UNIVERSITY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Advanced Informatics and Computing Systems - Software Development and Artificial Intelligence»

ΠΜΣ «Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Development of a prototype of a mobile quiz game using cutting-edge tools and practices** Εφαρμογή σύγχρονων εργαλείων και πρακτικών για τη δημιουργία ενός πρωτότυπου παιχνιδιού ερωταπαντήσεων
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**Iason – Dimitrios Ntokouzis** Ιάσων – Δημήτριος Ντοκούζης
Father's name: Πατρώνυμο:	**Marinos** Μαρίνος
Student's ID No: Αριθμός Μητρώου:	ΜΠΣΠ/2222
Supervisor: Επιβλέπων:	**Efthymios Alepis, Professor** Ευθύμιος Αλέπης, Καθηγητής

September 2024 / Σεπτέμβριος 2024

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Efthymios Alepis **Professor**	**Maria Virvou** **Professor**	**Evangelos Sakkopoulos** **Assistant Professor**
Ευθύμιος Αλέπης Καθηγητής	Μαρία Βίρβου Καθηγήτρια	Ευάγγελος Σακκόπουλος Επίκουρος Καθηγητής

Contents

Abstract 3
Introduction 4
The importance of Object-Oriented Programming (OOP) 5
Example 1: Application with user management 5
Example 2: Sorting mechanism 8
Example 3: Receipt editing 10
Jetpack Compose and good practices 16
Old version of UI creation (xml files and listeners in UI) 16
Common issues that arise when developing UIs 16
Modifiers 22
App Class Structure 26
Introduction to the application 26
Synchronizing players with the API 27
Defining quiz classes, as well as questions and answers: 34
Analyzing and converting units of time 37
Web server implemetation 38
Conclusion 41

Abstract

The purpose of this publication was to create a quiz game that would allow players from around the world to connect, go through quizzes and answer questions together. The answers would be graded by the application. Every player's answer to every quiz needed to be recorded and stored in a storage space and players would have the ability to view statistical data about themselves, as well as worldwide statistical data.

The importance of Object-Oriented programming when creating, maintaining or expanding applications will be explained first. Approaches that can be taken into account, as well as perspectives that should be considered will be described. Finally, practices that have been created by the author for the purpose of writing well-organized code will be explained. The above aspects will be shown to the reader through example applications, created by the author for the purposes of this paper. The example applications are console applications without UI elements, in order to display and focus on the logic of the application, rather than its User Interface.

The basics of designing graphical applications with Jetpack Compose, will be explained in the next chapter. Jetpack Compose allows programmers or designers to develop user interfaces for mobile applications with a high degree of freedom, speed and with a minimal amount of bugs. Jetpack Compose is a toolkit that provides an entirely new way to develop modern graphical User Interfaces and it allows for the latest technologies to be used. The basics of layout creation, modifiers and the importance of the order in which they are applied will be explained with examples and images that will hopefully provide a deep understanding of the toolkit to the reader.

The last chapter will explain the structure of the application, its classes and how they are connected and the two different executables that are used to provide the runtime functionality. Design problems that were found and the solutions that were given for these problems will be explained. By the end of the chapter, the reader will have a comprehensive understanding of the general structure of the application.

Περίληψη

Ο σκοπός αυτής της δημοσίευσης ήταν η δημιουργία ενός παιχνιδιού ερωταπαντήσεων, το οποίο θα επέτρεπε παίχτες σε ολόκληρο τον κόσμο να συνδεθούν, να περάσουν μέσα από *quiz* και να απαντάνε μαζί σε ερωτήσεις. Η απάντηση του κάθε παίχτη σε κάθε *κουίζ* έπρεπε να καταγραφόταν και να αποθηκευόταν σε έναν χώρο αποθήκευσης και οι παίχτες έπρεπε να έχουν την ικανότητα να δουν στατιστικά δεδομένα για τον εαυτό τους, ή και να δούνε παγκόσμια στατιστικά δεδομένα.

Η σημασία του Αντικειμενοστραφούς Προγραμματισμού κατά τη δημιουργία, συντήρηση ή επέκταση εφαρμογών θα εξηγηθεί πρώτα. Θα περιγραφούν προσεγγίσεις που μπορούν να ληφθούν υπόψη, καθώς και προοπτικές που πρέπει να εξεταστούν. Τέλος, θα εξηγηθούν πρακτικές που έχουν δημιουργηθεί από τον συγγραφέα με σκοπό τη συγγραφή καλά οργανωμένου κώδικα. Τα παραπάνω θέματα θα παρουσιαστούν στον αναγνώστη μέσω παραδειγματικών εφαρμογών, που έχουν δημιουργηθεί από τον συγγραφέα για τις ανάγκες αυτής της εργασίας. Οι παραδειγματικές εφαρμογές είναι εφαρμογές κονσόλας χωρίς στοιχεία διεπαφής χρήστη, με σκοπό την εμφάνιση και την εστίαση στη λογική της εφαρμογής και όχι στη διεπαφή χρήστη (UI) της.

Οι βασικές αρχές του σχεδιασμού γραφικών εφαρμογών με το Jetpack Compose θα εξηγηθούν στο επόμενο κεφάλαιο. Το Jetpack Compose επιτρέπει στους προγραμματιστές ή σχεδιαστές να αναπτύσσουν γραφικές διεπαφές για κινητές εφαρμογές με υψηλό βαθμό ελευθερίας, ταχύτητας και με ελάχιστο αριθμό σφαλμάτων. Το Jetpack Compose είναι ένα εργαλείο που προσφέρει έναν εντελώς νέο τρόπο για την ανάπτυξη σύγχρονων γραφικών διεπαφών χρήστη και επιτρέπει τη χρήση των τελευταίων τεχνολογιών. Οι βασικές αρχές της δημιουργίας διατάξεων, των τροποποιητών (Modifiers) και η σημασία της σειράς με την οποία εφαρμόζονται θα εξηγηθούν με παραδείγματα και εικόνες που ελπίζουμε να προσφέρουν στον αναγνώστη βαθιά κατανόηση αυτού του εργαλείου.

Το τελευταίο κεφάλαιο θα εξηγήσει τη δομή της εφαρμογής, τις κλάσεις της και το πώς συνδέονται, καθώς και τα δύο διαφορετικά εκτελέσιμα αρχεία που χρησιμοποιούνται για την παροχή της λειτουργικότητας εκτέλεσης. Θα εξηγηθούν προβλήματα σχεδίασης που βρέθηκαν και οι λύσεις που δόθηκαν για αυτά τα προβλήματα. Μέχρι το τέλος του κεφαλαίου, ο αναγνώστης θα έχει μια ολοκληρωμένη κατανόηση της γενικής δομής της εφαρμογής.

Introduction

The problem to be solved was the development of a quiz game. It was necessary for this quiz game to have many responsibilities. For instance, the functionality of the application itself needed to be split into two parts: A web server and a mobile program. The web server needed to keep players connected and synchronized, while the mobile application needed to provide players an easy to use and invisible connection to the game server. In fact, the more invisible this connection to the web server was, the better. The server was also responsible for accepting and storing images in its database. The mobile application was also requested to provide images captured by users, as well as other types of images in a correct format to the web server, which would analyze and store them in its database for further use.

Object-Oriented Programming (OOP) needed to be used carefully and extensively to allow the code to be easy to maintain and change. Care needed to be used when developing classes, so that their design and implementation were symbiotic, and one didn't obstruct the other's operation. For instance, a badly designed class would not allow room for its implementation to be flexible at all and bad implementation would be too slow for any good design of classes to shine.

Three example applications were developed and analyzed. These applications will provide information of the basic Object-Oriented Programming practices that had to be created and used throughout the development of the quiz game. For every example application, its code, examples of its execution and diagrams pertaining to some specific functionality of these example applications will be displayed to the reader.

The mobile application's User Interface (UI) was developed with the use of Jetpack Compose, a relatively recent development in the world of mobile development, which provides a very safe and different way to create User Interfaces that the systems that came before it. Comparisons between older systems and Jetpack will be made and various parts of Jetpack Compose functionality, like *Modifiers* will be explained.

After the explanation of the basics of Jetpack Compose, the development of the mobile application and web server, as well as problems that were found will be explained. A way to describe quizzes, questions and answers needed to be found and imported as classes in the mobile application. Special care was placed in the web server's functionality, so that it would send messages quickly, using the TCP (Transmission Control Protocol), which comes standard in all ASP .NET web servers/API (Application Programming Interfaces). Technologies were developed from scratch, so that players could be kept synchronized with this protocol, as well as react to information, or events from the server, nearly in real-time. These technologies will be explained to the reader.

Other problems that were deemed difficult to solve will be explained after player synchronization, in the last chapters. More specifically, the way that the server stores images and retrieves them for players on demand is one of the parts of the web server's functionality that will be explained. The way that players are stored in the web server's database and memory, as well as the way that the server keeps players informed of events, will also be explained there.

Finally, it is worth noting that security measures have not been developed and were not a part of the development of the mobile application or the web server. The security of the software developed can be studied as a way to highly improve the current web server and mobile application.

The importance of Object-Oriented Programming (OOP)

Applications built using software engineering are becoming increasingly independent, partly in order to separate the costs of creating and maintaining them. It is considered good practice to keep applications tethered to each other as little as possible. Apart from applications becoming more independent and less error prone, the same trend can be noted for parts of every single application. Programs are split into parts, similar to how machines are composites of different and interconnected parts. The parts that make up a piece of software can be considered as similar to the pieces of a machine, but there are important differences between the two.

The components that a program is split into need to allow programmers to focus on only one of them and pay little to no attention on the rest when maintaining theirs, or someone else's code. This creates the question: What is the best way to split programs? Should a purely practical philosophy be used, in which the output of one piece feeds into the next, or can a more logical and perhaps more abstract approach be used? The first approach is easier to perform, but it doesn't provide the best level of separation between classes. The following example should make this more clear.

Example 1: Application with user management

Consider an application, which allows users to register an account or remove their account from a database. Let's assume there are two methods, one called register (username, password) that creates a new account with the specified credentials and a method called removeAccount(username) that deletes a created account. Let's assume that there is another method called otherFunctions() that represents all other functionality of the app. The sequence diagram could look like this:

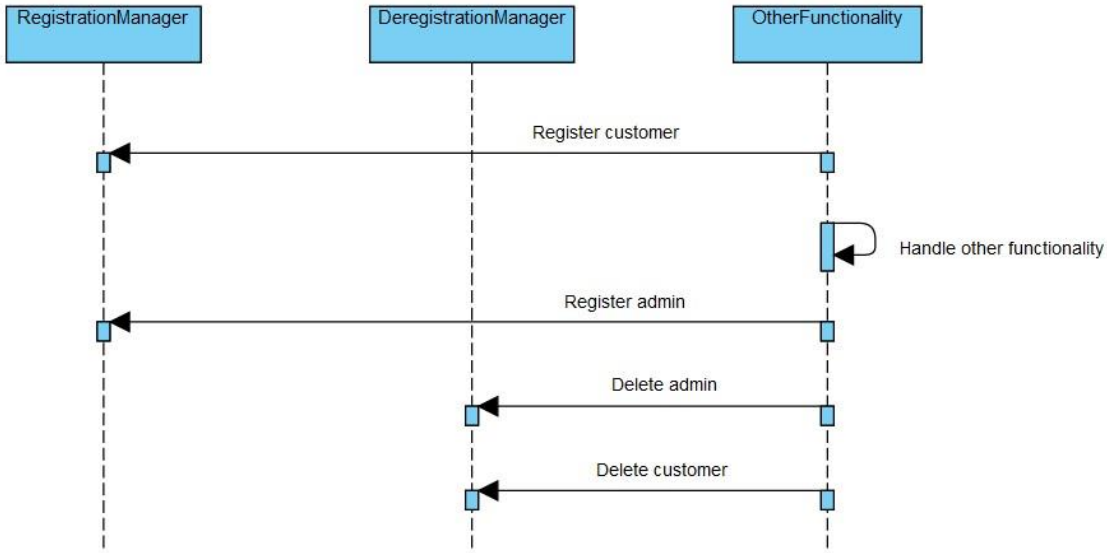


Figure 1: Sequence diagram of the application with registration manager and deregistration manager. Created using the free version of the web application Visual Paradigm.

The above diagram shows two basic classes, one for deleting all types of users and one for registering all types of users. The following diagram shows the two classes:

Figure 2: Class diagram for the example application. Created using the free version of the web application Visual Paradigm.

