ο χρήστης έχει επιλέξει να αντικαταστήσει κάποια αποστολή και τέλος το questChangedDay, που αποθηκεύεται η μέρα κατά την οποία ο χρήστης έχει αντικαταστήσει κάποια αποστολή καθώς η δυνατότητα αντικατάστασης κάποια αποστολής μπορεί να χρησιμοποιηθεί μια φορά την ημέρα.

```
val questLog: Flow<QuestLog> = dataStore.data
    .catch { this: FlowCollector<Preferences> exception ->
        if (exception is IOException) {
            emit(emptyPreferences())
        } else {
            throw exception
        }
    }.map { preferences ->
        val firstQuest = preferences[PreferencesKeys.FIRST_QUEST] ?: 0
        val firstQuestValue = preferences[PreferencesKeys.FIRST_QUEST_VALUE] ?: 0
        val secondQuest = preferences[PreferencesKeys.SECOND_QUEST] ?: 0
        val secondQuestValue = preferences[PreferencesKeys.SECOND_QUEST_VALUE] ?: 0
        val thirdQuest = preferences[PreferencesKeys.THIRD_QUEST] ?: 0
        val thirdQuestValue = preferences[PreferencesKeys.THIRD_QUEST_VALUE] ?: 0
        val fourthQuest = preferences[PreferencesKeys.FOURTH_QUEST] ?: 0
        val fourthQuestValue = preferences[PreferencesKeys.FOURTH_QUEST_VALUE] ?: 0
        val fifthQuest = preferences[PreferencesKeys.FIFTH_QUEST] ?: 0
        val fifthQuestValue = preferences[PreferencesKeys.FIFTH_QUEST_VALUE] ?: 0
        val previousDay = preferences[PreferencesKeys.PREVIOUS_DAY] ?: 0
        val isQuestChanged = preferences[PreferencesKeys.QUEST_CHANGED] ?: false
        val questChangedDay = preferences[PreferencesKeys.QUEST_CHANGED_DAY] ?: 0
        QuestLog(
            firstQuest = firstQuest,
            firstValue = firstQuestValue,
            secondQuest = secondQuest,
            secondValue = secondQuestValue,
            thirdQuest = thirdQuest,
            thirdValue = thirdQuestValue,
            fourthQuest = fourthQuest,
            fourthValue = fourthQuestValue,
            fifthQuest = fifthQuest,
            fifthValue = fifthQuestValue,
            previousDay = previousDay,
            isQuestChanged = isQuestChanged,
            questChangedDay = questChangedDay
        ) ^map
```

Εικόνα 4.5 QuestRepository - Datastore

4.2.3 Dictionary DataSource

Στη συγκεκριμένη κλάση υλοποιούνται όλες οι λειτουργίες που έχουν να κάνουν με την διαχείρηση δεδομένων και επιχειρηματικής λογικής. Η εξής κλάση αποτελείται από τις μεθόδους:

- getDictionary(), που αφορά τη σάρωση του Json αρχείου που χρησιμοποιήθηκε για την εισαγωγή των λέξεων στην εφαρμογή ενώ εισήχθησαν όλες τις λέξεις του Json αρχείου σε μια λίστα, όπως φαίνεται με την χρήση της μεθόδου parseJsonToModel() (βλ. **Εικόνα 4.6** Json File Parsing).
- getLevelImage(), που επιστρέφει ανάλογα με το επίπεδο την αντίστοιχη εικόνα που χρησιμοποιείται στο background της πίστας.
- getLevelMultiplier(), που εκτελεί την αξιολόγηση του χρήστη υπολογίζοντας ανάλογα με το επίπεδο που βρίσκεται ο χρήστης, τον χρόνο που ολοκλήρωσε το επίπεδο, τις επιπλέον λέξεις και το σύνολο των λέξεων του επιπέδου, τον συντελεστή (δηλαδή το σύνολο των υποεπίπεδων) που θα προσπεράσει. Στον πίνακα 4.7 και σχήμα 4.8, φαίνεται η αρχική βάση κατά την οποία, όταν ο χρήστης ολοκληρώσει το επίπεδο μέσα στα δευτερόλεπτα που αντιστοιχούν στην πρώτη στήλη, ανταμείβεται με επιπλέον επίπεδα που αντιστοιχούν στην δεύτερη στήλη και τα αντίστοιχα νομίσματα για κάθε επίπεδο. Για κάθε επιπλέον λέξη που βρίσκει, για κάθε επίπεδο που ανεβαίνει και για κάθε επιπλέον λέξη που υπάρχει στο γύρο (πέρα από τις 3 λέξεις του επιπέδου που είναι οι

ελάχιστες που μπορεί να έχει ένα υποεπίπεδο) ο χρόνος αυξάνεται κατά 5 δευτερόλεπτα προκειμένου να μπορεί να διεκδικήσει οποιδήποτε από τις παρακάτω ανταμειβές. Σε περίπτωση προσπεράσει την τρίτη κατηγορία που αντιστοιχούν στα 2 υποεπίπεδα, θα προσπεράσει απλά το τρέχον επίπεδο.

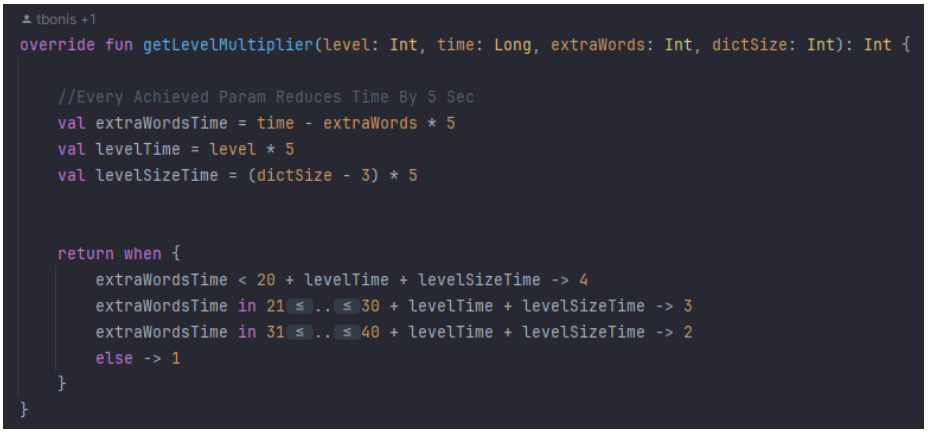
- getRandomQuest, τέλος η μέθοδος αυτή επιστρέφει μια τυχαία αποστολή πέρα από αυτές που έχει ήδη ο χρήστης.

```
tbonis +1
class DictDataSourceImpl @Inject constructor(
    @ApplicationContext private val context: Context
) : DictDataSource {


    tbonis
    override fun getDictionary(): List<DictionaryRemoteItem> {
        val jsonString = context.assets.open( fileName: "dictionary.json").bufferedReader().use { it.readText() }
        val dictionary = parseJsonToModel(jsonString)
        return dictionary
    }

    //Serializing and Adding Json Values to Model
    TheodorosB
    private fun parseJsonToModel(jsonString: String): List<DictionaryRemoteItem> {
        val gson = Gson()
        return gson.fromJson(jsonString, object : TypeToken<List<DictionaryRemoteItem>>() {}.type)
    }
```

Εικόνα 4.6 Json File Parsing To Model

Δευτερόλεπτα	Υποεπίπεδα
20	4
21-30	3
31-40	2
-	1

Πίνακας 4.7 Προσπέλαση Υποεπιπέδων



Εικόνα 4.8 Αξιολόγηση Χρήστη

Για την εισαγωγή δεδομένων-λέξεων χρησιμοποιήθηκε ένα Json αρχείο 360.000 ελληνικών λέξεων [28], το οποίο φιλταρίστηκε προκειμένου να μην εμπεριέχει λέξεις μικρότερες από 2 γράμματα και να μην περιλαμβάνονται σε αυτό ονόματα ανθρώπων, πόλεων και ποταμών. Ωστόσο εμπεριέχονται λέξεις τις αρχαίας ελληνικής, ξένες λέξεις καθώς και μέρη του λόγου όπως ουσιαστικά, επίθετα, αντωνυμίες, ρήματα, μετοχές, προθέσεις, επιρρήματα, συνδέσμους αλλά και επιφωνήματα.

4.3 Domain Layer

4.3.1 Dictionary Repository

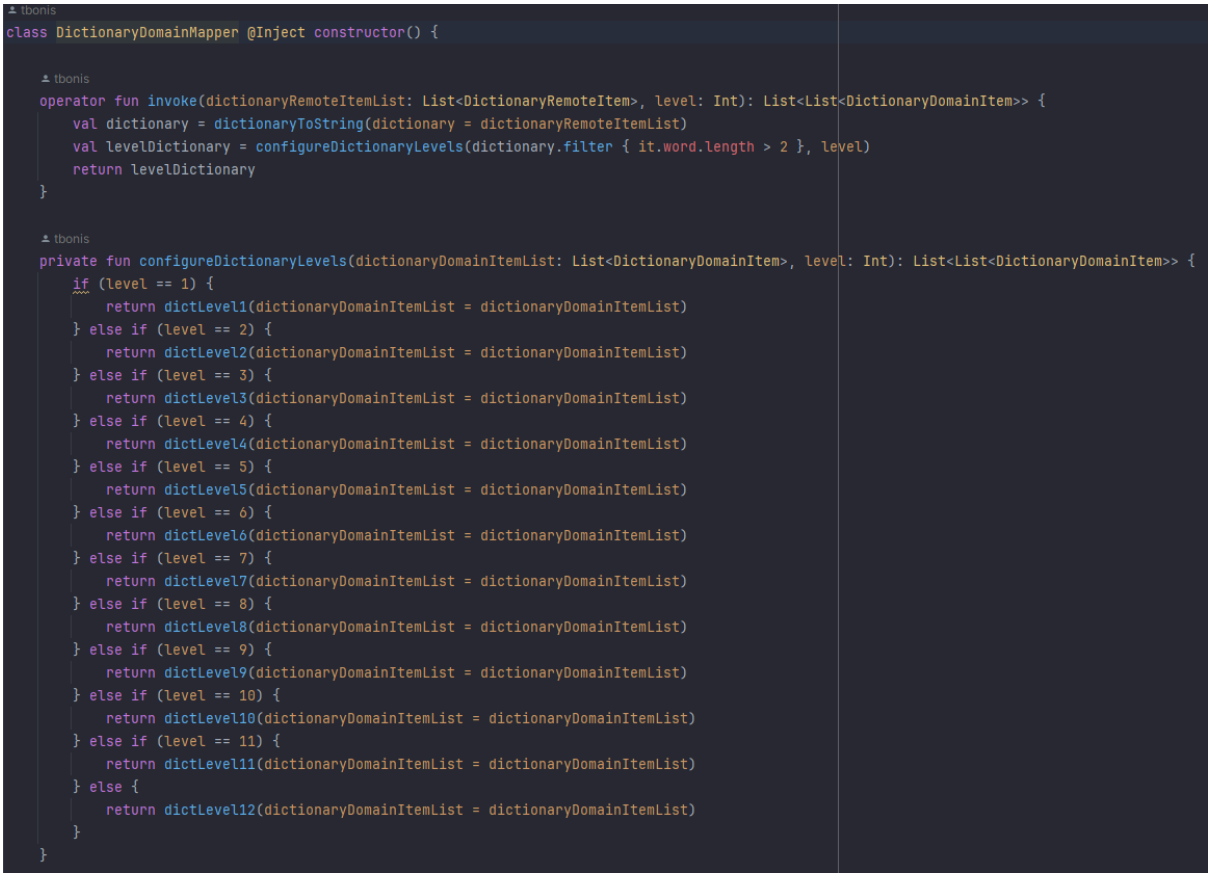
Το Dictionary Repository έχει ως σκοπό να λαμβάνει τα δεδομένα του Dictionary DataSource και να προετοιμάζει τα δεδομένα εισάγοντας την επιχειρηματική λογική ως προς την διαχείρηση των δεδομένων για το User Interface. Σε αυτή την κλάση καλούνται όλες οι μέθοδοι του DataSource με την διαφοροποίηση ότι λαμβάνεται το μοντέλο του Data Layer, τροποποιείται με την χρήση του Domain Mapper σε Entity μοντέλο και εισάγεται σε ένα «State» έτσι ώστε να διαχειρίζεται ανάλογα με το αν έρθει η λίστα γεμάτη, άδεια ή υπάρξει κάποιο πρόβλημα κατά την μετατροπή (βλ. **Εικόνα 4.9** *Dictionary Repository*).

```
class DictRepositoryImpl @Inject constructor(
    private val dictDataSource: DictDataSource,
    private val dictSubLevelBroadcast: DictSubLevelBroadcast,
    private val dictionaryResultMapper: DictionaryResultMapper,
    private val dictionaryDomainMapper: DictionaryDomainMapper
) : DictRepository {


    tbonis
    override fun getDictionary(dictionaryLevel: Int): DictionaryResult {
        val response = dictDataSource.getDictionary()
        return dictionaryResultMapper(dictionaryResponse = response, dictionaryLevel = dictionaryLevel)
    }

    tbonis
    override fun getWholeDictionary(): List<DictionaryDomainItem> {
        val response = dictDataSource.getDictionary()
        return dictionaryDomainMapper.getWholeDictionary(dictionaryRemoteItemList = response)
    }
```