Here is an example of how the code of the application could be laid out:

```
class RegistrationManager {
    fun registerCustomer(username: String, password: String): Boolean
{
        if (doesUserAlreadyExist(username)) return false
        if (!isPasswordFormatCorrect(password)) return false
        users().put(username, password)
        saveChangesInDatabase()
        return true
    }

    fun registerAdmin(username: String, password: String): Boolean {
        if (doesUserAlreadyExist(username)) return false
        if (!isPasswordFormatCorrect(password)) return false
        if (!getAdminWritePermission()) return false
        administrators().put(username, password)
        saveChangesInDatabase()
        return true
    }

    private val adminManagementPermissions: AdminPermissions

}
```

```
class DeregistrationManager {
    fun deleteCustomer(username: String): Boolean {
        if (!doesUserAlreadyExist(username)) return false
        users().remove(username)
        saveChangesInDatabase()
        return true
    }

    fun deleteAdmin(username: String): Boolean {
        if (!doesUserAlreadyExist(username)) return false
        if (!getAdminWritePermission()) return false
        administrators().remove(username)
        saveChangesInDatabase()
        return true
    }

    private val adminManagementPermissions: AdminPermissions

}
```

It is possible that the methods and the object states (values in memory) used for the registration and deletion of admins can be very similar. In addition, both classes may need to get administrator management permissions. The similarity of the code between the two classes means that any code that has to be changed will have to be changed for both classes, and any mistake using the required permissions for admin management may end up creating or deleting admins when it is not requested, or induce other undefined behavior. A better way to split this program into classes may be to make one class responsible for registering/deleting administrators and the other class responsible for managing customers:

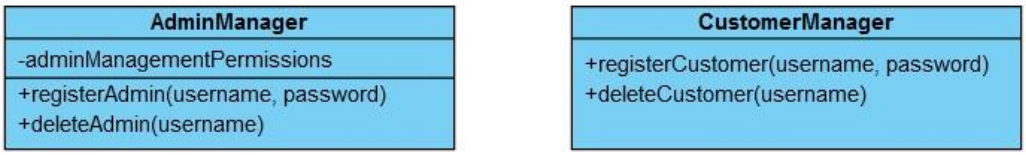


Figure 3: Class diagram of application after refactoring. Created using the free version of the web application Visual Paradigm.

In the above case, both classes don't need to be granted permissions for creating or removing administrators. Example code for this application is shown below:

```
class AdminManager {
    fun deleteAdmin(username: String): Boolean {
        if (!doesUserAlreadyExist(username)) return false
        if (!getAdminWritePermission()) return false
        administrators().remove(username)
        saveChangesInDatabase()
        return true
    }

    fun registerAdmin(username: String, password: String): Boolean {
        if (doesUserAlreadyExist(username)) return false
        if (!isPasswordFormatCorrect(password)) return false
        if (!getAdminWritePermission()) return false
        administrators().put(username, password)
        saveChangesInDatabase()
        return true
    }

    private val adminManagementPermissions: AdminPermissions

}
```

```
class CustomerManager {
    fun deleteCustomer(username: String): Boolean {
        if (!doesUserAlreadyExist(username)) return false
        users().remove(username)
        saveChangesInDatabase()
        return true
    }

    fun registerCustomer(username: String, password: String): Boolean
{
        if (doesUserAlreadyExist(username)) return false
        if (!isPasswordFormatCorrect(password)) return false
        users().put(username, password)
        saveChangesInDatabase()
        return true
    }
}
```

In the above code, there is less possibility for one class to influence the other. This means that there is more separation between the two classes and they can be maintained, expanded and compartmentalized independently.

The sequence diagram of the above version of the application is shown below:

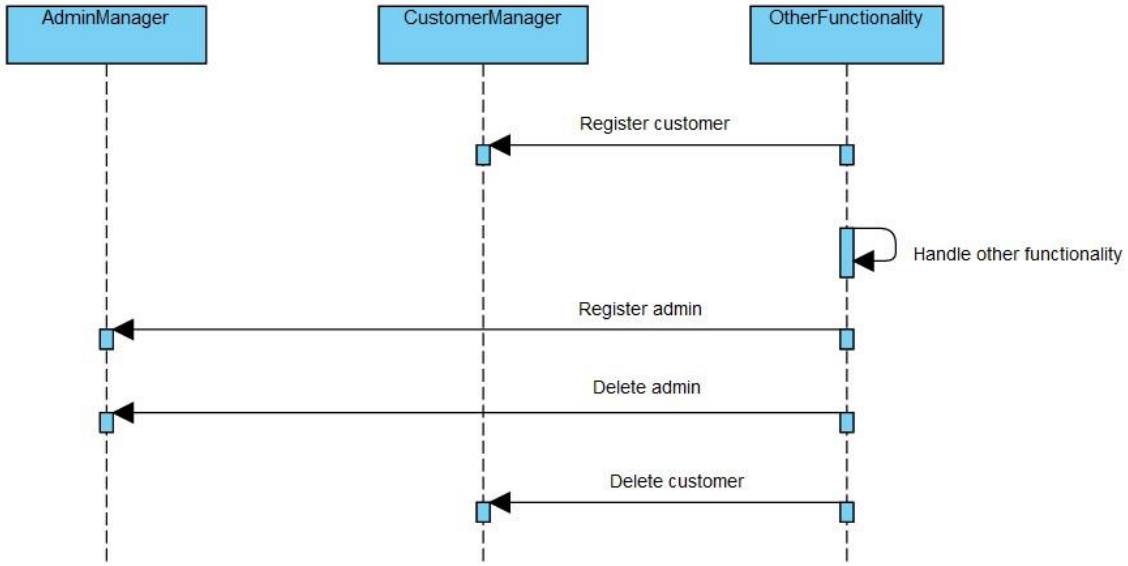


Figure 4: Class diagram of the application after refactoring. Created using the free version of the web application Visual Paradigm.

The above is an example of the programming practice called "The Single Responsibility Principle". According to this principle, it is not enough to only separate functionality of an application to different classes, but every class should have a single responsibility and a single area of application· common functionality should be kept in one class, instead of being repeated in many.

Example 2: Sorting mechanism

Consider an application that contains data structures and sorting mechanisms. More specifically, it contains a class with contains a data structure and is responsible for sorting it, using various sorting algorithms could be used to sort data structures. Similarly to the example above, two approaches will be explained. The class declaration in the first approach will have an interface that is too specific and the interface in the second example will be more flexible.

The class that will contain sorted collections will be named *SortedCollection* and it will contain collections of generic variables. Its purpose or responsibility is to sort collections. Its interface will consist of the following methods:

- splitIntoPartitions (): Splits the collection into two or more partitions, for each partition to be merged.
- mergeSortedPartitions (): Merges two sorted lists into a single sorted list.
- sortPartition (): Sorts one of the partitions that have been created after splitting.

```
class SortedCollection <Type> (val collection: List<Type>) {
    fun splitIntoPartitions(): List<List<Type>>
    fun sortPartition(partition: List<Type>)
    fun mergeSortedPartitions(
        partition1: List<Type>,
        partition2: List<Type>
    ): List<Type>
}
```

The class diagram of this application is the following:

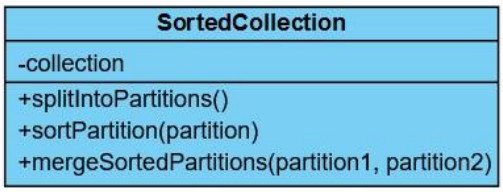


Figure 5: Class diagram of the first version of the example sorting application

In the above version of the program, it can be safely assumed that every sorting algorithm which will be used is an algorithm that recursively splits the collection into partitions, sorts the partitions and then merges them, creating the requested sorted list etc. This is effective for programs that only require one type of sorting algorithm, but if the program needs to be flexible enough to allow for any type of sorting algorithm, such an approach is inadequate.

Flexibility is one of the most important aspects of Object-Oriented Programming. If the classes that make up an application's codebase are effectively independent, they can be swapped out easily, without any fear of creating errors in the rest of the application, a practice which is similar to swapping out parts of a machine. If a machine is modular, parts can be removed and replaced easily. In addition, classes or groups of classes can be imported by being copied to new applications without any modification to their code and without changing variable names or values, as is commonly done when porting code, since the variables used will usually have a different purpose. For the reason above, it is generally a good idea to make an application as flexible as possible within the confines of the capabilities the program has been decided to have.

A more flexible approach would be to encapsulate algorithms into their own mechanisms/classes. This would allow every sorting algorithm to share a common interface, so that the application does not have to differentiate between different types of sorting algorithms. For instance, the common interface could be called *SortingAlgorithm*. It is also requested that the program be able to sort collections that contain any type of element, from primitive values like integers and strings to complex values like schools and restaurants. This creates problems in the sorting process, since objects of any type should be able to be compared and sorted. In the book "Design Patterns: Elements of Reusable Object-Oriented Software", it is stated that if an interface is not common between different object types, but a common interface needs to be found, perhaps a subset of the interface is common. For example, while it is not easy to find a common interface that allows integers, strings and all other objects to be sorted, it is possible to sort any collection, if it is known when one element in the list is considered larger than another. Therefore, a collection can be sorted if the sorting algorithm and the way to compare to elements in the list is given.

Utilizing the above limitations, the *SortedCollection* class holds collections and performs sorting on them, the *SortingAlgorithm* interface describes the way a collection is sorted and the *SortingComparator* interface describes when one element of the collection is larger than another. The combination of these three mechanisms allow any sorting algorithm to be used on any list. The code for this approach will be the following:

```
class SortedCollection <Type>(collection: List<Type>) {
    var collection: List<Type> = collection
        private set

    fun sort(
        algorithm: SortingAlgorithm,
        comparator: SortingComparator <Type>,
        descending: Boolean
    ) {
        collection = algorithm.sort(collection, comparator)
        if (descending) collection = collection.reversed()
    }
}
```

```
interface SortingAlgorithm {
    fun <Type> sort (
        list: List<Type>, comparator: SortingComparator<Type>
    ): List<Type>
}
```

```
interface SortingComparator <Type> {
    fun compare(value1: Type, value2: Type): Boolean
}
```

Example usage:

```
val list = listOf(1, 2)
val sortedCollection = SortedCollection(list)
sortedCollection.sort(
    RadixSort(),
    object: SortingComparator<Int> {
        override fun compare(value1: Int, value2: Int): Boolean =
            value1 > value2
    },
    false
)
```

Following is the class diagram for the above example:

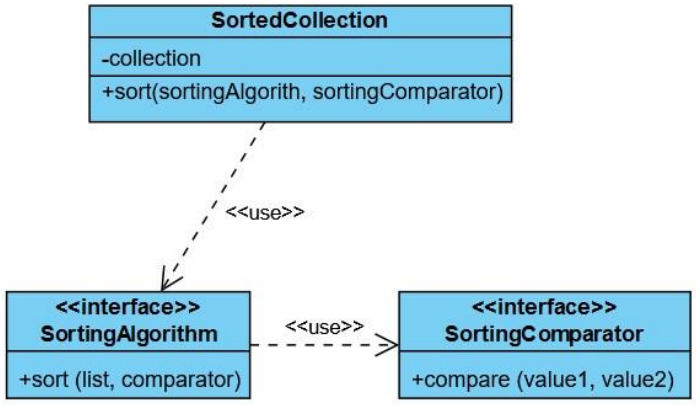


Figure 6: Class diagram of the seconds version of the example sorting application

Example 3: Receipt editing

For the final example, a theoretical application that creates, edits and prints receipts to the user's screen, using a console or terminal will be analyzed. This program contains a lightweight text document creator for creating receipt text. The connection between the receipt creator and the text creation mechanism of the application will be the focus of this example.

Creating the text of a receipt for printing is a complicated process, since it requires placing text in the middle, left or center of a text document. Text documents are two-dimensional text containers. Some of the typical actions, which can be performed in a text editor are placing text on the left, center, or right side of the currently open text document in the current line, removing text and changing lines. For the above actions to be planned out and programmed, precise calculations are generally needed. This is what will prove difficult in creating this kind of application.

The application should be able to create a purchase receipt and print it on the screen, as if it were printed on a real piece of paper. For instance, the receipt should contain the company's name at the top, written in the center of the top line. After that, the company motto and other useful information can be written with left alignment, then the products that were bought, their price and the total cost. Finally,

Some more information can be written after the product cost (i.e. Company Inc. 2024, All Rights Reserved). A receipt for an example company should be similar to the following image:

Figure 7: An example receipt. Not affiliated with Kit Kat.

Other companies may have different mottos, different products can be bought at different quantities or more products can be bought and the receipt should be able to show all of these products. This means that the application should be able to easily and continuously create these complicated pieces of text. Programmers should also be able to easily change how receipts are printed, without delving into the specific code that creates receipts. This requires a separation of the general logic and planning of receipt creation and its implementation in the code. If such separation does not exist, any change on how receipts are created would most likely involve copying and pasting specific code that has to do with receipt creation. In this case, more or less code than necessary can be copied, leaving behind useless code or creating errors, due to the lack of understanding by the programmer responsible for editing the current code.

To understand how this separation can be achieved, it is useful to first separately define the specific functions that should be executed for the design and creation of receipts. Designing receipts should be based on placing text in the left, right or center of the current line, as well as proceeding to a new line in a text document of a fixed width. Creating receipts should be based on keeping informed of the current position in the text document, adding the correct amount of spaces so that text can be placed in the correct alignment, placing many pieces of text with different alignments in the same line and keeping the document currently being created in memory.

An effective way to separate a specific mechanism from the more abstract logic relevant to creating text documents is to split the two into two separate classes. One class will send the necessary commands and the other class will execute them, performing any necessary actions to create the requested text document. It is necessary for the implementation to follow these commands, which means that it has to be able to follow all types of commands given to it by the class responsible for sending them.

For this example application, the class responsible for planning is called *Receipt* and the class responsible for creating the necessary text containers is called *TextContainer*. A successful separation of the two means that creating a text container can be as simple as the following code:

```
val textContainer = TextContainer(70)
textContainer.placeText(
    "This text is located in the center of the line",
    TextAlignment.CENTER
)
textContainer.changeLine()
textContainer.placeText(
    "This text is located in the left side",
    TextAlignment.LEFT
)
textContainer.changeLine()
textContainer.placeText(
    "This text is located in the right side",
    TextAlignment.RIGHT
```

```
)
println(textContainer.getText())
```

Running the above code gives the following output:

```
            This text is located in the center of the line
This text is located in the left side
                                This text is located in the right side
```

Figure 8: Text that has been placed in all three possible alignments with the newly developed lightweight text editor

To achieve this simplicity, the *TextContainer* class needs to follow any command or series of commands sent to it. This means that the container class needs to be in a position to follow all commands, so that the design can easily change without errors being caused. It also needs to have a simple interface, regardless of the complexity of its actions. This is what allows the receipt class to use simple commands to design the text container.

The following is the class diagram of this example application:

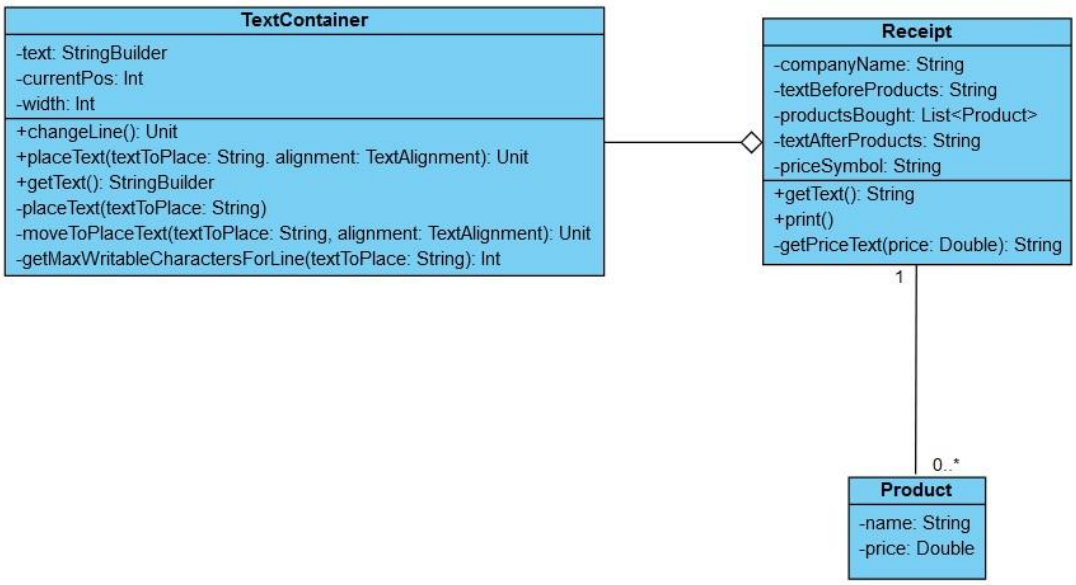


Figure 9: Class diagram of receipt creation application

The sequence diagram of the following application is the following:

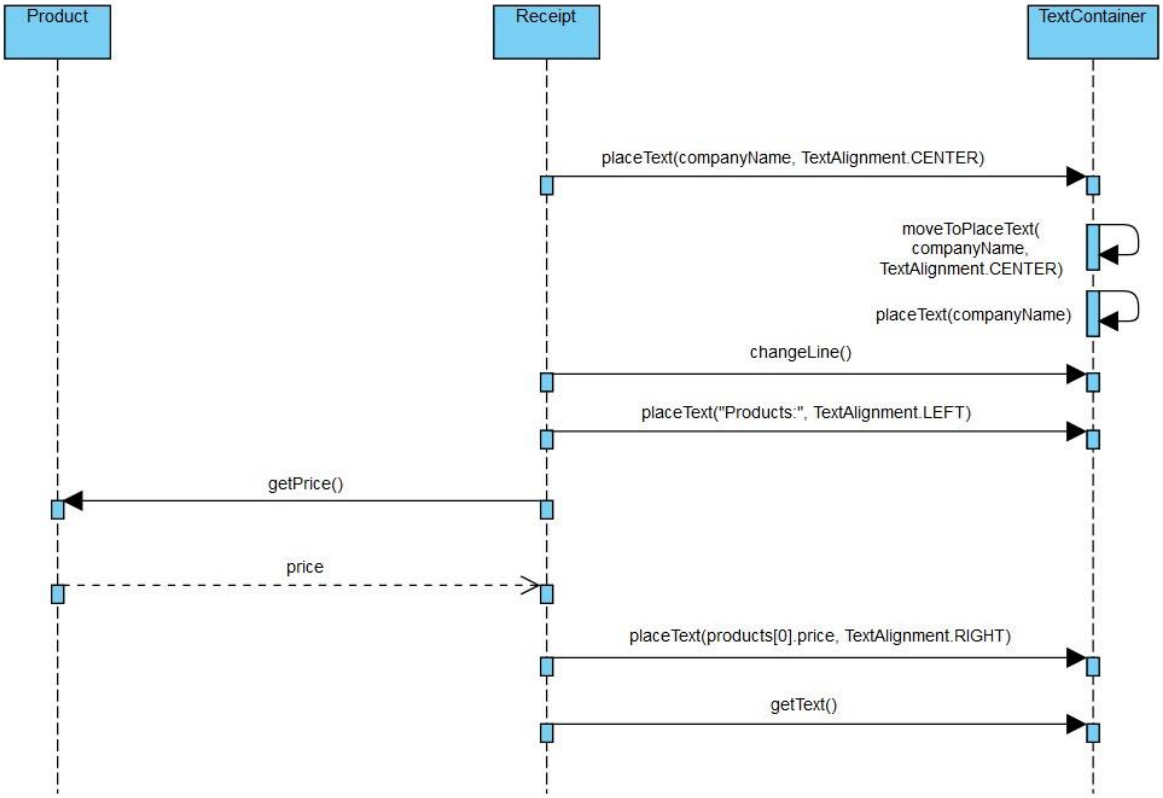


Figure 10: Sequence diagram of receipt creating application

As stated above, the *Receipt* class sends the general commands to the *TextContainer* class, which creates the receipt. After creation, the *Receipt* class edits the text created by the editor and creates borders around the receipt, so that it can look more natural after being printed. It performs those two functions with its two methods, the *getText()* method that creates the structure of the receipt and places any necessary texts and the *print()* method that adds the borders and prints the receipt on the screen. The code that is used by the Receipt class is shown below:

```
class Receipt(
    private var companyName: String,
    private var textBeforeProducts: String,
    private var productsBought: List<Product>,
    private var textAfterProducts: String,
    private val width: Int,
    private val priceSymbol: String = "€"
) {
    fun getText(): String {
        val textContainer = TextContainer(width)
        textContainer.placeText(companyName, TextAlignment.CENTER)
        textContainer.changeLine()
        textContainer.changeLine()
        textContainer.placeText(textBeforeProducts,
TextAlignment.LEFT)
        textContainer.changeLine()
        textContainer.changeLine()
        textContainer.placeText("Products:", TextAlignment.LEFT)
        var totalPrice = 0.0
        productsBought.forEach {
            textContainer.changeLine()
            textContainer.placeText("${it.name}:", TextAlignment.LEFT)
```

```
            textContainer.placeText(
                "${getPriceText(it.price)}${priceSymbol}",
                TextAlignment.RIGHT
            )
            totalPrice += it.price
        }
        textContainer.placeText("Total cost:", TextAlignment.LEFT)
        textContainer.placeText(
            "${getPriceText(totalPrice)}$priceSymbol",
            TextAlignment.RIGHT
        )
        textContainer.changeLine()
        textContainer.changeLine()
        textContainer.placeText(textAfterProducts, TextAlignment.LEFT)
        return textContainer.getText().toString()
    }

    private fun getPriceText(price: Double) =
        when (floor(price)) {
            price -> price.toInt().toString()
            else -> price
        }

    fun print() {
        val text = StringBuilder("_".repeat(width + 2))
        text.append("\n|")
        text.append(getText().replace("\n".toRegex(), "|\n|"))
        text.append("|\n")
        text.append("¯".repeat(width + 2))
        println(text)
    }
}
```

The *TextContainer* class needs to be in a position to follow every given command, like writing text with left, center or right alignment. It should move the cursor to the correct position and place text at the correct position, according to the requested alignment, move to the end of the current line and calculate how many characters of a text can be written in the current line, according to the current line and position of the cursor in the text container. The code for the *TextContainer* class is shown below:

```
class TextContainer(
    val width: Int
) {
    private var currentPos: Int = 0
    private var text: StringBuilder = StringBuilder()

    init {
        if (width <= 0) throw IllegalArgumentException()
    }

    fun placeText(textToPlace: String, alignment: TextAlignment) {
        var textToPlaceBuilder = StringBuilder(textToPlace)
        while (textToPlaceBuilder.length > 0) {
            val maxWritableCharacters =
                getMaxWritableCharactersForLine(
                    textToPlaceBuilder.toString()
                )
            val textToPlaceInCurrentLine =
                textToPlaceBuilder.subSequence(
                    0, maxWritableCharacters
                )
```

```
            try {
                moveToPlaceText(
                    textToPlaceInCurrentLine.toString(), alignment
                )
            }
            catch (ex: IllegalArgumentException) {
                changeLine()
                continue
            }
            catch (ex: IndexOutOfBoundsException) {
                changeLine()
                continue
            }
            placeText(textToPlaceInCurrentLine.toString())
            textToPlaceBuilder = StringBuilder(
                textToPlaceBuilder.removeRange(0,
maxWritableCharacters)
            )
        }
    }

    fun getText(): StringBuilder {
        val posToMove = width - " ".length
        val unitsToMove = posToMove - currentPos
        val textCopy = StringBuilder(text)
        textCopy.append(" ".repeat(unitsToMove + 1))
        return textCopy
    }

    fun changeLine() {
        val unitsToMove = width - currentPos
        text.append(" ".repeat(unitsToMove))
        text.append("\n")
        currentPos = 0
    }

    private fun placeText(textToPlace: String) {
        text.append(textToPlace)
        currentPos += textToPlace.length
    }

    private fun moveToPlaceText(
        textToPlace: String,
        alignment: TextAlignment
    ) {
        var posToMove = currentPos
        if (alignment == TextAlignment.CENTER) {
            posToMove = (width - textToPlace.length) / 2
        }
        else if (alignment == TextAlignment.RIGHT) {
            posToMove = width - textToPlace.length
        }
        if (posToMove < currentPos) {
            throw IllegalArgumentException()
        }
        if (posToMove == width) {
            throw IndexOutOfBoundsException()
        }
        val unitsToMove = posToMove - currentPos
```

```
        placeText(" ".repeat(unitsToMove))
    }

    private fun getMaxWritableCharactersForLine(textToPlace: String) =
        min(textToPlace.length, width - currentPos)
}
```

Jetpack Compose and good practices

Jetpack Compose is a UI toolkit that provides a modern way of developing and designing android applications. This toolbox creates new possibilities when creating user interfaces. The specifics of this toolkit prevent many common errors when creating user interfaces for applications, make it easy to manage UI code. Compose provides an entirely new perspective to showing information on the user's screen.

Old version of UI creation (xml files and listeners in UI)

Android application creation used to happen through xml files that usually described the entire screen, along with its buttons, text fields, etc. and code was always written so that when data changed, any screen elements that showed relevant data were updated to show the latest values. In other words, views had to be bound to the data hey showed programmatically and this had to happen every time a new application was developed. This was inconvenient due to the large amount of space that these data bindings take up. Elements of screens, also called views, were placed individually based on static or relative positions and any element needed to be placed and connected to its neighboring views.

Common issues that arise when developing UIs

Consider a new application with a graphical component, like a window or a screen. Buttons and pieces of text need to be placed and shown on the user's screen. How should the elements be placed? What constitutes a UI element? How does the UI system place elements in different positions? One of the first solutions to the problem of positioning that were discovered defined an element's position as two numbers. One number would describe the position of the element on the horizontal axis and the other, its position on the vertical axis. This system is still in use today in windows forms applications, written in C#. Similar systems can be used in game development.

The above positioning system in use by *Windows forms* applications would be considered highly deprecated for the creation of user interfaces, if they were to be used in android development. User interface development in Android is typically much more advanced than graphical development for computers. There are multiple reasons for this:

1. Computer screens have specific sizes: Even with the invention of 4K screens, common computer screens in the market typically have the following sizes:
    a. 720p (1280 x 720)
    b. 1080p (1920 x 1080)
    c. 1440p (2560 x 1440)
    d. 4K (3840 x 2160)

    These sizes are stable enough to have applications that don't need to pay special attention to alterations of screen size. When the graphical application starts, the user's screen size is retrieved from the operating system, and the application screen is resized accordingly. It is not common for a user to change the screen resolution, so the application does not need to be extremely responsive in detecting screen size changes.
2. Development on mobile machines is more demanding: When applications first started getting developed for android, a unique problem was revealed. How would these applications be effective at showing all the necessary information in the relatively small screen space that mobile devices provide, as well as be battery efficient? These challenges meant that more focus had to be placed on how these applications could be created for mobile devices. Computers do not usually place an emphasis on battery efficiency and latest technologies don't have to be used.

3. Development for mobile devices is more common: The creation of applications for desktop PCs is not as common as development for mobile devices, as mobile devices are used much more commonly. A desktop PC may be walked away from, but a mobile device rarely leaves its user's side pocket. Due to the increased usage of mobile devices, a large portion of the global market has moved to developing and selling mobile applications. Therefore, more focus is placed on this area, more recent technologies are used and any technologies used are advanced more rapidly.