Εικόνα 4.9 *Dictionary Repository*

4.3.2 Dictionary Domain Mapper

Κύριος σκοπός της κλάσης Dictionary Domain Mapper είναι να διαχωρίσει το μοντέλο του Data Layer (που πρόκειται για μια λίστα με όλες τις 360.000 λέξεις) σε υποκατηγορίες ή στην προκειμένη περίπτωση σε υποεπίπεδα. Ακόμα είναι απαραίτητο να γίνεται ο παραπάνω διαχωρισμός ώστε κάθε φορά που ο χρήστης θα προσπερνάει ένα υποεπίπεδο να μην χρειάζεται να γίνεται ξανά ολόκληρη σάρωση της λίστας καθώς αυτό θα έχει σαν αποτέλεσμα να υπάρχει μεγάλη καθυστέρηση κάθε φορά που ολοκληρώνεται ένα υποεπίπεδο πράγμα που επηρεάζει σημαντικά το User Experience. Με την κλήση του παραπάνω Mapper, παρέχουμε στην μέθοδο το τρέχον επίπεδο του χρήστη και έτσι γίνεται η σάρωση μονάχα του επιπέδου του οποίου θέλουμε να αντλήσουμε τα δεδομένα (βλ. **Εικόνα 4.10** Διαχωρισμός Επιπέδων).



```
tbonis
class DictionaryDomainMapper @Inject constructor() {

    tbonis
    operator fun invoke(dictionaryRemoteItemList: List<DictionaryRemoteItem>, level: Int): List<List<DictionaryDomainItem>> {
        val dictionary = dictionaryToString(dictionary = dictionaryRemoteItemList)
        val levelDictionary = configureDictionaryLevels(dictionary.filter { it.word.length > 2 }, level)
        return levelDictionary
    }

    tbonis
    private fun configureDictionaryLevels(dictionaryDomainItemList: List<DictionaryDomainItem>, level: Int): List<List<DictionaryDomainItem>> {
        if (level == 1) {
            return dictLevel1(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 2) {
            return dictLevel2(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 3) {
            return dictLevel3(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 4) {
            return dictLevel4(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 5) {
            return dictLevel5(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 6) {
            return dictLevel6(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 7) {
            return dictLevel7(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 8) {
            return dictLevel8(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 9) {
            return dictLevel9(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 10) {
            return dictLevel10(dictionaryDomainItemList = dictionaryDomainItemList)
        } else if (level == 11) {
            return dictLevel11(dictionaryDomainItemList = dictionaryDomainItemList)
        } else {
            return dictLevel12(dictionaryDomainItemList = dictionaryDomainItemList)
        }
    }
```

Εικόνα 4.10 *Διαχωρισμός Επιπέδων*

Δεύτερη και εξίσου πολύ σημαντική λειτουργία του Dictionary Domain Mapper είναι η εύρεση λέξεων με κοινά γράμματα, το φιλτράρισμα των λέξεων ώστε να μην εμπεριέχονται λέξεις με λιγότερα από 3 γράμματα. Επιπλέον, σε κάθε λίστα που υπάρχουν λέξεις με κοινά γράμματα σκοπός είναι να μην υπάρχουν λιγότερες από 3 λέξεις συνολικά καθώς και κατά την διάρκεια της αναβάθμισης των επιπέδων, λέξεις με το ίδιο φάρδος να μην εμφανίζονται στα ίδια επίπεδα. Ακόμη η εφαρμογή εμπεριέχει μέχρι 12 επίπεδα, στην οποία κάθε επίπεδο έχει διαφορετικές λέξεις με διαφορετικό φάρδος (βλ. **Εικόνα 4.11** Παράδειγμα Διαχωρισμού Επιπέδων {1-5}). Παραδείγματος χάριν στο Επίπεδο 1 , υπάρχουν λέξεις μονάχα με φάρδος έως 3 γράμματα, ενώ στο Επίπεδο 2 υπάρχουν λέξεις με μάκρος 3 και 4 γράμματα. Όσο αυξάνεται το επίπεδο, τόσο αυξάνεται το φάρδος των λέξεων ενώ σταδιακά μειώνεται και οι ποικιλομορφία σε αυτές.

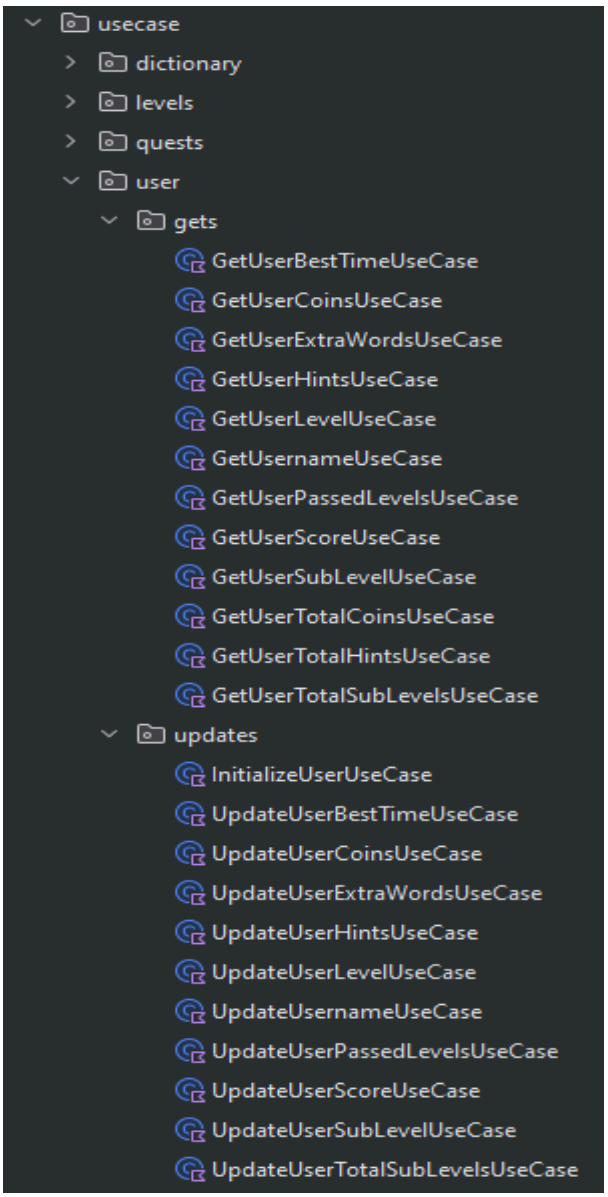
Η υλοποίηση της παραπάνω λειτουργίας γίνεται αρχικά με το φιλτράρισμα του μάκρους των λέξεων προκειμένου να λαμβάνουμε μόνο τις λέξεις ανάλογα πλήθος γραμμάτων, στην συνέχεια με την χρήση του .groupBy που αποτελεί ιδιότητα των λιστών δημιουργώ υπολίστες με κοινά γράμματα, ενώ παράλληλα παίρνω τις λίστες από τα προηγούμενα επίπεδα που μπορεί να εμπεριέχονται σε αυτό το επίπεδο. Τέλος ξαναφιλτράρω την τελική λίστα προκειμένου να αποτελείται από υπολίστες με τουλάχιστον 3 λέξεις. Αυτή η διαδικασία είναι ίδια σε κάθε μέθοδο με την διαφοροποίηση του μάκρους των λέξεων που παίρνω και την αφαίρεση των ανάλογων προηγούμενων υποεπιπέδων.

```
private fun dictLevel1(dictionaryDomainItemList: List<DictionaryDomainItem>): List<List<DictionaryDomainItem>> {
    val dictLevel1 = dictionaryDomainItemList.filter { it.word.length == 3 }
    val groupedByCommonLetters: List<List<DictionaryDomainItem>> = dictLevel1
        .groupBy { it.word.toSet() }.values.toList()
    return groupedByCommonLetters.filter { it.size >= 3 }
}

 tbonis
private fun dictLevel2(dictionaryDomainItemList: List<DictionaryDomainItem>): List<List<DictionaryDomainItem>> {
    val dictLevel2 = dictionaryDomainItemList.filter { it.word.length == 3 || it.word.length == 4 }
    val groupedByCommonLetters: List<List<DictionaryDomainItem>> = dictLevel2
        .groupBy { it.word.toSet() }.values.toList()
    val prevLevelDict = dictLevel1(dictionaryDomainItemList = dictionaryDomainItemList)
    return groupedByCommonLetters.filter { it.size >= 3 } - prevLevelDict.toSet()
}

 tbonis
private fun dictLevel3(dictionaryDomainItemList: List<DictionaryDomainItem>): List<List<DictionaryDomainItem>> {
    val dictLevel3 = dictionaryDomainItemList.filter { it.word.length == 4 }
    val groupedByCommonLetters: List<List<DictionaryDomainItem>> = dictLevel3
        .groupBy { it.word.toSet() }.values.toList()
    val prevLevelDict = dictLevel2(dictionaryDomainItemList = dictionaryDomainItemList)
    return groupedByCommonLetters.filter { it.size >= 3 } - prevLevelDict.toSet()
}

 tbonis
private fun dictLevel4(dictionaryDomainItemList: List<DictionaryDomainItem>): List<List<DictionaryDomainItem>> {
    val dictLevel4 = dictionaryDomainItemList.filter { it.word.length in 3 ≤ .. ≤ 5 }
    val groupedByCommonLetters: List<List<DictionaryDomainItem>> = dictLevel4
        .groupBy { it.word.toSet() }.values.toSet().toList()
    val prevLevelDict = dictLevel2(dictionaryDomainItemList = dictionaryDomainItemList)
    return groupedByCommonLetters.filter { it.size >= 3 } - prevLevelDict.toSet()
}

 tbonis
private fun dictLevel5(dictionaryDomainItemList: List<DictionaryDomainItem>): List<List<DictionaryDomainItem>> {
    val dictLevel5 = dictionaryDomainItemList.filter { it.word.length == 4 || it.word.length == 5 }
    val groupedByCommonLetters: List<List<DictionaryDomainItem>> = dictLevel5
        .groupBy { it.word.toSet() }.values.toSet().toList()
    val prevLevelDict = dictLevel3(dictionaryDomainItemList = dictionaryDomainItemList)
    return groupedByCommonLetters.filter { it.size >= 3  } - prevLevelDict.toSet()
}
```