Screen size is an important factor that has contributed to the advancement of mobile development, because of the myriads of screen sizes that mobile devices can have, as well as the rapid resizing of the screen that can happen when the application is running. One of the ways the screen size may change is by flipping the mobile device, at which case the device's display mode switches from portrait to landscape, or vice versa. More specifically, when the roll of the mobile device is altered by 90 degrees, the application needs to be rotated, so that any information on the screen can still be eligible to the user. For instance, when an application is first launched, it may be launched in portrait mode:

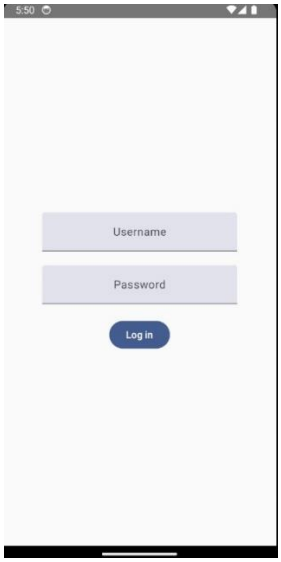


Figure 11: A typical android application launched in portrait mode.

When the user rolls their phone, the application may change to landscape mode, in which case the UI will look like this:

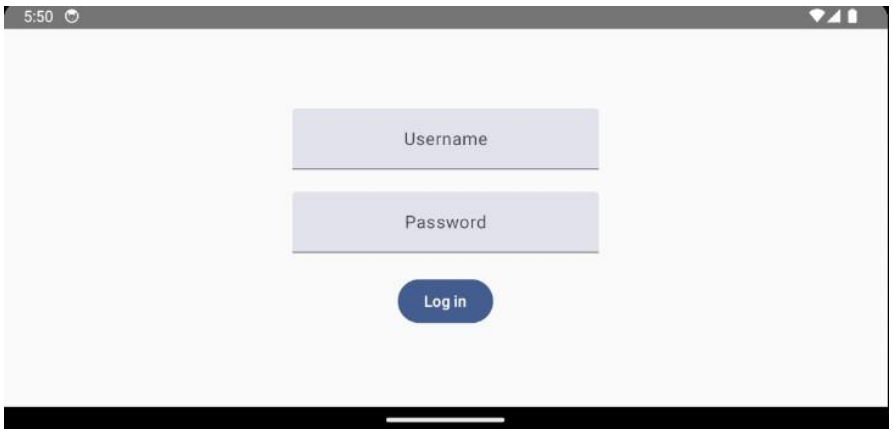


Figure 12: An android application that is running in an android device in landscape mode.

After rotating, the application needs to be rotated as well to fit to the rotated screen. The screen that this application uses would then have a different width and height. More specifically, the width of the screen in portrait mode is the height in landscape mode and the height in portrait mode is the width of the screen in landscape mode. After rotating, the application technically uses a new screen, since the resolution is different. In mobile devices, user actions like rotating the screen provide an immediate alteration to the screen size. The rapid alterations of the screen size mean that new technologies need to

be used that keep elements in the same position on screen, no matter its resolution. Consider an application that takes 200 pixels of the screen horizontally and vertically with a point in the middle of the window. This point should have the coordinates (100, 100). If instead of a point there is an image on the screen with both a width and height of 20, its position should be (90, 90) to be in the middle. In the event more pixels are used, either by resizing the window or changing the screen resolution, the element with the above coordinates will stop being in the middle of the screen. A new position will need to be provided to the element, so that it can be moved to the center. These two problems of position and size were also met during video game creation, due to games needing to work on multiple screen resolutions and have many window sizes.

The solution to this problem is the addition of relative positions to views in graphical applications. This system uses percentages to keep elements in the center or above, under, to the left or to the right of the center in all screen sizes. For instance, an element in the center of the screen would have 50% as its horizontal and vertical percentage. An element might be more to the right of the center, by using something larger than 50% as its horizontal percentage.

The size of elements also needs to be the same in all screen sizes. Therefore, sizes in pixels are not commonly used in mobile applications. Instead, sizes are calculated with a different unit of measurement, called dpi (dots per inch). This unit of measurement calculates the size of the element, as seen by the user and any element that has a size measured in dpi will have the same size in all screen types and resolutions.

A new advancement to this system allowed programmers to choose which elements used relative positions and which elements used absolute ones. Elements could also use absolute horizontal position and relative vertical position, or vice versa, as well as use absolute positions, measured from another corner of the screen, instead of the top-left corner. This is what the older android UI system, which uses *xml* files to describe the appearance of the window uses.

Even though the system above is more advanced, it is still not the system that is being used today. The older system had many common problems when designing an application. Deep knowledge was required to understand the basics of the UI design, such as relative and absolute positions, as well as knowledge of the strengths and weaknesses of every type of position and size when designing UIs. A lack of knowledge would lead to errors, which appeared when the screen became larger or smaller. These issues typically revolved around elements being pushed too close together and there not being enough space to show all the required information. Due to these common issues, a practice that was used was that User Interfaces were first developed for the smallest possible screens. If the UI was functional for these screens, it was safe enough to assume that there would be enough space for all the information to be shown in larger screens. Apart from that practice, testing for larger screens was still useful for app production, due to some errors that could still arise.

The challenge that was met when developing UI for mobile applications, as well as the low speed of developing mobile applications led to the creation of *Jetpack Compose*. As mentioned above, *Jetpack Compose* provides a completely new way to develop UIs for mobile applications and handles UI creation in such a way that programmers don't need to worry about common errors. *Compose* requires programmers to state how views should be laid out on screen without paying attention to absolute or relative positions of elements, or how elements are connected to each other. This way, programmers don't have to have a deep knowledge of how the elements are placed.

In *Compose*, programmers describe how elements will be laid out on screen as a group, instead of handling each element on its own. The groups are responsible for laying out elements on the screen. For example, if elements are placed in a column, they will all be laid out on a vertical line. The types of layouts will be explained below:

- Column: Elements are placed on a vertical line with one element above another. The first element is placed at the topmost point. Compose allows for the elements to be grouped on the top, middle, or bottom of the column. In addition, the line of elements can be placed on the left, middle or right side of the column.

Figure 13: Elements placed in a column

- Row: Elements are placed on a horizontal line with one element next to the other. The elements can be placed at the left, middle or right size of the line and the line can be moved to the top, middle or bottom of the row.

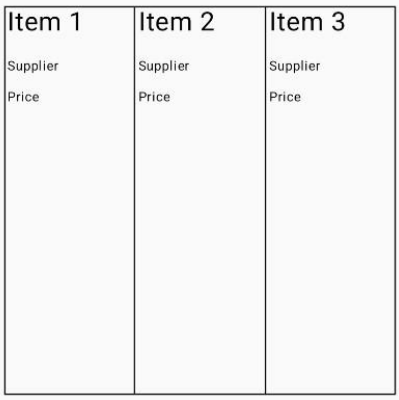


Figure 14: Elements placed in a row

- LazyColumn: Elements are placed in a column layout, with only the visible elements being loaded into the applications memory. This would be represented by a *RecyclerView* in the older UI architecture, which would use a vertical linear layout (Smyth, 2023). The loading of only the visible elements helps reduce resources used and is usually utilized if large amounts of data need to be displayed in a column. For instance, a lazy column could be used to display books in a digital bookstore. The name lazy column comes from lazy loading, which means that elements are only loaded when needed.
- LazyRow: Same as the above layout, but elements are placed in a row. Lazy rows and columns have built in scrolling support, so users can scroll vertically or horizontally without programmers needing to add or program that feature in the lazy loaded layouts.
- Box: Elements are placed in the same area and one element is placed on top of another. This helps when views need to occupy the same space. For instance, if an error message is placed on top of the application screen, it is safe to assume that the box layout is being used. The below images display the layers of UI elements that are placed one on top of the other, to create a box layout:

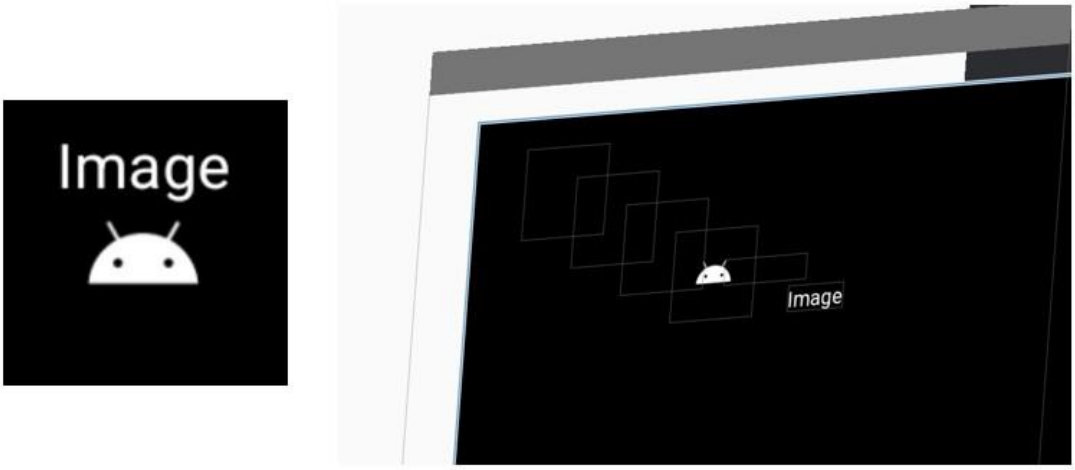


Figure 15: Box that places a text on top of an image on a black background.

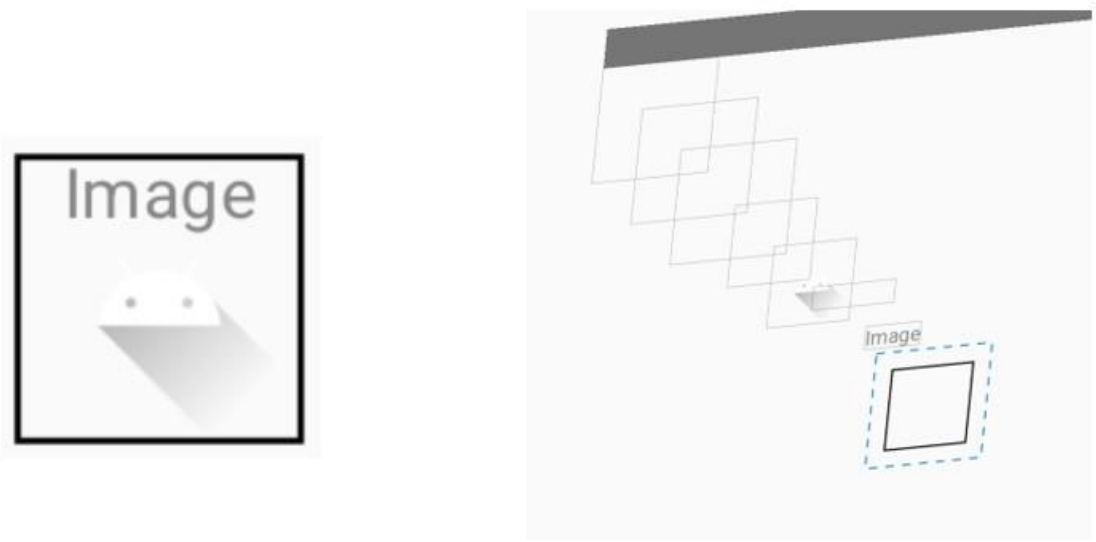


Figure 16: Box that places text on an image in a white background.

- LazyHorizontalGrid: Elements are placed in a grid, forming equally sized columns. Elements placed in the rows can be configured to all have fixed size, or they can fill as much vertical space as possible (dynamic size), allowing every column to have a different number of elements. Lazy loading is used and only elements that need to be displayed are loaded. Scrolling is automatically implemented.

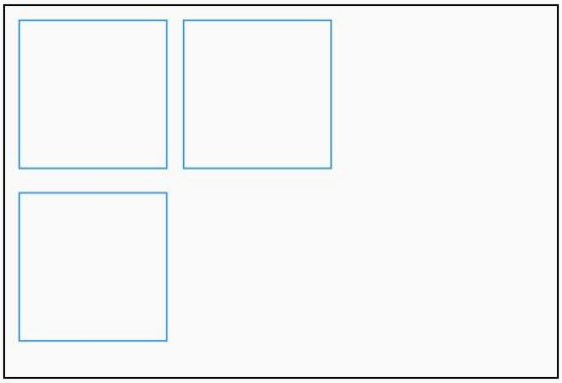


Figure 17: A lazy horizontal grid. Note how the grid is filled with squares, column by column

- LazyVerticalGrid: Elements are placed in a grid with equally sized rows. Elements in rows can have fixed or dynamic sizes. Scrolling is automatically implemented.

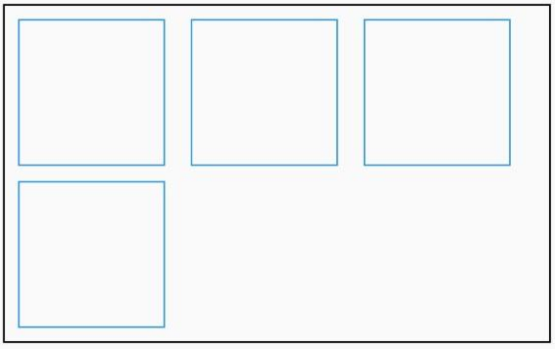


Figure 18: A lazy vertical grid. Not how the elements fill the grid row by row

This kind of grid is used for the quiz game that was developed. Specifically, it was used to present the possible answers a user could choose for a question on screen.

Figure 19: A question, as shown in the mobile application, in the duration of a quiz game

When programmers place views in a group, the views are considered to be children of the group and the group is considered to be their parent. Groups are considered views themselves. This functionality is a direct implementation of the *Composite* design pattern, where an element and a group of elements have the same interface and are used interchangeably in many parts of the code. Parent views decide the position of any child views. This is evident when trying to create a custom layout. In addition, when creating a custom layout, a programmer needs to define a mechanism for calculating the position of its child elements, but changing their sizes is not a part of the standard *Jetpack Compose* custom layouts. Any views that are not placed in a group are considered to have the screen as their parent.

As mentioned above, groups decide the position of their children. Any Jetpack Compose coders have to decide how to group their views. The grouping will decide the final layout of the elements and their position on screen. Jetpack Compose avoids common UI errors by removing the responsibility of the placement of elements from the programmer. A programmer only has to use or define a layout. It is the toolkit's responsibility to place the UI elements and connect them to the screen and to each other, when necessary, in order to achieve the requested layout. Coders do not have to explain how the elements will be placed in a specific layout, but only how they would like them to be placed, making Compose a more

declarative type of toolkit than the older systems for UI creation, which are more imperative. As for any problems that arise when the screen size is too small or too large, or when it is not easy to place elements in a UI, *Jetpack Compose* is hopefully capable of solving them on its own.

It is possible for coders to use older systems for UI creation through *Jetpack*. It is worth noting that *Jetpack* can create most UIs by using only layouts provided by the toolkit. However, if a large amount of control is needed over the UI, or when a specific UI is simply not possible by using the standard layouts provided by Compose, the usage of old-school *Constraint* layouts of the old *xml*-based system, in which programmers can place elements using relative positions, connect controls with each other and also move elements using absolute positioning with the usage of pixels is possible to be used in this toolkit. In cases where enough control to place lines, create polygons using pixel perfect precision is required over the entire screen, or a specific part of it, it can be achieved using *Compose's* Canvas layout. This is used for graph creation, as well as the creation of any shapes that may be requested but are not a part of standard Compose functionality.

The UI in *Jetpack* consists of smaller UI elements that have been grouped together. These UI elements are called composables. The ability of composables to be separated from one another, allows for composables to be developed and tested independently from others, similar to how classes in Object-Oriented programming are separated. As mentioned above, groups are interchangeable with their children, which is an implementation of the *Composite* design pattern (Gamma, et al., 1994).

Modifiers

When a view or composable is placed on screen, it is possible that some of its properties will need to be changed. In the old *xml*-based system, it was possible to select a view in the designer tab and edit all its properties. This method required the programmer to scroll through the entire list of the view's properties, or use the search bar to search for a specific property. In *Compose*, the properties used in the old system are called *modifiers*. Views are always considered to have default modifiers and programmers explicitly state which properties they want to change from the default ones. This allows for programmers to only type in and view properties that are altered, leaving default properties out of the view of programmers and helping reduce boilerplate code. Programmers are also capable of moving, resizing and clipping views using modifiers. Some basic modifiers are:

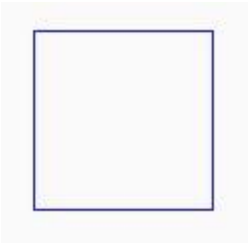
- fillMaxWidth(): The view will fill as much horizontal space as allowed by its parent. If there is no parent, the view's width will fill the entire screen.
- fillMaxHeight: The view will fill as much vertical space as allowed by its parent or the screen.
- fillMaxWidth: The view will fill as much space as allowed by its parent or the screen.
- padding: Makes the view smaller and places empty space around it. The programmer can choose how much empty space will be added in each cardinal direction. There are also overloads to this that allow for more convenient placement of padding. For instance, instead of the padding in each direction being set separately, the padding can be set for all directions using one value.
- clip: Clips the view using a specific shape. For instance, if a profile picture is square, but it needs to be cut as a circle, the clip method can be used to remove the corners and make the picture circular.
- size: The view is resized to the requested amount. This method only accepts measurements in dpi.
- offset: The view is moved by the requested amount. The view can be moved in all cardinal directions. However, any layout that contains this element will not change the placement of other elements in its layout and the view that was offset can end up being the odd one out in the layout. This means that in the case of moving composables, every child of the layout needs to be offset by the same amount for the layout to look natural.

When using modifiers, it is important to understand in what order they are applied. Consider a view that has the following modifiers:

```
@Composable
fun Element(modifier: Modifier = Modifier) {
    Column {
        Column(
            modifier
                .padding(10.dp)
```

```
                .border(1.dp, Color.Blue)
                .size(100.dp)
        ) {}
    }
}
```

This is the result that will be computed and displayed by Compose in the user's screen:

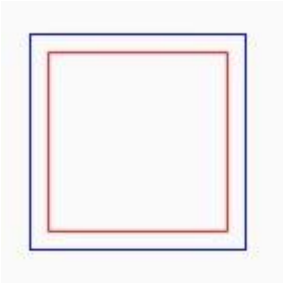


The .padding(10.dp) command adds padding to all directions of size 10 dpi. The .size(100.dp) modifier ensures that the composable has a width and height of 100 dpi. The .border(1.dp, Color.Blue) command creates a border at the outside of the composable that is 1 dpi thick using the color blue.

Consider also another composable with the following modifiers:

```
@Composable
fun Element(modifier: Modifier = Modifier) {
    Column {
        Column(
            modifier
                .padding(10.dp)
                .border(1.dp, Color.Blue)
                .padding(10.dp)
                .border(1.dp, Color.Red)
                .size(100.dp)
        ) {}
    }
}
```

This is the result that will be computed and displayed by Compose:



At a first glance, it seems that the composable should have the red border on the outside and the blue border on the inside. However, this isn't the case. A question that may arise is: Does this happen because the modifiers are applied backwards, from the modifier at the bottom towards the modifier at the top?

The answer to the above question is that typically, modifiers are applied from top to bottom, with a few exceptions that will be explained in a later paragraph. A common misconception it that when the padding modifier of 10 dpi in all directions is applied, the composable will be enlarged by 10 dpi in all directions when this is not how this modifier functions. In reality, the padding modifier shrinks the composable, instead of enlarging it. In the above case, when the padding modifier is applied, the Element composable is shrinked by 10 dpi in all directions. More specifically, this is the order in which the padding and border operations are applied:

1. The composable shrinks by 10 dpi in all directions.

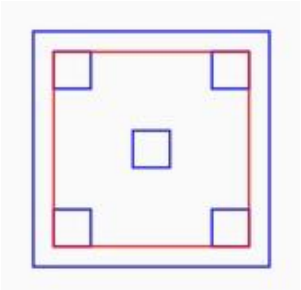
2. The blue border is added around the shrinked composable. This border will no longer be moved by further operations.
3. The composable shrinks again (in all cardinal directions).
4. The red border is created around the composable, which has now been made smaller in total by 20 dpi in all directions.

The above order of operations explains why the blue border is placed around the red border. However, there are two more important matters to take into consideration when applying modifiers, which will be explained below.

As mentioned above, when the padding modifier is applied to any UI element, the UI element becomes smaller. How does this affect UI elements that are placed in it? The answer to this question is that anything else that is placed in the composable will be placed on the shrinked area and will have to shrink as well to fit the smaller composable. In addition, any elements placed will have to be moved inwards to be inside the smaller area. Let's perform changes to the above example and add boxes to all its corners, before shrinking the composable. Instead of a column, we will use a box layout, which allows elements to be placed in every one of its corners, as well as its center, making this layout perfect for this example:

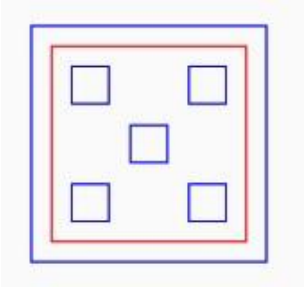
```
@Composable
fun Element(modifier: Modifier = Modifier) {
    Column {
        Box(
            modifier
                .padding(10.dp)
                .border(1.dp, Color.Blue)
                .padding(10.dp)
                .border(1.dp, Color.Red)
                .size(100.dp)
        ) {
            Box(
                modifier = Modifier
                    .align(Alignment.TopStart)
                    .border(1.dp, Color.Blue)
                    .size(20.dp)
            )
            Box(
                modifier = Modifier
                    .align(Alignment.TopEnd)
                    .border(1.dp, Color.Blue)
                    .size(20.dp)
            )
            Box(
                modifier = Modifier
                    .align(Alignment.BottomStart)
                    .border(1.dp, Color.Blue)
                    .size(20.dp)
            )
            Box(
                modifier = Modifier
                    .align(Alignment.BottomEnd)
                    .border(1.dp, Color.Blue)
                    .size(20.dp)
            )
            Box(
                modifier = Modifier
                    .align(Alignment.Center)
                    .border(1.dp, Color.Blue)
                    .size(20.dp)
            )
        }
```

```
    }
}
```



After placing the padding around the squares, the composable will look like this:

```
Box(
    modifier
        .padding(10.dp)
        .border(1.dp, Color.Blue)
        .padding(10.dp)
        .border(1.dp, Color.Red)
        .size(100.dp)
        .padding(10.dp)
) {...}
```



As can be seen above, the padding was placed after the size modifier. This was necessary for the padding to work as was described in the above paragraph. This is an example of a principle of vital importance that should be remembered when applying any modifiers in *Jetpack Compose*: Any modifiers that are applied are not altered by any modifiers that are applied after them. Let's consider the above placing of modifiers. The size modifier is placed after two padding and border modifiers. In a more traditional UI system, when the size modifier was applied, the entire composable might have changed its size to meet the specifications provided by the programmer. In the case above, the entire composable, along with its padding, would shrink until it was 100 dpi horizontally and vertically. In *Compose*, the size modifier doesn't alter the border and padding modifiers placed before and the composable's padding and borders will not shrink.

In a typical UI system, the area within the blue border, along with the 10 dpi padding around it would be resized to 100 dpi. In Jetpack, since the size modifier is placed after the padding and border modifiers, the area inside the red border, which is what exists after the modifiers have been placed will be resized to 100 dpi. The entire composable will have a horizontal size of 10 dpi (left padding) + 1 dpi (left blue border) + 10 dpi (left padding) + 1 dpi (left red border) + 100 dpi + 1 dpi (right red border) + 10 dpi (right padding) + 1 dpi (right blue border) + 10 dpi (right padding) = 144 dpi, same as its vertical height. The padding modifier that is placed after the size modifier will not alter the size of the composable, but the padding and border modifiers placed before the size modifier will. In other words, modifiers are applied from earlier to later and modifiers that have been applied first will not be changed b modifiers that are placed after.

If the padding was placed before, the entire composable would have been larger horizontally and vertically by 20 dpi:

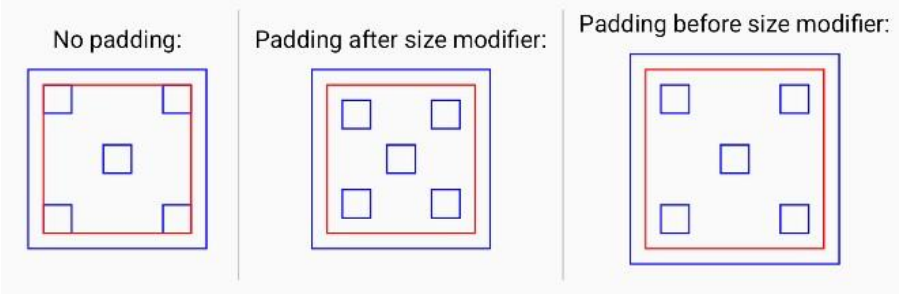


Figure 20: How the placement of the padding modifier causes a different composable to be created

App Class Structure

Introduction to the application

The project is split into a mobile application and an ASP .NET web application which acts as a companion of the mobile application and handles any functionality that is relevant to multiplayer. The web application also keeps track of time remaining in games and stores the results of previous games. The mobile application is responsible for sending the appropriate requests to the server to handle logging in, registering, viewing player and world statistics, showing complete information about other players and quizzes as well as taking all the necessary steps to keep players in a single party and keep them synchronized. The specifics of how the synchronization happens will be explained in a later paragraph.

The classes in this application have been developed with a focus on their interface. Any subclasses also keep the same interface as their parent classes in order for the application to be compliant with good programming practices and make further improvements easy to implement (Gamma, et al., 1994).

The mobile application consists of classes with the following responsibilities:

- Communicating with the web API
- Synchronizing players with the API
- Defining quizzes and any mechanisms that are used
- Analyzing time durations and converting units of time

Communication with the web API provided a significant challenge. Any messages sent to and from the server needed to be understood by both parties. Both the server and the mobile application needed to contain the same classes to describe requests and responses. The mobile application's classes were written in *Kotlin*, and the server's classes were written in *C#*. Many endpoints needed to be used. It was also important for this system to be easy to use, to simplify communication with the game server, when communication of a mobile application with a game server can be considered an intrinsically complex task.

A central mechanism needed to be developed that would perform these requests. This mechanism was described by a class called *APICommunicator*. The *APICommunicator* needed to create many different types of requests, use many different endpoints and translate responses into many different data types. All these actions shouldn't be taken in the same class, to prevent one class from becoming too complicated. This is why this class was split into three parts: The *EndpointRetriever* class had the duty of allowing easy access to any endpoints that would be used, the *RequestPreparer* was responsible for creating the necessary requests that were going to be sent to the server and the *APICommunicator* class contained the general logic, created the necessary requests, sent them to the server, retrieved the server response and translated the server response into the correct data type. This separation allowed for the request preparer to create many types of requests using the same endpoint and for the API communicator to handle translation of server responses and to choose which request it required to perform a certain communication action.