Εικόνα 4.11 *Παράδειγμα Διαχωρισμού Επιπέδων {1-5}*

4.3.3 Use Cases

Μετέπειτα από το Domain Layer, υπάρχουν τα UseCases τα οποία έχουν ως κύριο σκοπό να πραγματοποιούν μονάχα μια συγκεκριμένη λειτουργία ενώ διατυπώνονται με τρόπο που να γίνεται σαφή η λειτουργία τους. Τα UseCases χρησιμοποιούνται ως μεσολαβητές προκειμένου να επικοινωνήσει το ViewModel με το Domain Layer και κάθε UseCase αντιπροσωπεύει κάθε μία από τις παραπάνω μεθόδους που έχει υλοποιηθεί στο Dictionary Repository. Για παράδειγμα στην περίπτωση ανάκτησης των στοιχείων του χρήστη καθώς και ανανέωσης τους υπάρχει ένα ξεχωριστό UseCase προκειμένου να αποφευχθεί η σάρωση όλων των στοιχείων του για την ανανέωση μονάχα ενός πεδίου. Αντίστοιχα έχουν δημιουγηθεί ξεχωριστά UseCases για την ανάκτηση στοιχείων σε ότι αφορά τις αποστολές, τα επίπεδα και το ίδιο το Λεξικό (βλ. **Εικόνα 4.12** *UseCases*).



Εικόνα 4.12 *Use Cases*

4.4 UI Layer

Στο υποκεφάλαιο αυτό όπως έχει προαναφερθεί, υπάρχουν μοντέλα και κλάσεις που σχετίζονται αποκλειστικά με την απεικόνιση των δεδομένων και την τροποποίηση του τρόπου εμφάνισης του περιεχομένου. Σε αυτό το επίπεδο το ViewModel είναι ο κυρίαρχος παράγοντας που καλεί κάθε UseCase που χρειάζεται προκειμένου να το φιλτράρει και να το τροποποιήσει σε μοντέλα προκειμένου να παρουσιάσει το περιεχόμενο του όπως πρέπει να εμφανιστεί. Με αυτό τον τρόπο διαχείρισης αξιοποιείται το πρότυπο MVVM(Model-View-ViewModel) ενώ παράλληλα διατηρείται η κατάσταση του αντικειμένου κατά μέσω διαμορφώσεων.

4.4.1 Οθόνες Αρχικού Menu

Menu Fragment

Αρχικά δημιουργήθηκε το Menu, το οποίο αφορά την αρχική οθόνη που εμφανίζεται όταν ανοίξει η εφαρμογή. Στο αρχικό Activity της εφαρμογής φιλοξενήθηκε ένα Fragment το οποίο έχει το δικό του κύκλο ζωής και θα αποτελεί ολόκληρη την οθόνη και όχι μέρος του καθώς επίσης θα περιέχει το βασικό MenuComposable που απεικονίζει/σχεδιάζει τον τρόπο εμφάνισης της αρχικής οθόνης. Με την χρήση της μεθόδου onViewCreated(), που καλείται όταν η δομή της διεπαφής χρήστη του Fragment έχει δημιουργηθεί και είναι έτοιμη να εμφανιστεί στην οθόνη. Σε αυτό το στάδιο του κύκλου ζωής, τέθηκε ένας observer για την πλοήγηση μεταξύ Fragments ενώ καλούνται οι απαραίτητες μέθοδοι για την ανάκτηση των στοιχείων χρήστη και των αποστολών του. Τέλος έγινε η εισαγωγή του ComposeView στο xml αρχείο αυτού του Fragment προκειμένου να εισαχθεί το Composable σε αυτό ενώ του παραχωρήθηκε το MenuViewModel προκειμένου να λαμβάνει εσωτερικά τα δεδομένα προς εμφάνιση(βλ. **Εικόνα 4.13** *MenuFragment*).

```
@AndroidEntryPoint
class MenuFragment : Fragment(R.layout.fragment_menu) {

    private val viewModel: MenuViewModel by viewModels()

    private var _binding: FragmentMenuBinding? = null

    protected val binding get() = _binding!!

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {
        val baseInflater = LayoutInflater.from(requireActivity())
        _binding = FragmentMenuBinding.inflate(baseInflater)
        return _binding?.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setupObservers()
        getUserProfile()
        getQuests()
        binding.menuComposeView.setContent {
            WorldOfWordsTheme {
                Surface(
                    modifier = Modifier
                        .fillMaxSize()
                ) {
                    MenuScreen(
                        viewModel = viewModel,
                        onStartGameClick = viewModel::startGame,
                        onUsernameEdit = viewModel::updateUsername,
                        onCompleteClick = viewModel::onCompleteQuest,
                        onChangeQuestClick = viewModel::onChangeQuest
                    )
                }
            }
        }
    }
```

Εικόνα 4.13 *MenuFragment*

MenuViewModel

Σε αυτό το σημείο είναι απαραίτητο να εξετάσουμε τα δεδομένα καθώς και την διαχείριση τους πριν την εμφάνιση της αρχικής οθόνης. Όπως παρατηρείται στην Εικόνα 4.13 Ανάκτηση Στοιχείων Χρήστη, οι πρώτες βασικές μέθοδοι είναι οι εξής:

- getUserProfile(), η οποία εκτελείται σε viewModelScope προκειμένου να είναι ενεργή για όσο το MenuViewModel είναι ζωντανό και χρησιμοποιεί Dispatchers προκειμένου να θέσει την κορουτίνα στο main Thread. Η χρήση του είναι απαραίτητη προκειμένου να ανακτηθούν τα στοιχεία του χρήστη ασύγχρονα χωρίς να εμποδιστεί το ίδιο το main Thread εφόσον φυσικά δεν εκτελούνται βαριές ή μεγάλες επιχειρήσεις (βλ. Σχ. 4.14 Ανάκτηση Στοιχείων Χρήστη)

```
fun getUserProfile() {
    viewModelScope.launch(Dispatchers.Main) { this: CoroutineScope
        _userCoinsUi.value = getUserCoinsUseCase().firstOrNull()
        _userHintsUi.value = getUserHintsUseCase().firstOrNull()
        _userNameUi.value = getUsernameUseCase().firstOrNull()
        _userScoreUi.value = getUserScoreUseCase().firstOrNull()
        _userLevelUi.value = getUserLevelUseCase().firstOrNull()
        _userSubLevelUi.value = getUserSubLevelUseCase().firstOrNull()
        _userBestTimeUi.value = getUserBestTimeUseCase().firstOrNull()
        _userExtraWordsUi.value = getUserExtraWordsUseCase().firstOrNull()
        _userPassedSubLevelsUi.value = getUserPassedLevelsUseCase().firstOrNull()
        _userTotalCoinsUi.value = getUserTotalCoinsUseCase().firstOrNull()
        _userTotalHintsUi.value = getUserTotalHintsUseCase().firstOrNull()
        _userTotalSubLevelsUi.value = getUserTotalSubLevelsUseCase().firstOrNull()
    }
}
```

Εικόνα 4.14 *Ανάκτηση Στοιχείων Χρήστη*

- getQuests(), αντίστοιχα με την προηγούμενη μέθοδο εκτελείται και αυτή στο main Thread (βλ. **Εικόνα 4.15** Ανάκτηση Αποστολών Χρήστη). Ωστόσο εμπεριέχει επιπλέον μεθόδους όπως τον QuestLogUiMapper() στο οποίο διασταυρώνει τις υπάρχουσες αποστολές και τις εισάγει σε UI Μοντέλο (βλ. **Εικόνα 4.16** *Υλοποίηση Mapper Μοντέλου Αποστολής*). Επιπλέον η checkQuestChanged() ελέγχει αν ο χρήστης έχει αντικαταστήσει την αποστολή και τέλος η checkForNewQuests() ελέγχει αν υπάρχει χώρος για καινούργιες αποστολές . Σε περίπτωση που ο χρήστης έχει λιγότερες από 5 αποστολές καλεί τη getRandomQuestUseCase() προκειμένου να προσθέσει από 1 έως 2 διαφορετικές αποστολές από τις υπάρχουσες, ενώ παράλληλα ανανεώνει την ημέρα που προστέθηκαν οι νέες αποστολές ώστε να μην μπορεί να πάρει παραπάνω από 2 αποστολές την ημέρα. Τέλος ανανεώνει το QuestDataStore με τις νέες αποστολές που μόλις του δόθηκαν.
- Πέρα από τις βασικές αυτές μεθόδους εμπεριέχει και τις μεθόδους addOneQuest(), addTwoQuests(), updateQuestPreferences(), onChangeQuest(), onCompleteQuest() κ.α. τα όποία εκτελούν λειτουργίες σχετικές με την ονομασία τους μετά από αλληλεπίδραση του χρήστη με τα αντίστοιχα elements στην οθόνη.

```
fun getQuests(){
    viewModelScope.launch(Dispatchers.Main) { this: CoroutineScope
        val firstQuestId = getFirstQuestUseCase().firstOrNull()
        val secondQuestId = getSecondQuestUseCase().firstOrNull()
        val thirdQuestId = getThirdQuestUseCase().firstOrNull()
        val fourthQuestId = getFourthQuestUseCase().firstOrNull()
        val fifthQuestId = getFifthQuestUseCase().firstOrNull()
        val wholeQuestLog = getAllQuestsUseCase()
        val remoteQuestLog = listOf(firstQuestId,secondQuestId,thirdQuestId,fourthQuestId,fifthQuestId)

        //Mapping RemoteQuests to Local
        val curQuestLog = questLogUiMapper(remoteQuestLog = remoteQuestLog, wholeQuestLog = wholeQuestLog)

        //Check If Quest is Changed
        val calendar = Calendar.getInstance()
        val currentDay = calendar.get(Calendar.DAY_OF_MONTH)
        checkQuestChanged(currentDay = currentDay)

        //Update Quests
        _questLogUi.value = checkForNewQuests(curQuestLog = curQuestLog, currentDay = currentDay)

    }
}
```