An instance of the *APICommunicator* class contains an instance of the *RequestPreparer* class, which in turn contains an instance of *EndpointRetriever*. As mentioned above, the separation of these classes makes it possible for the API communicator to use many different requests and for the request preparer to use the same endpoint for many requests. The practice of not copying code between different

classes and instead, storing it in a central location, accessible by all of the classes that need it is called *DRY* (Don't Repeat Yourself). The usage of instances, instead of classes with static interfaces or singletons to perform all these requests, allows for subclasses of every one of the three above classes to be made and easily implemented into the code and for the usage of code completion through the IDE when writing new code for these classes. This programming principle also creates a logical connection between all three classes that can be easily understood by other programmers that may wish to analyze the code.

The API communicator was the main entry point for any communication with the server. This is similar to the implementation of the *Facade* design pattern, where an entry point with a simple interface is used to hide the usage of a more complicated program that it sends messages to in the background. In this case, the API communicator is the *Facade* and the contained classes constitute the complicated server communication mechanism (Gamma, et al., 1994).

Synchronizing players with the API

For any single player game, a player does not need to be synchronized with a server, because the game itself can maintain its own state and understand when the game is opened, closed or resumed. A single-player game is usually capable keeping players on its intended path by limiting players' actions and preventing them from escaping the game's rules. Multiplayer games provide unique challenges that may not be noted at first glance. For instance, if one player's phone shows something different than the other player's phone when they should be showing the same data, does one or both players displays show incorrect data? During a game session, should the decisions of players with incorrect data on their game be counted, or should they be discarded? What causes a player to possess incorrect data about a game session? These are only a few of the problems that are met when creating a multiplayer game from scratch.

Typically, in multiplayer games, if players wish to play together, they first connect to the same game room, which is called a "party" or "lobby". Depending on the type of connection (i.e. peer-to-peer connection) between players, one of the players might be considered the party "leader" or "host". There are three basic connection types that are used in games today:

1. Listen Server with Host Migration: All players are connected to each other and there is no connection to a game server for synchronization reasons, but there can be communication with a game server to send results, statistical data etc. so that it can be stored (Mildenberger, 2024). One of the members in the party is considered to be the party leader. The party leader that exists has the added responsibility of acting as a game server and synchronizing the rest of the players with the actual game state. If the party leader leaves, the game has to choose a new player to become the leader. This is called host migration and some games have managed to achieve "hidden" host migration, where the host can disconnect, but if everything works well, the players may continue playing the game until a new leader is chosen and not realize anything is wrong.
2. Peer-to-peer connection: Similarly to the above connection type, all players are connected to each other. There is no party leader and every player communicates with every other player. All of the players send data to each other and any data sent by one player is considered just as important as the data sent by any other player. This type of connection causes slowdowns when players join or leave the game session, as all players are considered as important as leaders and leaving or joining changes the communication dynamic of the players (Mildenberger, 2024).
3. Dedicated server: The players all connect to an external game server and send requests to it for synchronization. The server takes complete responsibility for responding to players quickly, handling many players and game sessions simultaneously and sending correct data to the players, so that the gaming experience will be as smooth as possible. Any game servers are owned by the game creators. This connection type is highly successful, if the servers are quick at responding to requests and keeping players connected. This method and has been getting increasingly used recently.

Generally, players communicate with each other, the party leader or an external server and periodically ask for the current state of the game session. Players in a game session are usually synchronized with the usage of the UDP (User Datagram Protocol) because of its ability to send and receive data very quickly, due to the lack of verification of data sent and received of this protocol. TCP requests are usually only made to transfer game-critical information (statistics, results, rewards and other data that should not be lost).

When players coexist in a game session, their devices contain what they believe to be the current session of the game state (i.e. current question being asked in a quiz game and time remaining). This means that for *n* players in a lobby, there are typically *n* game states that are loaded. This means that there are *n* game states loaded in the clients and one more game state that can be considered as the real state, or the actual game state that the other players need to keep informed about. So, in total, there are *n + 1* game states. In cases where there is a party leader, the game state shown in the party leader's device is considered to be the same as the actual game state and the leader sends this state to the other players to keep them informed. In the case of *peer-to-peer* connections, it can be assumed that apart from all the client game states, there exists one game state that is real and needs to be replicated on every client, even if it is not in any of the player's devices. In the case of dedicated servers, the servers contain the actual game state and they keep all of the players updated. In all cases, whichever device is responsible for keeping players updated of the current game state regularly sends messages to make sure that all players are kept informed.

One of the design issues that had to be solved was that apart from the type of connection, there can be two types of messages that can be sent to players for synchronization reasons. More specifically, whenever the server or party leader is to send a message to the players to synchronize them, they can be either complete or incomplete. Complete messages contain all the game session state. Players can load the values of these messages to immediately update their client game state. Incomplete messages only contain changes that have happened since the last update that was sent. These messages are smaller in size, but they do not describe the entire game session.

Two different types of protocols can be used for communication between players, or between players and the game server. The *UDP* does not verify that messages have been sent or received. The messages simply leave the sender, after which they will hopefully reach the receiver intact. Messages can be lost or corrupted along the way to their destination. The sender cannot know whether the messages arrived and the receiver cannot be certain that someone is trying to send a message to them. This protocol is used for streaming purposes, or for any time-sensitive applications.

The *TCP* (Transmission Control Protocol) is far more reliable than the *UDP*. Both the sender and receiver are aware that a message is being transferred and it is considered certain that the message will reach its destination. Senders typically send *ACK* messages, which imply that a packet was sent successfully. If a message is not successfully sent, receiver will request the specific message again. Receivers recognize server congestion by the number of *ACK* messages that are sent, or if there are no *ACK* messages sent at all and classify the server congestion as either light or heavy. If there is server congestion, the sender will stop sending messages for a small duration. This protocol is much more reliable for message sending, but the rate of messages sent through this protocol is unstable and changes quickly and drastically in a short period of time.

Multiplayer games of any kind can theoretically use any of the two above message types, as well as either the *UDP* or *TCP* protocol to have these messages sent to players. This creates four possible combinations of the protocol and the message type that can be used to enable communication between the client and the server. Consider a 2D multiplayer game, in which two players are in a game session. These two players have a specific position and rotation:

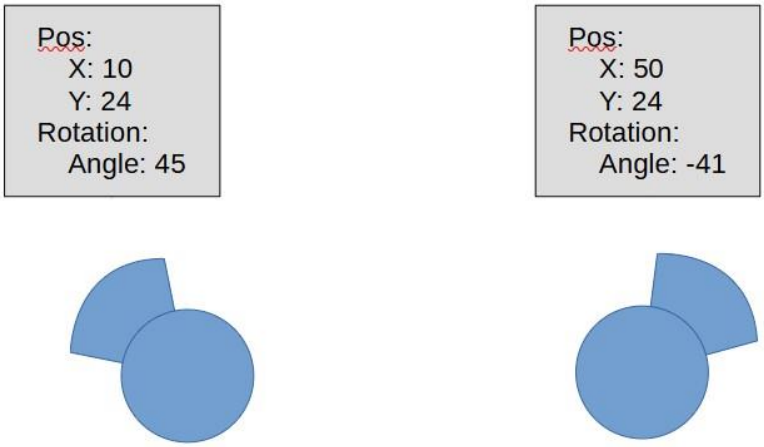


Figure 21: Synchronized players in game session. Both contain their own position and rotation. Graph created with the application LibreOffice Draw.

In this theoretical game, the two players have a specific position and rotation, as mentioned previously. Both players should see the entire game space and how the players are located, similar to the above image. In case of desynchronization, the devices of the two players will show different data.

The four possible combinations between message types and protocols will be explained with an example game session:

- TCP with incomplete messages: In this case, players are synchronized with the use of messages that only contain the changes that have occurred since the last update. The usage of TCP ensures that messages will most likely be received by both players. After loading the changes, the player devices will show the current state.

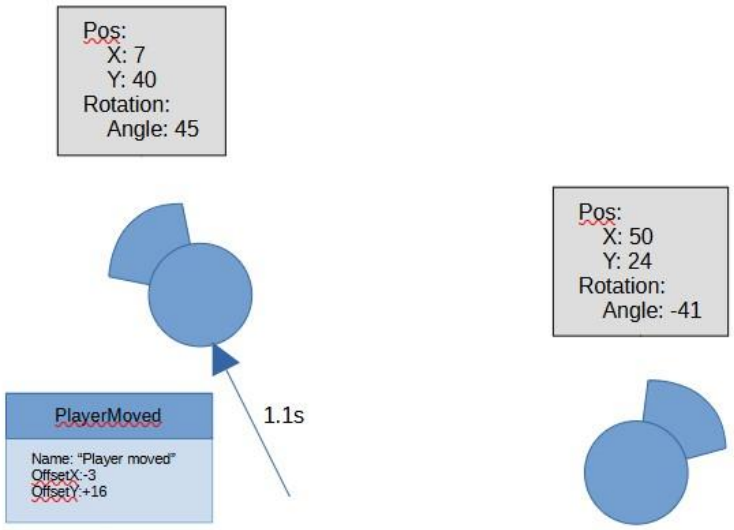


Figure 22: A graph describing two players in a 2D game, in which they can move and rotate. The graph is shown from the perspective of one of the players. The arrow represents a movement made, as sent by the server

In the above diagram, the arrow indicates that a message has been received that caused the left player to move. The text next to the arrow indicates the time that has passed for the message to be received from the viewpoint of one of the players and the rectangle next to the arrow describes the messages that has been received. In this case, the message is of type "PlayerMoved" and it contains the difference since the last updates in x and y coordinates, called "OffsetX" and "OffsetY" respectively. This type of design will be used also be used in later parts of this chapter. In some figures, only the message types will be shown and not the specifics of the sent message due to spatial constraints. The following image demonstrates the data that both players will contain after receiving messages with this protocol and message type:

According to the image above, both players are showing the same data. However, the usage of TCP means that all players will need to receive the messages, before more updates can be given. This means that communication between players will be as slow as the connection speed of the player with the highest network latency.

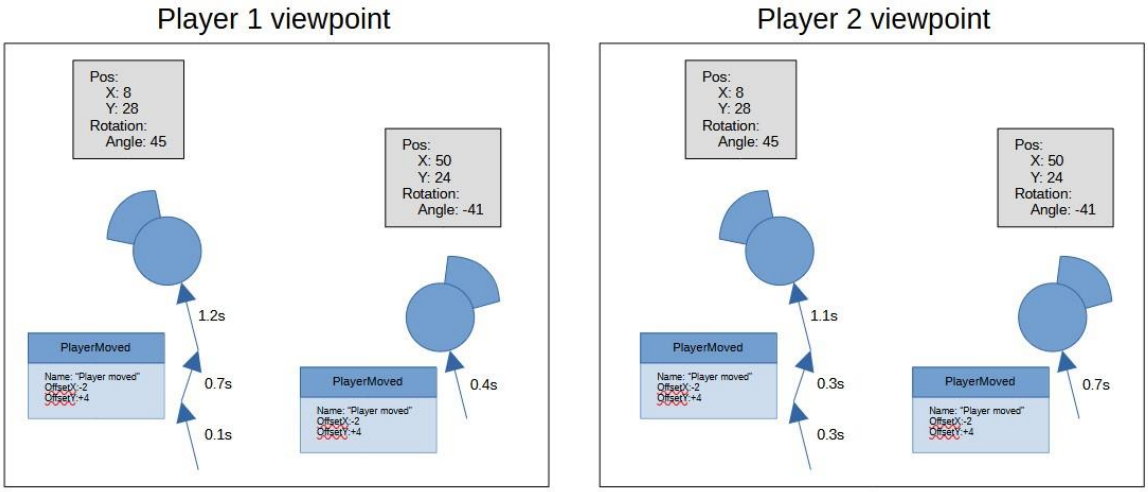


Figure 23: Both players' view points in the case of a game using the TCP, with incomplete messages

- TCP with complete messages: In this case, the Transmission Control Protocol provides the same reliability when transferring messages. Similarly to the case above, all players will receive all the sent messages and be synchronized. Any messages sent will contain the entirety of the game state. This means that all messages will be of the same type and receivers don't have to differentiate between different message types. However, the messages will be extremely large, making message transfer even slower than the smaller message transfer using TCP in the case above.

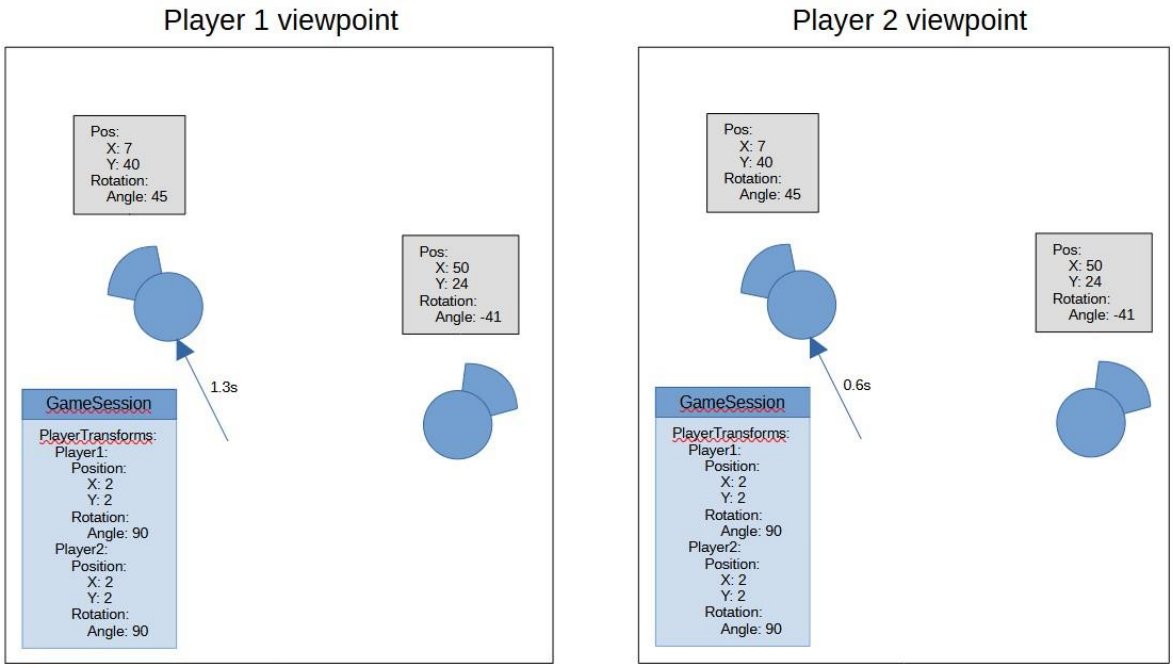


Figure 24: Message corruption that causes the two players to view a completely different state of the game. Notice how the positions and rotations of the left player are not the same for both players devices.