Εικόνα 4.15 *Ανάκτηση Αποστολών Χρήστη*

```
class QuestLogUiMapper @Inject constructor() {

    operator fun invoke(remoteQuestLog: List<Pair<Int,Int>?>, wholeQuestLog: List<QuestUiItem>) : List<QuestUiItem> {

        val questLogUi = listOf(QuestUiItem(), QuestUiItem(), QuestUiItem(), QuestUiItem(), QuestUiItem())
        val curQuestLog = questLogUi.mapIndexed { index, questUiItem ->
             val remoteItem = remoteQuestLog[index]
             if (remoteItem != null && remoteItem.first != 0) {
                 QuestUiItem(
                     id = remoteItem.first,
                     initValue = remoteItem.second,
                     targetValue = questUiItem.targetValue,
                     descriptionRes = questUiItem.descriptionRes,
                     iconRes = questUiItem.iconRes,
                     rarity = questUiItem.rarity,
                     reward = questUiItem.reward,
                     isNew = questUiItem.isNew,
                     titleRes = questUiItem.titleRes,
                     questType = questUiItem.questType
                 ) ^mapIndexed
             } else {
                 QuestUiItem(
                     id = questUiItem.id,
                     initValue = questUiItem.initValue,
                     targetValue = questUiItem.targetValue,
                     descriptionRes = questUiItem.descriptionRes,
                     iconRes = questUiItem.iconRes,
                     rarity = questUiItem.rarity,
                     isNew = questUiItem.isNew,
                     reward = questUiItem.reward,
                     titleRes = questUiItem.titleRes,
                     questType = questUiItem.questType
                 ) ^mapIndexed
             }
         }

        val updatedQuestLog = matchQuestsId(curQuestLog = curQuestLog, wholeQuestLog = wholeQuestLog)
        return updatedQuestLog.filter { it.id != 0 }
    }
```

Εικόνα 4.16 *Υλοποίηση Mapper Μοντέλου Αποστολής*

MenuScreen

Το MenuScreen πρόκειται για το βασικό αρχικό Composable που εμφανίζεται και διατυπώνεται στο MenuFragment (βλ. **Εικόνα 4.13** *MenuFragment*). Αρχικά το πρώτο πράγμα που παρατηρούμε με το που ανοίγει η εφαρμογή είναι το background το οποίο έχει υλοποιηθεί με την χρήση της βιβλιοθήκης Lottie. Μέσω αυτής της open-source βιβλιοθήκης, έχει χρησιμοποιηθεί ένα animation που είναι σε μορφή Json αρχείου και έχει τροποποιηθεί ο τρόπος εμφάνισης του (βλ **Εικόνα 4.18** *Υλοποίηση Lottie*).

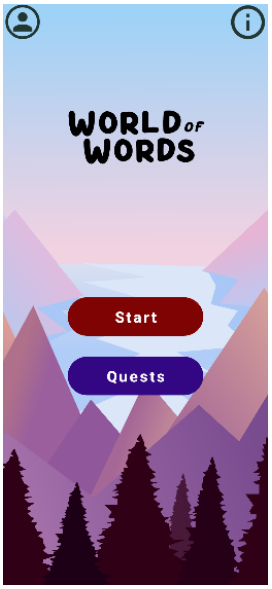
```
val isPlaying by remember {
    mutableStateOf( value: true)
}

val speed by remember {
    mutableStateOf( value: 1.5f)
}

val composition by rememberLottieComposition(
    spec = LottieCompositionSpec.RawRes(R.raw.world_background)
)

LottieAnimation(
    composition = composition,
    iterations = LottieConstants.IterateForever,
    isPlaying = isPlaying,
    speed = speed,
    restartOnPlay = false,
    modifier = Modifier.fillMaxSize(),
    contentScale = ContentScale.FillHeight
)
```

Εικόνα 4.18 *Υλοποίηση Lottie*

Τα Elements τα οποία απαρτίζουν αυτήν την οθόνη είναι τα εξής (βλ. **Εικόνα 4.19** *Αρχική Οθόνη Menu)*:

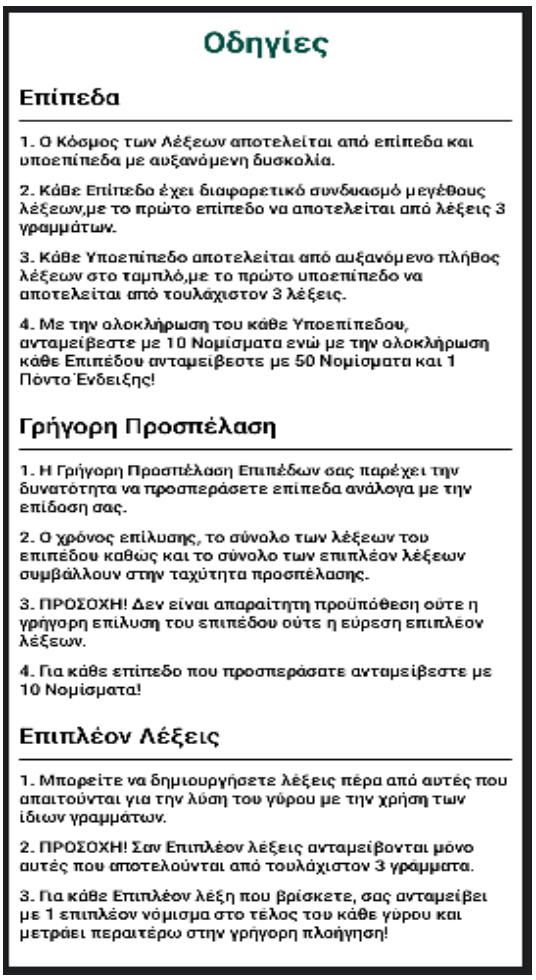
- Το κουμπί «Start», που προκαλεί το fragment navigation προς το GameFragment στο οποίο απεικονίζεται η τρέχουσα πίστα του χρήστη.
- Το κουμπί «Quests», που εμφανίζει το QuestDialog με το ανάλογο animation που εμπεριέχει τις τρέχουσες αποστολές του χρήστη, όπου όταν υπάρχουν καινούργιες αποστολές εμπεριέχει χρωματιστό animation όπως και οι αντίστοιχες αποστολές εσωτερικά.
- Το εικονίδιο του UserProfile, το οποίο εμφανίζει ένα UserDialog με τα βασικά στοιχεία του χρήστη στο επάνω μέρος, τα οποία δείχνουν το όνομα, το επίπεδο, το υποεπίπεδο, τα νομίσματα καθώς και τις ενδείξεις. Αμέσως κάτω από αυτό εμφανίζονται όλες οι πληροφορίες του ιστορικού του χρήστη, όπως το σύνολο περασμένων επιπέδων, των περασμένων υποεπιπέδων, των νομισμάτων, των προσπερασμένων υποεπιπέδων, των χρησιμοποιημένων ενδείξεων, των επιπλέων λέξεων και του καλύτερου χρόνου (βλ. **Εικόνα 4.20** *UserDialog*).
- Το εικονίδιο του Info, το οποίο μας εμφανίζει ένα InfoDialog που περιέχει όλες τις οδηγίες ενημερώνοντας τον χρήστη σχετικά με τον τρόπο διεξαγωγής του παιχνίδιού. Οι οδηγίες που

περιγράφονται, αναφέρονται στον τρόπο με τον οποίο είναι διαμορφωμένα τα επίπεδα, πως λειτουργεί η γρήγορη προσπέλαση, την δυνατότητα να βρίσκει επιπλέον κρυφές λέξεις και τέλος κάθε πότε εμφανίζονται νέες αποστολές και τις κατηγορίες τους (βλ. **Εικόνα 4.20** *InfoDialog)*.



Εικόνα 4.19 *Αρχική Οθόνη Menu*

Εικόνα 4.20 *UserDialog*



Εικόνα 4.21 *InfoDialog*

Στο QuestDialog Composable, όπως προαναφέρθηκε δίνει την δυνατότητα στον χρήστη να έχει έως 5 αποστολές την φορά και κάθε μέρα του δίνονται έως και 2 τυχαίες διαφορετικές αποστολές από αυτές που έχει ήδη. Το μοντέλο της αποστολής έχει τα παρακάτω πεδία:

- i**d(Int)**, που είναι το primary key που το διαχωρίζει από τα υπόλοιπα και χρησιμοποιείται για την αντιστοίχιση με την λίστα όλων των αποστολών προκειμένου να ανακτήσουμε τις υπόλοιπες πληροφορίες για κάθε αποστολή που διαθέτουμε.
- **initValue(Int)**, που είναι η αρχική του τιμή. Όταν ο χρήστης καταφέρνει να επιτύχει μέρος της αποστολής η τιμή της initValue ανανεώνεται και αποθηκεύεται με την χρήση useCase στο Quest Repository του Datastore.
- **targetValue(Int)**, το οποίο αφορά τον στόχο προκειμένου να ολοκληρωθεί η αποστολή.
- **reward(Int)**, που είναι η ανταμοιβή που θα λάβει ο χρήστης με την ολοκλήρωση της.
- **titleRes(Int)**, το id του κειμένου στο string.xml αρχείο που αφορά τον τίτλος της αποστολής.

- **descriptionRes(Int)**, ομοίως με το titleRes με την διαφορά ότι αφορά την περιγραφή της αποστολής.
- **iconRes(Int)**, αφορά την εικόνα της αποστολής.
- **questType(QuestType)**, που διαχωρίζει την κάθε αποστολή με βάση το είδος της (Coin,Hint,Time,Level,ExtraWords,ExtraLevels). Στο σύνολο αυτή τη στιγμή υπάρχουν 40 αποστολές με στόχους όπως συλλογή νομισμάτων, χρησιμοποίηση ενδείξεων, πέρασμα επιπέδων, προσπέρασμα επιπέδων με την επίτευξη καλύτερου χρόνου και την εύρεση επιπλέων λέξεων πέρα από τις υπάρχουσες του επιπέδου. Κάθε μία από τις παραπάνω διακρίνεται σε 4 επιμέρους κατηγορίες με μεγαλύτερους στόχους και μεγαλύτερες ανταμοιβές όπως αναλύονται στο πεδίο rarity.
- **rarity (String)**, απο αφορά την δυσκολία της αποστολής. Κάθε αποστολή έχει ένα βαθμό δυσκολίας, όσο πιο δύσκολη η αποστολή τόσο μεγαλύτερη η ανταμοιβή ωστόσο οι ανταμοιβές διαφέρουν όχι μόνο από την δυσκολία αλλά και τον στόχο της. Οι βαθμίδες δυσκολίας είναι κατά αύξουσα σειρά {common, rare, epic και legendary} και αντίστοιχα τα χρώματα τους για τον πιο εύκολο διαχωρισμό τους {πράσινο, μπλε, μωβ και πορτοκαλί} (βλ. **Εικόνα 4.22** Quest Dialog)

Εικόνα 4.22 *Quest Dialog*

4.4.2 Οθόνες Παιχνιδιού

GameViewModel

Η μέθοδος init() του GameViewModel είναι η πιο βασική σε σχέση με τις υπόλοιπες μεθόδους του καθώς δημιουργεί ολόκληρο το επίπεδο του χρήστη (βλ **Εικόνα 4.23** *Μέθοδος init() του GameViewModel*). Η init() εκτελείται κάθε φορά που είτε ο χρήστης ολοκληρώνει ένα επίπεδο δηλαδή καταφέρνει να λύσει όλα τα υποεπίπεδα της κατηγορίας είτε έχει μόλις ανοίξει την εφαρμογή. Με το κάλεσμα αυτή της μεθόδου ανακτούνται τα στοιχεία του χρήστη όπως για παράδειγμα τα νομίσματα, τις ενδείξεις που έχει, σε ποιο επίπεδο και υποεπίπεδο βρίσκεται. Έπειτα με την χρήση της μεθόδου getDictionaryLevelUseCase() παρέχεται στο Data Layer το επίπεδο του χρήστη προκειμένου να επιστραφεί το State της λίστας με τις υπολίστες λέξεων που απαρτίζουν ουσιαστικά το τρέχον επίπεδο. Εφόσον επιστραφεί το State της λίστας από το Domain την αντιστοιχούμε σε UiState που αποτελείται από :

1. ErrorLevelUiState, που επιστρέφει στην περίπτωση που δεν έρθει λίστα από το Domain Layer.
2. LoadingLevelUiState, που ορίζεται όταν υπάρχει αναμονή κατά την ανάκτηση της λίστας του επιπέδου.
3. SuccessLevelUiState, που επιστρέφει όταν ο χρήστης έχει ολοκληρώσει το επίπεδο προκειμενου να εμφανιστεί η οθόνη επιτυχίας.
4. DefaultLevelUiState, που επιστρέφει όταν έχει ανακτηθεί η λίστα από το Domain Layer. Με την χρήση του DictionaryLevelUiMapper μετατρέπεται κάθε λέξη του Domain Μοντέλου σε Ui Μοντέλο το οποίο και επιστρέφεται και αποθηκεύεται μέσα στο DefaultUiState.

```
fun init() {
    stopTimer()
    _extraWordsSetUi.value = emptySet()

    //Reading User Stats
    getUserBasicStats()

    //Creating values to avoid Smart Casting
    val userLevel = _userLevelUi.value
    val userSubLevel = _userSubLevelUi.value

    //Checking Nullability
    if (userLevel != null && userSubLevel != null) {

        //Receiving the Remote State
        val domainResult = getDictionaryLevelUseCase(dictionaryLevel = userLevel)

        //Handling Background Images
        _imageUi.value = getLevelImageUseCase(level = userLevel)

        //Handling the Dictionary Level State
        _dictionaryLevelUiState.value = handleLevelDictionaryState(dictionaryResultState = domainResult)
        val dictionaryLevelUiState = _dictionaryLevelUiState.value

        //Handling the Dictionary Sub Level State
        _dictionarySubLevelUiState.value =
            handleSubLevelDictionaryState(dictionaryLevelUiState = dictionaryLevelUiState, userSubLevel = userSubLevel)

        _dictionaryLevelSize.value = retrieveLevelDictionaryUi(dictionaryLevelUiState).size

        updateSubLevelDictUseCase(
            subLevelDictItem = retrieveSubLevelDictionaryUi(dictionarySubLevelUiState = _dictionarySubLevelUiState.value)
        )
        updateProgressBar()
    }
}
```

Εικόνα 4.23 *Μέθοδος init() του GameViewModel*

Αντίστοιχα με τον ίδιο τρόπο αντιστοιχείται το DictionaryLevelUiState με το DictionarySubLevelUiState και γίνεται χρήση του DictionarySubLevelUiMapper για να διαχωριστεί το υποεπίπεδο (ανάλογα με το τρέχον υποεπίπεδο του χρήστη) από τα υπόλοιπα επίπεδα προκειμένου να εισαχθεί στο DefaultSubLevelUiState (βλ **Εικόνα 4.24** *Διαχείριση Domain και Ui State*). Όλη η παραπάνω διαδικασία γίνεται προκειμένου να υπάρχει έλεγχος ως προς την κατάσταση του αντικειμένου ενώ κάθε Ui Mapper (Level και SubLevel) να τροποποιεί το μοντέλο ανάλογα με τις ανάγκες της εκάστοτε κατάστασης.

```
private fun handleLevelDictionaryState(dictionaryResultState: DictionaryResult): DictionaryLevelUiState {
    val dictionaryLevelState = when (dictionaryResultState) {
        is DictionaryResult.DefaultResult -> DictionaryLevelUiState.DefaultLevelUiState(
            dictionaryLevelUiItems = dictionaryLevelUiMapper(
                dictionary = dictionaryResultState.dictionaryDomainItems
            ).sortedByDescending { it.size }.reversed()
        )

        DictionaryResult.ErrorResult -> DictionaryLevelUiState.ErrorLevelUiState
        DictionaryResult.LoadingResult -> DictionaryLevelUiState.LoadingLevelUiState
    }
    return dictionaryLevelState
}

private fun handleSubLevelDictionaryState(dictionaryLevelUiState: DictionaryLevelUiState, userSubLevel: Int): DictionarySubLevelUiState {
    val recentSubLevelItem = getSubLevelBroadcastUseCase()
    val dictionaryLevelState = when (dictionaryLevelUiState) {
        is DictionaryLevelUiState.DefaultLevelUiState -> {
            if (recentSubLevelItem.isNullOrEmpty()){
                DictionarySubLevelUiState.DefaultSubLevelUiState(
                    dictionarySubLevelUiItems = dictionarySubLevelUiMapper(
                        dictionary = dictionaryLevelUiState.dictionaryLevelUiItems,
                        userSubLevel = userSubLevel
                    )
                )
            }else{
                DictionarySubLevelUiState.DefaultSubLevelUiState(
                    dictionarySubLevelUiItems = recentSubLevelItem
                )
            }
        }
        DictionaryLevelUiState.ErrorLevelUiState -> DictionarySubLevelUiState.ErrorSubLevelUiState
        DictionaryLevelUiState.LoadingLevelUiState -> DictionarySubLevelUiState.LoadingSubLevelUiState
        DictionaryLevelUiState.SuccessLevelUiState -> DictionarySubLevelUiState.SuccessSubLevelUiState
    }
    startTimer()
    return dictionaryLevelState
}
```

Εικόνα 4.24 *Διαχειριση Domain και Ui State*

Ενώ οι λίστες στο Domain Layer ως τώρα αποτελούνταν απλά από λέξεις τώρα εισάγεται η κάθε μία λέξη στο νέο μοντέλο DictionaryUiItem με τα παρακάτω χαρακτηριστικά (βλ. **Εικόνα 4.25** *DictionaryUiItem, LevelUiState, SubLevelUiState*)

1. word (String), όπου η λέξη που προέρχεται από το Domain Layer εισάγεται σε αυτό το πεδίο.
2. wordIsVisible (Boolean), όπου έχει αρχική τιμή false καθώς η λέξη δεν είναι αρχικά εμφανής στο ταμπλό.
3. charsVisibility (List<CharsVisibility>), που πρόκειται για μια λίστα που περιέχει κάθε ένα χαρακτήρα της λέξης word ξεχωριστά με το δικό του πεδίο charIsVisible. Με αυτόν τον τρόπο είναι δυνατή η διαχείριση καθενός χαρακτήρα της λέξης ξεχωριστά σε περίπτωση που επιθυμούμε να εμφανίσουμε ένα τυχαίο χαρακτήρα.

```
sealed class DictionaryLevelUiState{
    object LoadingLevelUiState : DictionaryLevelUiState()
    data class DefaultLevelUiState(val dictionaryLevelUiItems : List<List<DictionaryUiItem>>) : DictionaryLevelUiState()
    object SuccessLevelUiState : DictionaryLevelUiState()
    object ErrorLevelUiState : DictionaryLevelUiState()
}

sealed class DictionarySubLevelUiState{
    object LoadingSubLevelUiState : DictionarySubLevelUiState()
    data class DefaultSubLevelUiState(val dictionarySubLevelUiItems : List<DictionaryUiItem>) : DictionarySubLevelUiState()
    object SuccessSubLevelUiState : DictionarySubLevelUiState()
    object ErrorSubLevelUiState : DictionarySubLevelUiState()
}

data class DictionaryUiItem(
    val word: String = "",
    val wordIsVisible: Boolean = false,
    val charsVisibility: List<CharsVisibility> = listOf()
)

data class CharsVisibility(
    val char: Char,
    val charIsVisible: Boolean
)
```

Εικόνα 4.25 *DictionaryUiItem, LevelUiState, SubLevelUiState*

Καθώς τα επίπεδα είναι μεγάλα και προκειμένου να αποφευχθεί η ανάκτηση ολόκληρου του επιπέδου σε περίπτωση που ο χρήστης έχει ολοκληρώσει κάποια υποεπίπεδα, κρίθηκε απαραίτητη προυπόθεση η δημιουργία μιας μεθόδου updateSubLevel() η οποία έχει ως σκοπό να ανανεώνει μονάχα το SubLevelUiState προκειμένου να προχωράει στο επόμενο υποεπίπεδο χωρίς καθυστέρηση (βλ. **Εικόνα 4.26** *Ανανέωση Υποεπιπέδου - updateSubLevel()*).

```
fun updateSubLevel() {

    stopTimer()
    _extraWordsSetUi.value = emptySet()

    //Reading User Stats
    getUserBasicStats()

    //Creating values to avoid Smart Casting
    val userSubLevel = _userSubLevelUi.value

    //Checking Nullability and Handling SubLevelUiState
    if (userSubLevel != null) {
        val dictionaryLevelUiState = _dictionaryLevelUiState.value
        _dictionarySubLevelUiState.value =
            handleSubLevelDictionaryState(dictionaryLevelUiState = dictionaryLevelUiState, userSubLevel = userSubLevel)

        updateProgressBar()
    }
    updateSubLevelDictUseCase(subLevelDictItem = retrieveSubLevelDictionaryUi(
        dictionarySubLevelUiState = _dictionarySubLevelUiState.value)
    )
}
```

Εικόνα 4.26 *Ανανέωση Υποεπιπέδου - updateSubLevel()*

Κάθε φορά που ο χρήστης προσπαθεί να βρει μια λέξη είτε αυτή είναι στο ταμπλό είτε προσπαθεί να βρει μια κρυφή επιπλέον λέξη, εκτελείται η μέθοδος updateWordsFound(), η οποία αρχικά ελέγχει αν η λέξη που σχημάτισε ο χρήστης είναι μέσα στο τρέχον επίπεδο,

στην περίπτωση που είναι γίνεται χρήση της μεθόδου mapFoundWords() του DictionarySubLevelUiMapper προκειμένου να ανανεωθεί η τιμή wordIsVisible του μοντέλου DictionaryUiItem που εμπεριέχει αυτή την λέξη. Διαφορετικά γίνεται έλεγχος σε περίπτωση που ανήκει μέσα στο λεξικό και σε αυτή την περίπτωση εισάγεται σε μια λίστα με τις επιπλέον λέξεις αυτού του γύρου (βλ. **Εικονα 4.27** *Μέθοδος Εύρεσης Λέξεων*).

```
fun updateWordsFound(wordFound: String) {

    val dictionarySubLevelState = _dictionarySubLevelUiState.value
    val curDictionary = retrieveSubLevelDictionaryUi(dictionarySubLevelUiState = dictionarySubLevelState)
    val wholeDictionary = _wholeDictionary.value

    //If word in dictionary, add it to collection
    if (curDictionary.find { it.word == wordFound } != null) {
        updateFoundSetWords(wordFound = wordFound)
    } else if (wholeDictionary != null && wholeDictionary.find { it.word == wordFound } != null) {
        updateExtraSetWords(wordFound = wordFound)
    }

    //Updating SubLevelState Model for the word Found
    val colBoxes = _colBoxesUi.value
    if (colBoxes != null) {
        val newDictionary = mapFoundWords(dictionary = curDictionary, wordFound = wordFound, colBoxes = colBoxes)
        _dictionarySubLevelUiState.value =
            updateSubLevelDictionaryState(dictionarySubLevelUiState = dictionarySubLevelState, newDictionary = newDictionary)

        viewModelScope.launch  { this: CoroutineScope
            delay( timeMillis: 500)
            checkLevelCompletion(dictionary = newDictionary)
        }
    }
}
```