- *UDP* with incomplete messages: In this case, messages are transferred much more quickly due to the usage of the *UDP*. The lack of any kind of transmission control, makes it much more likely for errors to occur, but makes message sending much faster. In fact, *UDP* allows sending to be extremely fast, fast enough to update all players 60 times per second, allowing for games to update their game state very rapidly and for players to be more immersed in the game's world, as long as all of the messages sent reach their destination. If any messages don't reach their destination though, massive problems can occur. For instance, if one player doesn't receive a specific message, any next messages that are received can completely desynchronize the players from the rest in the session. When incomplete messages are used, it is imperative that they all reach any target players. Desynchronization happens because players may not receive state info and any state info that is received is based on the info that was sent before it.

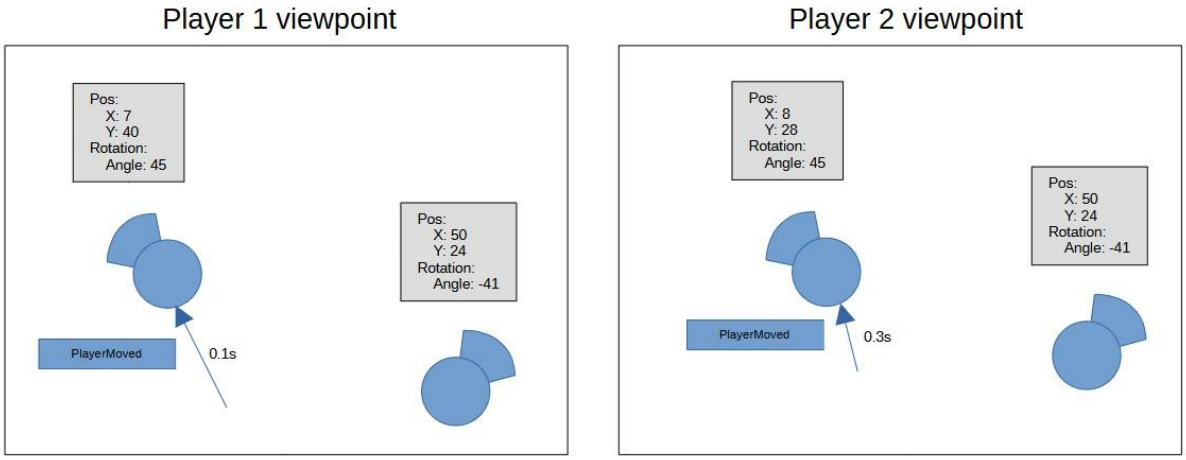


Figure 25: Message corruption that causes the two players to view a completely different state of the game. Notice how the positions and rotations of the left player are not the same for both players devices.

- UDP with complete messages: The use of complete messages allows players to be resynchronized, even if messages get corrupted, or fail to be sent or received. Any messages not received provide a small duration of desynchronization, but when new messages arrive, players will quickly obtain the real state of the game session again. This method is favored by games, in which all players can make decisions quickly (i.e. 60 times per second) so that messages can be sent to any servers or players and inform them of the latest actions.

A sequence diagram that explains how a server can send and receive data from players is shown below:

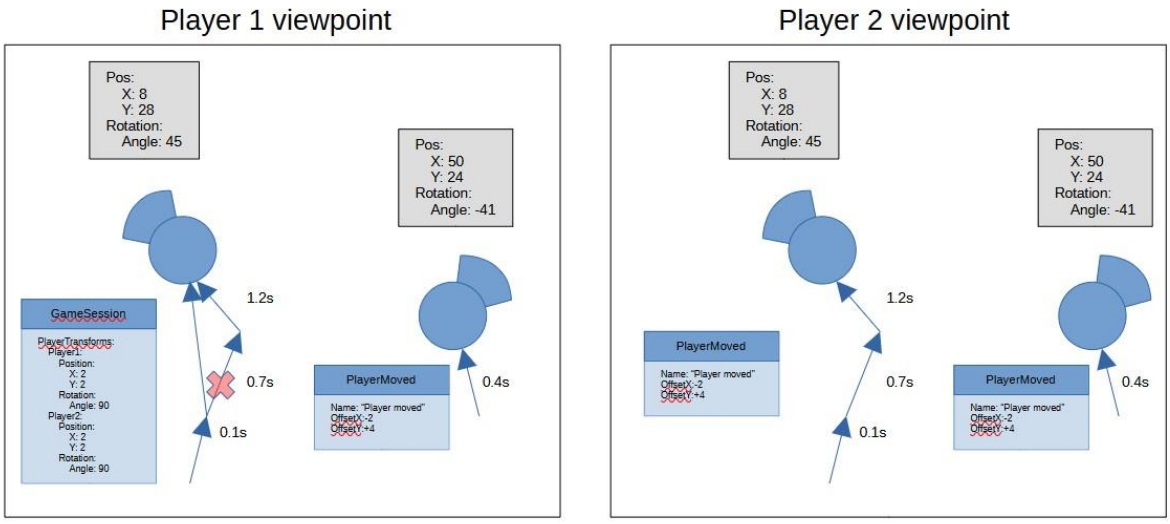


Figure 26: Both players' view points when using the UDP with complete messages. Note how in this example, the data is corrupted for player 1, but player 1 is resynchronized with the next data packet that is received to their device

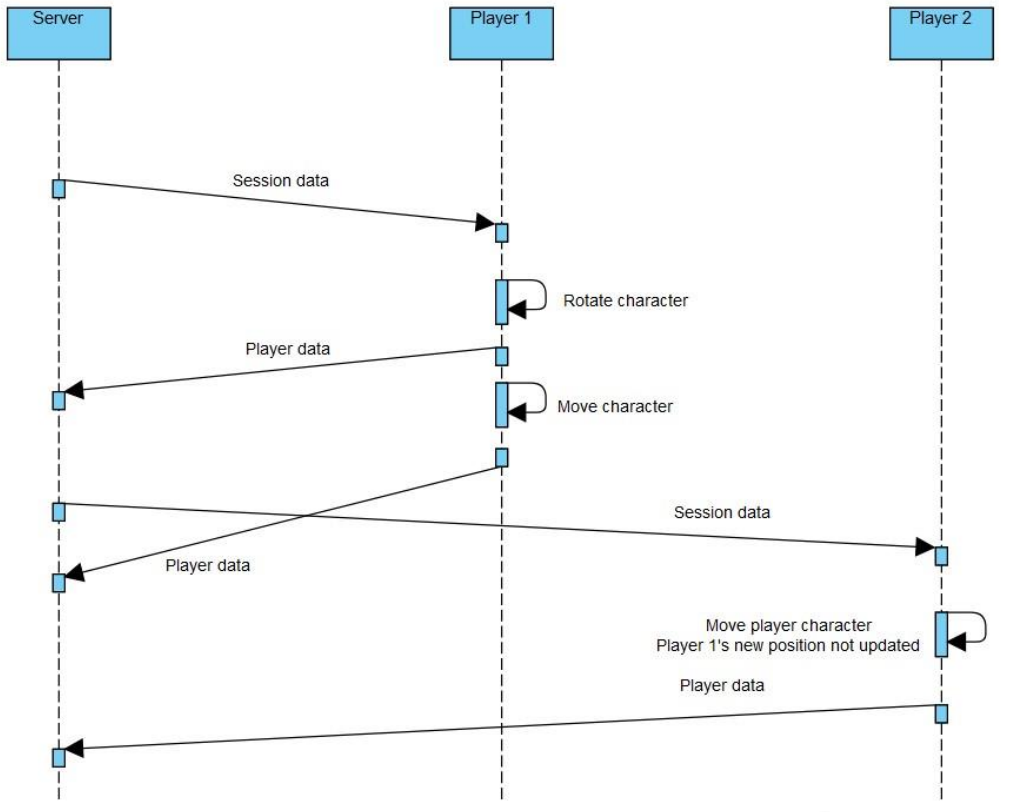


Figure 27: Sequence diagram that described player communication with a game server

In the context of the quiz game in its mobile form, synchronization went through multiple iterations. The class that was used to perform all the necessary actions for quizzes to progress normally was called *QuizPlayer* and it was expanded and improved, until the one that is currently in use was created. The first iteration didn't synchronize with the server at all and provided a single-player experience. This iteration was mostly just a test to see what functions would need to be performed to make the quiz playable. Afterwards, a new iteration was created, called *SyncingQuizPlayer* and slowly perfected, until it was used in the final version of this application. Multiple and complicated techniques had to be used to achieve synchronization between the players in the game and the server, one of these being using multiple threads (one for syncing, one for calculating the timer's progression) and preventing them from running simultaneously and creating deadlocks. Each of the two threads that was used ran its own suspend function to perform its dedicated work. In addition, updates to the timer given by the server were given less priority than updates given by the clients themselves. When the mobile application received an update, it waited until the client finished its work, before updating the current timer and question with the updated values. This helped prevent deadlocks, as well as the client's timer receiving changes from the server and immediately discarding them in favor of its own state.

When players joined the same party, they had to be able to see each other, as well as view basic information about all of the other players in the party. More specifically, they had to be directed to a menu in which they could see the rest of the players in a column layout. For every player, the username and picture needed to be shown, so that a group of players could make sure that the players they are looking for have joined the same party and are about to play the same quiz. Similarly to the approach that was followed for synchronizing a party of players in a quiz that is in progress, a mechanism that imitates real-time updating is used. This is achieved by every player regularly sending requests to the server and waiting for a response. When something important happened, the server would respond and give the current state of the party, including the players in it, the game about to be played and the current time remaining before the game starts. Functions that were used for handling server events were called hooks, due to the similarity of these functions with hook functions of the *Template method* design pattern.

Defining quiz classes, as well as questions and answers:

For the purpose of the development of the quiz game, an important aspect of its function is its definition of quizzes, questions and answers, correct and wrong answers, how questions and answers are compared and graded and how questions are displayed on screen. The challenge that is presented here is that there are multiple types of questions and answers. According to the design principles that were decided on prior to development, there are also many types of UI *composables* that should be able to display the questions and answers. This creates the problem of communication between three different superclasses and all their subclasses. The program's classes should be structured similarly to the diagram below:

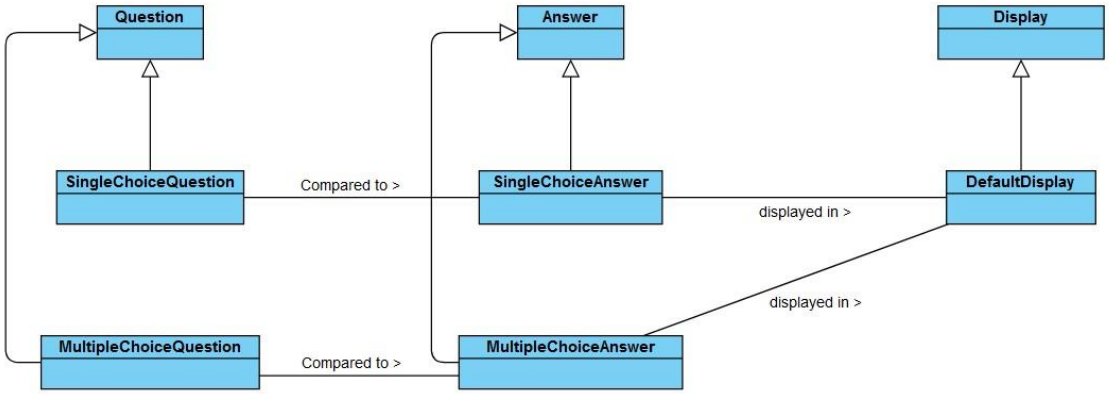


Figure 28: Class diagram describing how the quiz game's questions and answers should communicate

According to the above diagram, questions of any kind should be able to be compared to answers of the same kind. Also, questions and answers should be displayed in a part of the application's *User Interface* (UI). This was especially difficult to develop, because of the existence of three superclasses, the subclasses of which should communicate without incident.

Communication between two classes with different interfaces is a problem that has already been solved. The solution is explained in a design pattern, called *Visitor*. The *Visitor* pattern was created, due to the necessity of having classes analyzed by other classes with a different interface. However, the *Visitor* pattern can also be used to facilitate communication between subclasses of any two superclasses. For the above example, it can be used to allow calculating the grade of an answer for a specific question.