Εικόνα 4.27 *Μέθοδος Εύρεσης Λέξεων*

Τέλος η τελευταία από τις πιο βασικές μεθόδους αυτού του ViewModel αποτελεί η onRoundComplete(). Όπως περιγράφεται και από τον ορισμό της αφορά την ολοκλήρωση του επιπέδου, όπου αφότου γίνει ο έλεγχος ότι έχουν βρεθεί όλες οι λέξεις του υποεπιπέδου, με σκοπό την ενημέρωση όλων των στοιχείων του χρήστη. Κατά την ολοκλήρωση επιπέδου ελέγχεται αν το νέο υποεπίπεδο ξεπερνάει το μέγεθος την λίστας των υποεπιπέδων. Σε περίπτωση που το ξεπερνάει τότε ο χρήστης προχωράει στο επόμενο επίπεδο και προσθέτονται τα υπόλοιπα υποεπίπεδα εφόσον περισσεύουν σε αυτό. Με την ολοκλήρωση ενός επιπέδου δίνονται επιπλέον 40 νομίσματα και 1 επιπλέον ένδειξη και πέρα από αυτό ανανεώνονται τα στοιχεία του χρήστη. Τη μόνη διαφορά αποτελεί η ανανέωση διαφορετικού State, όπου όταν αναβαθμίζεται το επίπεδο αλλάζει η κατάσταση του DictionaryLevelUiState από Default σε Success. Αντιθέτως στην αναβάθμιση του υποεπιπέδου αλλάζει το DictionarySubLevelUiState από Default σε Success. Ακόμη ανάλογα με το QuestType της κάθε αποστολής , ανανεώνεται η πρόοδος των αποστολών (βλ. **Εικόνα 4.28** *Ανανέωση Στοιχείων Χρήστη – onRoundComplete()*).

```
viewModelScope.launch { this: CoroutineScope
    //Navigating to Next Level
    if (subLevel > totalSubLevels) {
        newUserCoins += 40
        newLevel += 1
        subLevel = subLevel - totalSubLevels + 1
        updateUserHintsUseCase(hints = userHints + 1)
        _userLevelUi.value = getUserLevelUseCase().firstOrNull()
        _dictionaryLevelUiState.value = DictionaryLevelUiState.SuccessLevelUiState

    //Navigating to Next SubLevel
    } else {
        _dictionarySubLevelUiState.value = DictionarySubLevelUiState.SuccessSubLevelUiState
    }
    updateUserLevelUseCase(level = newLevel)
    updateUserSubLevelUseCase(sublevel = subLevel)
    updateUserCoinsUseCase(coins = newUserCoins + extraWords.size + (10 * levelMultiplier))
    updateUserScoreUseCase(score = userScore + 5)
    updateUserBestTimeUseCase(newTimer = _timer.value)
    updateUserExtraWordsUseCase(extraWordsCount = extraWords.size)
    updateUserTotalSubLevelsUseCase(passedSubLevels = levelMultiplier)
    updateUserPassedLevelsUseCase(skippedSubLevels = levelMultiplier - 1)
    _userSubLevelUi.value = getUserSubLevelUseCase().firstOrNull()
    updateQuests(
        questLog = _questLogUi.value,
        newCoins = newUserCoins + extraWords.size + (10 * levelMultiplier),
        newSubLevels = levelMultiplier,
        newExtraLevels = levelMultiplier - 1,
        newExtraWords = extraWords.size,
        newHints = _hintsUsedUi.value,
        newTime = _timer.value.toInt()
    )
}
```

Εικόνα 4.28 *Ανανέωση Στοιχείων Χρήστη – onRoundComplete()*

GameScreen

Η Υλοποίηση του GameFragment είναι παρόμοια με το MenuFragment με ένα composeView που εμπεριέχει το composable GameScreen. Το GameScreen είναι στην κορυφή των composables καθώς μέσα σε αυτό υλοποιούνται όλα τα composables που σχεδιάζουν το ταμπλό του παιχνιδιού αλλά και τις αντίστοιχες λειτουργίες με τις οποίες ο χρήστης αλληλεπιδρά με αυτό (βλ. **Εικόνα 4.36** *Οθόνη Παιχνιδιού κατά την αλληλεπίδραση με τον χρήστη*).

Πρώτη προυπόθεση πριν την εμφάνιση του περιεχομένου είναι η δημιουργία ενός LaunchedEffect() με βάση το οποίο παρατηρούμε την κατάσταση των States (DictionaryLevelUiState , DictionarySubLevelUiState) με την χρήση των keys. Στην περίπτωση που αλλάξει το State τότε εκτελούνται οι μέθοδοι init() ή updateSubLevel() ανάλογα με ποια κατάσταση άλλαξε. Ενώ στην συνέχεια ανάλογα με την κατάσταση εμφανίζεται η ανάλογη οθόνη. Η διαχείριση του DictionarySubLevelUiState είναι ανάλογη με αυτή του DictionaryLevelUiState (βλ **Εικόνα 4.29** *Διαχείρισης State στο Composable*).

```
LaunchedEffect(key1 = viewModel.dictionaryLevelUiState) { this: CoroutineScope
    viewModel.init()
}

when (dictionaryLevelState) {
    DictionaryLevelUiState.LoadingLevelUiState -> LoadingScreen()
    DictionaryLevelUiState.SuccessLevelUiState -> GameSuccessScreen(
        userCoins = userCoins,
        userLevel = userLevel,
        userSubLevel = userSubLevel,
        userHints = userHints,
        userScore = userScore,
        imageUi = imageUi,
        totalSubLevels = dictionarySize,
        onDismissClick = { onLevelDismissClick() },
        extraWords = extraWords,
        gameTimer = gameTimer,
        levelMultiplier = levelMultiplier,
        progressBarValue = progressBarValue,
        levelUp = true
    )
    DictionaryLevelUiState.ErrorLevelUiState -> ErrorScreen()
    is DictionaryLevelUiState.DefaultLevelUiState -> HandleGameSubLevelState(
        viewModel = viewModel,
        dictionaryItemFound = dictionaryItemFound,
        onShowRandomChar = onShowRandomChar,
        imageUi = imageUi,
        userCoins = userCoins,
        userLevel = userLevel,
        userSubLevel = userSubLevel,
        userHints = userHints,
        userScore = userScore,
        wordsFound = wordsFound,
        dictionarySize = dictionarySize,
        gameTimer = gameTimer,
        onSubLevelDismissClick = { onSubLevelDismissClick() },
        extraWords = extraWords,
        onUpdateSharedBoxes = onUpdateSharedBoxes,
        progressBarValue = progressBarValue,
        levelMultiplier = levelMultiplier
    )
```

Εικόνα 4.29 *Παράδειγμα Διαχείρισης State Compose*

CrosswordKeyboard

Μια από τις μεγαλύτερες προκλήσεις σε επίπεδο User Interface είναι αυτή της υλοποίησης του πληκτρολογίου καθώς είναι απαραίτητη προυπόθεση αρχικά να δημιουργούνται δυναμικά τα κουμπιά και να τοποθετούνται σε κυκλική τροχιά προκειμένου να μπορεί ο χρήστης εύκολα και γρήγορα να σχηματίζει λέξεις καθώς είναι από τις πιο απολαυστικές αλληλεπιδράσεις κάθε χρήστη σε τέτοιου είδους εφαρμογές και κατά δεύτερον η υλοποίηση της δυνατότητας του χρήστη να ζωγραφίζει από γράμμα σε γράμμα και την διαχείριση των κινήσεων του με το δάχτυλο που θα εκτελεί ενέργειες όταν βρίσκεται πάνω από τα γράμματα.

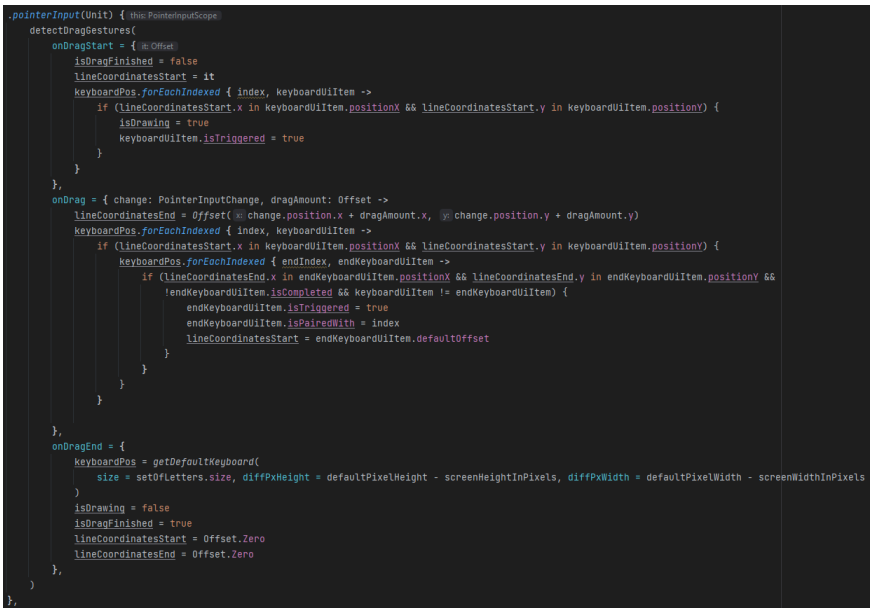
Αρχικά για την δημιουργία της κυκλικής τροχιάς των γραμμάτων στο πληκτρολόγιο καθώς δεν υπάρχει προεπιλεγμένη δομή, είναι απαραίτητη η δημιουργία ενός Custom Layout προκειμένου να μπορέσω να διαχειριστώ ακριβώς πως θέλω να τοποθετηθούν τα γράμματα. Με την χρήση του MeasureScope που αποτελεί μέρος του, μπορούμε να «μετρήσουμε» και να τοποθετήσουμε τα αντικείμενα στο σημείο που επιθυμούμε και σε συνδυασμό με το Θεώρημα Πολικών Συντεταγμένων μπόρεσα να τοποθετήσω τα γράμματα σε μια κυκλική τροχιά με το καθένα από αυτά να απέχει την μέγιστη δυνατή απόσταση μεταξύ τους (βλ **Εικόνα 4.30** *Τοποθέτηση Γραμμάτων σε Κυκλική Τροχιά*)

```
{ this: MeasureScope measurables, constraints ->

  val placeables = measurables.map { it: Measurable
      it.measure(constraints)
  }
  val angleIncrement = (2 * PI) / setOfLetters.size

  layout(constraints.maxWidth, constraints.maxHeight) { this: Placeable.PlacementScope
      val radius = constraints.maxWidth.toDp() / 3

      for (i in placeables.indices) {
          val angle = i * angleIncrement
          val buttonX = radius * cos(angle).toFloat()
          val buttonY = radius * sin(angle).toFloat()

          placeables[i].placeRelative(x = buttonX.roundToPx(), y = buttonY.roundToPx())
      }
  } ^Layout
```

Εικόνα 4.30 *Τοποθέτηση Γραμμάτων σε Κυκλική Τροχιά*

Για την υλοποίηση του σχηματισμού της λέξης και του ζωγραφιστού πληκτρολογίου έγινε χρήση του .pointerInput στο *Modifier* του *CustomLayout*, με αυτόν τον τρόπο μας δίνεται η δυνατότητα ανίχνευσης των events που προκαλούνται από τον χρήστη ενώ παράλληλα παρακολουθούμε τις συντεταγμένες του δακτύλου του καθόλη τη διάρκεια βρίσκεται μέσα σε αυτό το composable του πληκτρολογίου. Αρχικά επιλέγουμε να παρατηρήσουμε τις κινήσεις που αφορούν drag events, στην *onDragStart* ο χρήστης ξεκινήσει από κάποιο σημείο στο πληκτρολόγιο στο οποίο υπάρχει κάποιο κουμπί, θέτουμε σαν αρχή το κουμπί στο οποίο πάτησε αλλάζωντας την κατάσταση του για να εκτελέσει την λειτουργία του. Στην *onDrag*, παρατηρούμε τις μεταβολές στις συντεταγμένες στο δάχτυλο του χρήστη προκειμένου να ζωγραφίζουμε μια ευθεία από την αρχική θέση του *onDragStart* μέχρι την τρέχον θέση του ενώ στην *onDragEnd*, μηδενίζουμε όλες τις αποθηκευμένες καταστάσεις αυτής της διαδικασίας και αλλάζουμε την κατάσταση του *isDragFinished* σε true προκειμένου να επιστρέψει την σχηματισμένη λέξη για έλεγχο στο *GameViewModel* (βλ **Εικόνα 4.31** *Πληκτρολόγιο - PointerInput*). Όσον αφορά την υλοποίηση για τον ζωγραφισμό των ευθειών κατά την διάρκεια *onDrag* αλλά και όταν έχει ολοκληρωθεί ο προορισμός αξιοποίησα την .drawBehind του Modifier για να χρωματίσω το background του πληκτρολογίου, να ζωγραφίσω την τρέχων ευθεία που ο χρήστης

αυτή τη στιγμή δημιουργεί και τέλος τις ευθείες που έχουν ήδη σχηματιστεί από αφετηρία προς προορισμό (βλ. **Εικόνα 4.32** *Πληκτρολόγιο - drawBehind*).



Εικόνα 4.31 *Πληκτρολόγιο - pointerInput*

Εικόνα 4.32 *Πληκτρολόγιο - drawBehind*

CrosswordBoard

Η δεύτερη μεγαλύτερη πρόκληση ήταν ένας έξυπνος τρόπος τοποθέτησης των λέξεων, όπως και με το πληκτρολόγιο και σε αυτή τη περίπτωση δεν υπάρχει προκαθορισμένη αξιόπιστη δομή για τοποθέτηση των γραμμών και στηλών που να πραγματοποιείται δυναμικά και να λειτουργεί το ίδιο αποτελεσματικά και σε διαφορετικές οθόνες. Αντίστοιχα και σε αυτή την περίπτωση έγινε χρήση του *CustomLayout* προκειμένου να μπορεί να ελεγχθεί που θα μπει κάθε γραμμή και στήλη. Για την τοποθέτηση της πρώτης λέξης ελέγχθηκε η μεγαλύτερη λέξη από τις λέξεις του τρέχον επιπέδου και για την τοποθέτηση της χρησιμοποιήθηκαν οι διαστάσεις της οθόνης της συσκευής προκειμένου να μην υπάρχουν διαφοροποιήσεις από συσκευή σε συσκευή. Αφότου τοποθετηθεί η πρώτη λέξη στο ταμπλό είτε σε γραμμή είτε σε στήλη, κάθε επόμενη λέξη τοποθετείται σε αντίθετη δομή από την προηγούμενη δηλαδή σε περίπτωση που η πρώτη λέξη μπει σε γραμμή τότε η δεύτερη θα τοποθετηθεί σε στήλη και κάθε επόμενη αντίστοιχα. Ακόμα μετά την τοποθέτηση της πρώτης λέξης εκτελούνται δύο διαφορετικοί αλγόριθμοι με σκοπό το ταίριασμα κοινών γραμμάτων της προηγούμενης και της επόμενης λέξης.

Πρώτος αλγόριθμος αφορά τις λέξεις με μέγεθος έως και 4 γράμματα, σε αυτήν την περίπτωση είναι απαραίτητο πρώτα να γίνει έλεγχος των λέξεων των άκρων ενώ στην συνέχεια να γίνει έλεγχος για κοινά γράμματα στο ενδιάμεσο. Κύριος σκοπός αυτού του αλγόριθμου είναι να δημιουργηθεί μια «ουρά» όπου το πρώτο γράμμα της μίας λέξης είναι κοινό με το τελευταίο γράμμα της άλλης και αντίστροφα προκειμένου να ανοίξει το σταυρόλεξο και να αποφευχθει η σύγκρουση ανάμεσα στις λέξεις σε γράμματα που δεν είναι κοινά (βλ **Εικόνα 4.33** *Αλγόριθμος Τοποθέτησης Μικρών Λέξεων*).

Δεύτερος αλγόριθμος αφορά τις λέξεις με μέγεθος από 5 γράμματα και πάνω, μιας και σε αυτή την περίπτωση οι λέξεις αποτελούν μεγαλύτερο μάκρος αυτό έχει σαν αποτέλεσμα το σταυρόλεξο να μεγαλώνει παραπάνω όπου στην περίπτωση του πρώτου αλγόριθμου είναι πολύ εύκολο να ξεφύγει εκτός οθόνης όταν εμπεριέχονται πάρα πολλές λέξεις στο ίδιο επίπεδο.Με την χρήση αυτού του αλγορίθμου γίνεται έλεγχος των λέξεων σε όλο το μάκρος τους χωρίς να δίνεται προτεραιότητα στα γράμματα που αποτελούν τις άκρες των λέξεων (βλ **Εικόνα 4.34** *Αλγόριθμος Τοποθέτησης Μεγάλων Λέξεων*).

```
else if (previousWord.length < 4) {
  if (targetWord.first() == previousWord.first() && previousUsedPosition != previousWord.indexOf(previousWord.first())) {
      return Triple(first = 0, second = 0, third = targetWord.first())
  } else if (targetWord.last() == previousWord.first() && previousUsedPosition != previousWord.indexOf(previousWord.first())) {
      return Triple(first = 0, second = previousWord.length - 1, third = targetWord.last())
  } else if (targetWord.first() == previousWord.last() && previousUsedPosition != previousWord.indexOf(previousWord.last())) {
      return Triple(first = previousWord.length - 1, second = 0, third = targetWord.first())
  } else if (targetWord.last() == previousWord.last() && previousUsedPosition != previousWord.indexOf(previousWord.last())) {
      return Triple(first = previousWord.length - 1, second = previousWord.length - 1, third = targetWord.last())
  } else {
      for (targetLetter in targetWord) {
          if (targetLetter != targetWord.first() && targetLetter != targetWord.last()) {
              if (targetLetter == previousWord.first()) {
                  return Triple(
                      first = previousWord.indexOf(previousWord.first()),
                      second = targetWord.indexOf(targetLetter),
                      third = targetLetter
                  )
              } else if (targetLetter == previousWord.last()) {
                  return Triple(
                      first = previousWord.length - 1,
                      second = targetWord.indexOf(targetLetter),
                      third = targetLetter
                  )
              }
          }
      }
  }
```

Εικόνα 4.33 *Αλγόριθμος Τοποθέτησης Μικρών Λέξεων*

```
if (previousWord.length > 4) {
    if (previousIsRow) {
        for (previousLetter in previousWord) {
            if (previousWord.indexOf(previousLetter) != previousUsedPosition) {
                for (targetLetter in targetWord.reversed()) {
                    if (targetLetter == previousLetter) {
                        return Triple(
                            first = previousWord.indexOf(previousLetter),
                            second = targetWord.indexOf(targetLetter),
                            third = targetLetter
                        )
                    }
                }
            }
        }
    } else {
        for (previousLetter in previousWord.reversed()) {
            if (previousWord.indexOf(previousLetter) != previousUsedPosition) {
                for (targetLetter in targetWord) {
                    if (targetLetter == previousLetter) {
                        return Triple(
                            first = previousWord.indexOf(previousLetter),
                            second = targetWord.indexOf(targetLetter),
                            third = targetLetter
                        )
                    }

                }
            }
        }
    }
```

Εικόνα 4.34 *Αλγόριθμος Τοποθέτησης Μεγάλων Λέξεων*

Οι παραπάνω αλγόριθμοι επιστρέφουν δύο θέσεις που η πρώτη δείχνει την θέση του κοινού γράμματος στην προηγούμενη λέξη που έχει ήδη τοποθετηθεί και η δεύτερη δείχνει την θέση του κοινού γράμματος στην επόμενη λέξη που πρόκειται να τοποθετηθεί. Έχοντας τις κοινές θέσεις και των δυο λέξεων τοποθετείται η επόμενη λέξη στο ταμπλό και η τοποθέτηση γίνεται πάντα σε διαστάσεις ανάλογα με το μέγεθος της οθόνης προκειμένου να αποφευχθούν διαφοροποιήσεις από συσκευή σε συσκευή (βλ. **Εικόνα 4.35** *Τοποθέτηση Λέξεων στο Ταμπλό*). Έχοντας υλοποιήσει την τοποθέτηση των λέξεων δημιουργείται το επόμενο και τελικό πρόβλημα που έχει να κάνει με σύγκρουση δυο κελιών που τοποθετούνται το ένα πάνω στο άλλο, για την επίλυση αυτου του προβλήματος δημιουργείται ένα callback με τις θέσεις των κοινών κελιών προκειμένου να ανανεώνοται ταυτόχρονα όταν αποκαλύπτεται το κοινό τους γράμμα.

```
if (i % 2 == 0) {
    val pos = findLetterPos(
        pos = i,
        dictionaryUiItem = dictionaryUiItem,
        previousUsedPosition = previousUsedPosition,
        previousIsRow = previousIsRow
    )
    if (pos != null) {
        previousIsRow = !previousIsRow
        previousUsedPosition = pos.first
        placeables[i].placeRelative(x = xPosition - pos.second * staticSpacing, y = yPosition + pos.first * staticSpacing)
        colBoxes[i].previousPos = pos.first
        colBoxes[i].targetPos = pos.second
        colBoxes[i].char = pos.third
        xPosition -= pos.second * staticSpacing
        yPosition += pos.first * staticSpacing
    }
} else {
    val pos = findLetterPos(
        pos = i,
        dictionaryUiItem = dictionaryUiItem,
        previousUsedPosition = previousUsedPosition,
        previousIsRow = previousIsRow
    )
    if (pos != null) {
        previousIsRow = !previousIsRow
        previousUsedPosition = pos.second
        placeables[i].placeRelative(x = xPosition + pos.second * staticSpacing, y = yPosition - pos.first * staticSpacing)
        colBoxes[i].previousPos = pos.first
        colBoxes[i].targetPos = pos.second
        colBoxes[i].char = pos.third
        xPosition += pos.second * staticSpacing
        yPosition -= pos.first * staticSpacing
    }
}
```

Εικόνα 4.35 *Τοποθέτηση Λέξεων στο Ταμπλό*

Οθόνη Επιτυχίας

Με την εύρεση και της τελευταίας λέξης του υποεπιπέδου εμφανίζεται ένα *Dialog* με την χρήση του ανάλογου animation για την μεγέθυνση όπως και υλοποιείται ξεχωριστό animation για κάθε element του περιεχομένου για να εμφανίζονται σταδιακά. Στο περιεχόμενο αυτού του Dialog εμφανίζεται ο χρόνος που χρειάστηκε ο χρήστης για να ολοκληρώσει το επίπεδο, υπάρχουν δύο ProgressBars, το καθένα με το δικό του animation, όπου το πρώτο υποδηλώνει τα πόσα επίπεδα κατάφερε να προσπεράσει ο χρήστης ενώ το δεύτερο στο κάτω μέρος να υποδηλώνει το στάδιο που βρίσκεται ο χρήστης σε σχέση με το σύνολο των υποεπιπέδων προκειμένου να ολοκληρώσει το επίπεδο. Ακόμα αμέσως κάτω από τα επίπεδα που προσπέρασε φαίνονται οι ανταμοιβές που έχει λάβει που σχετίζονται με την εύρεση των λέξεων, την ολοκλήρωση του υποεπιπέδου και του επιπέδου. Ενώ με το πάτημα του κουμπιού “Next” μπορεί να προχωρήσει στο επόμενο επίπεδο (βλ. **Εικόνα 4.37** *Οθόνη Επιτυχίας Επιπέδου*).

Εικόνα 4.36 *Οθόνη Παιχνιδιού κατά την αλληλεπίδραση με τον χρήστη*

Εικόνα 4.37 *Οθόνη Επιτυχίας Επιπέδου*

Συμπεράσματα και Μελλοντικές Προκλήσεις

Κατά την γνώμη μου θεωρώ πως η ανάπτυξη μιας Android εφαρμογής παιχνιδιού είναι μια εμπειρία που ενσωματώνει την τεχνολογική και δημιουργική σκέψη. Μέσα από την αντιμετώπιση προκλήσεων και την εφαρμογή νέων γνώσεων, η δημιουργία ενός παιχνιδιού όχι μόνο προσφέρει ευχαρίστηση και ικανοποίηση στους χρήστες, αλλά και βελτιώνει τις δεξιότητες των προγραμματιστών. Παρόλο που η τεχνολογία έχει προχωρήσει σημαντικά, πολλές υπάρχουσες εφαρμογές σταυρολέξων για Android δεν εκμεταλλεύονται αυτές τις έξυπνες τεχνικές. Συχνά προσφέρουν μια στατική και λιγότερο ελκυστική εμπειρία, που δεν ενθαρρύνει τον χρήστη να συνεχίσει να παίζει ή να βελτιώνεται. Η έλλειψη καινοτομίας στον τομέα αυτό μπορεί να αποδοθεί σε διάφορους παράγοντες, όπως η έλλειψη επαρκούς έρευνας και ανάπτυξης, οι περιορισμένοι πόροι και η απουσία μιας ολιστικής προσέγγισης στο σχεδιασμό της εφαρμογής. Για το λόγο αυτό θεώρησα εξίσου σημαντικό να εισάγω έξυπνους τρόπους αλληλεπίδρασης οι οποίοι μπορούν να βελτιώσουν σημαντικά την εμπειρία των παικτών με τους εξής τρόπους: Μέσω της ανάλυσης των προτιμήσεων και των επιδόσεων του χρήστη, το παιχνίδι μπορεί να προσαρμόζει τη δυσκολία των σταυρόλεξων και να προσφέρει προτάσεις που ταιριάζουν καλύτερα στο προφίλ του. Επιπλέον, πρόσθεσα χρήσιμες οδηγίες που βοηθούν τον χρήστη να προχωρήσει χωρίς να απογοητεύεται. Ταυτόχρονα χρησιμοποίησα έξυπνους τρόπος αξιολόγησης της επίδοσης του χρήστη, προκειμένου να του παρέχω τη βέλτιστη δυνατή αλληλεπίδραση με το παιχνίδι με βάση τις δεξιότητες τους.

Ωστόσο παρατηρώ πως η ανάπτυξη μιας Android εφαρμογής δεν σταματά με την υλοποίηση της. Υπάρχουν συνεχείς προκλήσεις και παράγοντες που πρέπει να λαμβάνονται υπόψη, προκειμένου να διασφαλιστεί η επιτυχία και η βιωσιμότητα της εφαρμογής. Αρχικά κρίνω απαραίτητη την δημιουργία μια βάσης δεδομένων στην οποία θα αποθηκεύονται και ανακτόνται τα στοιχεία του χρήστη, το λεξικό καθώς και οι αποστολές του. Η συγκεκριμένη εφαρμογή αποτελεί μια προσπάθεια προσέγγισης του «Clean Architecture» και μελλοντικά στοχεύω στην διεύρυνση των γνώσεων μου στο κομμάτι αυτό. Ακόμη άλλοι στόχοι μου είναι η αύξηση του εύρους των λέξεων, η δημιουργία ήχου και ενδεχομένως η ενσωμάτωση τεχνητής νοημοσύνης για την αυτόματη δημιουργία σταυρόλεξων που είναι μοναδικά και προσαρμοσμένα στα ενδιαφέροντα του χρήστη.

Συνοψίζοντας, η παρούσα πτυχιακή εργασία αναλύει και παρουσιάζει την ανάπτυξη μιας εφαρμογής Android για σταυρόλεξα, η οποία ενσωματώνει έξυπνες τεχνικές για την αλληλεπίδραση με τον χρήστη. Η ανάγκη για πιο δυναμικές και εξατομικευμένες εμπειρίες στα παιχνίδια σταυρόλεξα επισημάνθηκε από την έλλειψη αυτών των χαρακτηριστικών στις υπάρχουσες εφαρμογές της αγοράς. Εν κατά κλείδι, η εργασία αυτή αποτελεί μια σημαντική συμβολή στον τομέα των ψηφιακών παιχνιδιών σταυρόλεξου και ανοίγει τον δρόμο για περαιτέρω έρευνα και καινοτομία. Ελπίζω πως οι ιδέες και οι τεχνικές που παρουσιάστηκαν, θα εμπνεύσουν και άλλους προγραμματιστές να δημιουργήσουν ακόμα πιο καινοτόμες και διασκεδαστικές εφαρμογές στο μέλλον.

Βιβλιογραφία

[1] Robert C. Martin *«Clean Architecture, A Craftmans's Guide to Software Structure and Design»*, Pearson Education, September 2017.

[2] Wei Sun, Haohui Chen, Wen Yu, *«The Exploration and Practice of MVVM Pattern on Android Platform»*, 4th International Conference on Machinery, 2016.

[3] Aymen Daoudi. Naouel Moha, Ghizlane ElBoussaidi, Sègla Kpodjedo, *«An Exploratory Study of MVC-based Architectural Patterns in Android Apps»*, Université du Québec à Montréal, 2019.

[4] Anirudh Sohil, Shikha Rastogi, Rajan Chawla *«Analysis for Performance Optimization of Android Applications»*,International Journal of Computer Science & Engineering Technology (IJCSET), March 2017.

[5] Mark H. Goadrich, Michael P. Rogers,*«Smart Smartphone Development: iOS versus Android»*, SIGCSE'11, March 2011.

[6] Selver Pepić, Marija Mojsilović, Muzafer Saračević, *«Android Game Development»*, 9 th International Scientific Conference Technics and Informatics in Education – TIE 2022, September 2022.

[7] Avisekhar Roy, *«The Android Game Developer's Handbook»*, Packt Publishing Ltd, August 2016.

[8] Nayab Akhtar, Sana Ghafoor, *«Analysis of Architectural Patterns for Android Development»*, Fatima Jinnah Women University, The Mall, Rawalpindi, Pakistan, September 2017.

[9] Smith, John. «*Mastering Android Development with Android Studio»*. Packt Publishing, 2020.

[10] Antonio Pachón Ruiz, *«Mastering Android Application Development»,* Pearson Education, September 2017.

[11] Bikesh Maharjan, *«Puzzle game using Android MVVM Architecture».* Metropolia University of Applied Science, April 2018

[12] Beselam Asefa, *«Building Android Component Library Using Jetpack Compose»,* Metropolia University of Applied Sciences, March 2022.

[13] Kevin Pelgrims, Caesar Eriksson, *«Gradle for Android»,* Packt Publishing, 2015.

[14] Anders Goransson, *«Efficient Android Threading, Asynchronous Processing Techiques for Android Applications»,* O'Reilly Media Inc., 2014.

[15] Yingjun Lyu, Jiaping Gui, Mian Wan, William G.J. Halfond, *«An Empirical Study of Local Database Usage in Android Applications»*, IEEE International Conference on Software Maintenance and Evolution, 2017.

[16] Giang Pham, «Develop maintainable animated Android applications», Metropolia University of Applied Sciences, April 2021.

[17] Rajkumar A. Soni, «A study paper on Android UI », International Journal of Enterprise Computing and Business Systems, 2013.

[18] Rishu Mishra, *«MVP (Model-View-Presenter) Architecture Pattern in Android with Example»*. Διαθέσιμο στον διαδικτυακό τόπο: https://www.geeksforgeeks.org/mvp-model-view-presenter-architecture-pattern-in-android-with-example/

[19] Kaalel, *«MVC Framework Introduction»*. Διαθέσιμο στον διαδικτυακό τόπο: https://www.geeksforgeeks.org/mvc-framework-introduction

[20] Rishu Mishra, *«MVVM (Model-View-ViewModel) Architecture Pattern in Android»*. Διαθέσιμο στον διαδικτυακό τόπο: https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android

[21] Tomas Repcik, *«Dependency Injection with Hilt in Android Development»*. Διαθέσιμο στον διαδικτυακό τόπο: https://tomas-repcik.medium.com/dependency-injection-with-hilt-in-android-development-e23fc636d65c

[22] Android Developers, *«Kotlin Coroutines on Android»*. Διαθέσιμο στον διαδικτυακό τόπο: https://developer.android.com/kotlin/coroutines

[23] Android Developers, *«The Activity Lifecycle»*. Διαθέσιμο στον διαδικτυακό τόπο: https://developer.android.com/guide/components/activities/activity-lifecycle

[24] Android Developers, *«Fragment Lifecycle»*. Διαθέσιμο στον διαδικτυακό τόπο: https://developer.android.com/guide/fragments/lifecycle

[25] Simona Milanovic, *«All About Preferences Datastore»*. Διαθέσιμο στον διαδικτυακό τόπο:https://medium.com/androiddevelopers/all-about-preferences-datastore-cc7995679334

[26] Android Developers, «App Manifest Overview». Διαθέσιμο στον διαδικτυακό τόπο: https://developer.android.com/guide/topics/manifest/manifest-intro

[27] Kausal Vasava, *«ViewModel's Internal working in Android»*. Διαθέσιμο στον διαδικτυακό τόπο:https://medium.com/@KaushalVasava/viewmodels-internal-working-in-android-a-full-guide-757afaf6510

[28] Kyparissiadis, A., van Heuven, W.J.B., Pitchford, N.J., & Ledgeway, *«GreekLex 2: A comprehensive lexical database with part-of-speech, syllabic, phonological, and stress information»,* University of Nottingham, 2017.