The communication between three superclasses is not as simple as the communication between two. In the book "Design patterns: Elements of Reusable Object-Oriented Programming", only the *visitor* pattern for two superclasses is explained. The same solution cannot be applied for three. This means that a different version of the *visitor* pattern had to be used. The question class had to communicate with the answer class and both of these classes needed to communicate with the display class to present questions and answers of all types on screen. How could the display class know the types of questions and answers that are given to it? Whenever communication between these three classes was requested the following order of operations was used to fix the above problem:

1. The question begins the process.
2. The question sends itself to the answer.
3. The answer analyzes the question and figures out the question's specific type.
4. The answer already knows its type, so it sends itself and the question to the display class .
5. The display class now knows the specific types of the question and answer given to it and, according to its own type, creates the necessary composable to display them.

The display system will have to use the interfaces of the questions and answers to display the relevant data. This means that questions, answers of all kinds will need to have an interface that can be used for displaying their information. This is the reason why specific methods were added to the interfaces of questions and answers that were responsible for displaying them in the UI.

The above system was copied to the final version of the application with the only difference being that there ended up being only one question type that needed to be created for the purposes of the

project. The existence of only one type does not provide any detriment for the creation of more question and answer types, because the system is completely able to handle more than one and there can be more than one display system used in the application.

The final version of the class diagram used is the following:

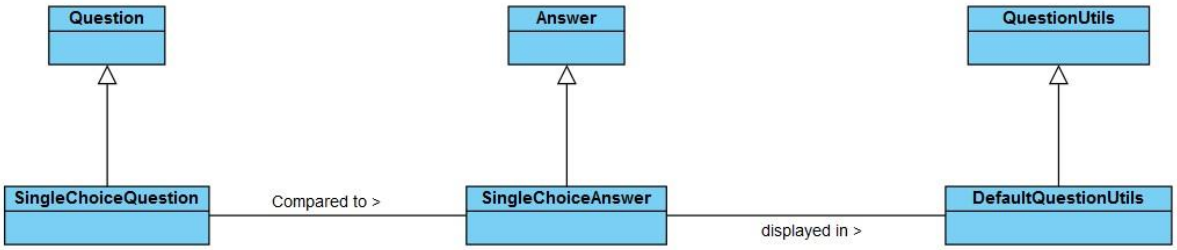


Figure 29: The final class diagram of the quiz game

Answering given questions was another challenge that needed to be overcome, due to the existence of multiple possible question and answer types. The *Design Patterns* book states that programmers should only the highest classes that are in the class hierarchy to achieve their goals, whenever possible. Subclasses should almost never be used, as this can diminish the required separation between the application's classes / concerns. The interface that exists for questions and answers must be able to allow communication between questions and answers themselves and not their subclasses. This means that questions should not care about what answer has been given, and answers given should not care about what type of question they were given to. This creates a problem when answers and questions need to be compared and a grade has to be given. More specifically, every question should be able to be compared with any answer for grading and every answer should be able to be compared with any type of question.

The interface that was designed for the question class had the following method:

1. getText(): Retrieves the question in text form.
2. getMaxGrade(): Retrieves the maximum grade that this question can produce.
3. getTime(): Gets the time that this question will be displayed and answerable for.
4. accept(answer): Starts the visitation process to get the answer grade.

The interface that was designed for the answers had the following form:

1. getGrade(singleChoiceQuestion): Analyzes the question sent and produces the grade. For every question type that is created, a new *getGrade* method will also need to be developed to handle the new type of question.
2. copy(): Creates a copy of the answer.
3. notifyAPI(playerId, currentQuestionIndex): sends the results of the answer and the current question that this was the answer to to the web server, so that they can be stored.

The answers are responsible for providing a grade when compared with a question. This is because answers have a small amount of responsibilities, so the grading responsibility was placed on them.

In the answer interface, the getGrade method can accept all kinds of questions and provide a grade. The UI also is also able to function normally without knowing the exact type of question or answer given to it. This means that some methods had to be developed that allowed for this functionality. The methods that were added to the interfaces of the classes are the following:

For the *Question class*:

1. PresentedTitle(presenter)
2. PresentedAnswers(answer, questionUtils, locked, onAnswerChanged, modifier)
3. createEmptyMatchingAnswer(): Creates an empty answer of the matching type to this question. This is used to display the possible answers in the *User Interface*, when the question is first loaded and has not been answered yet.

For the *Answer class*:

1. PresentedWith(question, questionUtils, locked, onAnswerChanged, modifier)

The *PresentedTitle* method is responsible for showing the question itself to the users, but not the possible answers. The *PresentedAnswers* method displays all of the possible answers for the same question. This method is able to display all types of possible answers. The reason that the two methods are not unified into one method is because every question is considered to have a title, but questions are not required to have the same answer types. The answer class is the middle man of the communication with the UI and transfers all the information to the *QuestionUtils* class, which will display the information to the user's screen.

Apart from the above methods, it was also important to allow the questions and answers to be displayed on screen. Multiple display subclasses allow for multiple was for questions and their respective answers to be displayed. As mentioned above, this required a system in which the subclasses of three superclasses would communicate. In this application, the superclass that displays the necessary information to the user through the user interface is called *QuestionUtils*. This class is responsible for displaying all kinds of questions and their matching answers.

```
interface Question {
    fun getText(): String
    fun getMaxGrade(): Int
    fun getTime(): Long
    fun accept(answer: Answer): Int
    @Composable
    fun PresentedTitle(presenter: QuestionPresenter)
    @Composable
    fun PresentedAnswers(answer: Answer, questionUtils: QuestionUtils,
locked: Boolean, onAnswerChanged: (Answer) -> Unit, modifier:
Modifier)
    fun createEmptyMatchingAnswer(): Answer
}
```

```
interface Answer {
    fun getGrade(question: SingleChoiceQuestion): Int
    @Composable
    fun PresentedWith(question: SingleChoiceQuestion, questionUtils:
QuestionUtils, locked: Boolean, onAnswerChanged: (Answer) -> Unit,
modifier: Modifier)
    fun copy(): Answer
    suspend fun notifyAPI(playerId: Int, currentQuestionIndex: Int)
}
```

```
class DefaultQuestionUtils: QuestionUtils {

    @Composable
    override fun PresentedSingleChoiceAnswers(
        question: SingleChoiceQuestion,
        answer: SingleChoiceAnswer,
        locked: Boolean,
        onAnswerChanged: (Answer) -> Unit,
        modifier: Modifier
    ) {
        AnswerResources.SingleChoiceAnswers(
            question = question,
            chosenIndex = answer.chosenIndex,
            onChosenIndexChanged = {
                answer.chosenIndex = it
                onAnswerChanged(answer.copy())
            },
            locked = locked,
            modifier = modifier
        )
```

```
        }
}
```

The questions that are currently developed in the quiz game have a single correct answer and so the max grade is 1. This means that a user answers the question correctly, they get a grade of 1 and if they have a wrong answer, the grade they will get is 0. This is a good way to differentiate between right and wrong answers, but how will this grading system work in case new question and answer types get added? New types of questions could have partially correct answers, which also had to be supported by the application's code base. In order to add support for all question and answer types, every question is able to have a different max grade, so that answers can be completely wrong, partially correct or completely correct. A convention that was made is that the grade that are given after answering any type of question are always numbers, never text or other, more complicated data types to allow the server to analyze and compare how correct answers are easily. Finally, the entire grading system was developed to only use integers and not decimal numbers, to avoid floating point errors.

The grading processes are developed in such a way that the highest classes in the hierarchy are used without knowing which questions, answers or question displaying utilities specifically are used and any type of question and answer can be later added to the quiz game. This also had to be developed in the context that the grading system will not use decimal numbers. For instance, a possible answer type that could be developed could have many correct answers and a user will need to select all of them to get the max grade. Due to the removal of floating-point numbers, the questions themselves needed to have a different max grade, so that partially correct answers are supported without using fractions and decimal numbers.

Another problem that needed to be answered is the following: What happens if non-matching answers are given to questions? How should the interface be created so that it can be effective at communication between non-matching questions and answers? The solution to this problem was a specific piece of computer logic that was placed in the *getGrade()* method. Whenever a question is sent to an answer for analysis and grading, in accordance with the *Visitor* pattern, the question type is checked and if it does not match the answer, the grade given is equal to zero. This means that the answer is completely wrong and should not be graded.

Analyzing and converting units of time

When developing the quiz game, it became clear that handling units of time was necessary for displaying questions and synchronizing players. Displaying question timers in-game and handling synchronization of question timers with the server also required specific *Kotlin* classes and techniques. Lambda functions were used extensively in the creation of timers, both timers for the game and timers that perform functions in the background.

Lambda functions are functions that programmers use to give specific implementations to solve abstract problems. Abstract problems require different solutions to be given for every specific time the problem is met. Programmers give their own solutions to abstract problems in the scope of their own application as *lambda* functions. Their ability to give a specific solution to a general problem is why they are called higher-order functions (Smyth, 2023).

Kotlin provides a way to provide specific variables, objects and functions to lambda functions, so that the creators of the *lambda* functions will have access to them. These may be given for convenience's sake, to transfer important information to the programmer that creates the lambda function, or to allow the lambda designer to use specific functions that perform difficult operations.

In the context of the mobile application, timers were used extensively to mark the remaining time a question would be displayed for and to perform operations at regular intervals (i.e. requesting the time duration that a question has been shown for from the server). Any timers needed to be reusable and able to solve a large amount of different problems. In other words, they needed to be flexible enough to be placed in many different places in the code. They also needed to properly handle all events that may occur. Because *suspend* functions could be used in the event handlers, any event handlers were also *suspend* functions. Finally, the time itself needed to be editable from within the event handler functions, in order to change the time remaining if specific events occurred. The resulting class that was created was the following:

```
class NanoTimerObserver(
    val onTimerStart: suspend NanoTimeConverter.(timeNano: Long) ->
Unit,
    val onTimeChange: suspend NanoTimeConverter.(timeNano: Long) ->
Unit,
    val onTimerEnd: suspend () -> Unit
) {}
```

The above class contains the *lambda* functions *onTimerStart* and *onTimerChange*. These lambda functions provide their own scope, which allows easier analysis and manipulation of units of time. These are the actions currently supported by the scope:

- nanoToSeconds: Useful for converting nanoseconds to seconds.
- secondsToNano: Used for the opposite of the above operation.

The code that was used in the creation of this scope is shown below:

```
class NanoTimeConverter {
    val nanoPerSecond = 1_000_000_000
    fun nanoToSeconds(nano: Long) =
        nano / nanoPerSecond

    fun secondsToNano(seconds: Long) =
        seconds * nanoPerSecond
}
```

These classes needed to be used in the context of a timer, as previously explained. Due to the difficulty of creating a high-precision timer in *Jetpack Compose*, multiple different implementations needed to be developed and tested. For this reason, not only the timer observers, but also the timers themselves needed to be interchangeable.

Timer interchangeability was achieved by using an interface that contained all the definitions for timers that needed to be tested as methods with default implementation. After any testing, some implementations for timers could be added, altered or deleted in this interface. In the timer implementations that ended up being used the most, a nano timer observer was added as a parameter. This way, not only was the timer able to be replaced, but the handling of timer events could be completely separated from the timer itself, allowing for the same timer to be used for completely different purposes.

Web server implemetation

The server was developed as a secondary worker to the mobile application. Its purpose was to connect to the mobile application and provide a permanent storage, as well as allow players to connect to each other, form parties and play the game together. The multiplayer functionality is only possible with the usage of this server. The game server provides a set of ways that players can communicate with it, while stopping players from having more control than necessary over the game server and the data and current statues of other players.

When players are considered to exist together in a part of a game, both players' data should be synchronized. This is important as it prevents players from giving instructions to the game based on wrong information and in the context of a quiz game, it allows both players to know when the current question that is being answered has changed, or when the game has ended. When players request the current game session status from the server, any players that have a higher distance to the game server will take longer to receive the responses, meaning that they will be slightly less synchronized with the actual game data. Temporary failures connecting to the game server also create problems when synchronizing.

When creating the server, several changes needed to be made to the Program.cs file, which is the file that starts the web server and sets some of its configuration options. For instance, cycles needed to be removed, to allow for complex connections between models in the game server.

In general, the controllers contained in the server were responsible for:

- Uploading players' profile images
- Starting games and remaining updated on their progression
- Starting, destroying, joining and leaving player parties

- Staying updated with other party members in the menu
- Creating and deleting new players
- Retrieving world statistics
- Retrieving statistics for each player
- Creating new questions and quizzes

The above controllers contained methods/endpoints that allowed remote users to communicate with the server and read, update, create or delete data with highly specific points of communication. For instance, players can join another party by using the *LobbyMigrationController*. This controller contains the following methods:

- joinLobby(playerId, lobbyId): Makes player leave their current lobby and join another one. The player and the lobby ids need to be provided to the server. If the player is already in the party, an error code is returned.
- leaveLobby(playerId): Makes the player leave their current party. This method does not allow a player to leave a party when they are the only member, returning an error message if this is attempted.

The above methods are the only ones that are possible to be used for manipulating party data. The lack of redundant or futile methods prevents players from using untested mechanisms and making changes that are not intended. More specifically, all the endpoints in a web application are like the interface of a class, which allows programmers to communicate with it. Keeping the underlying mechanisms and stored variables hidden is similar to the encapsulation that occurs in a well-developed class.

One of the biggest challenges that needed to be faced was keeping players updated of a game's progression when players in a party were playing a quiz or waiting in the menu for the game to start. Any changes that affected players needed to be transmitted immediately, so that players could be informed of these changes in real time. Two approaches were considered:

1. Players would send requests to the server at regular intervals to get the current game or menu state: This approach would make it easier for players to synchronize their data with the data of the game server. However, if an event occured between requests sent by players, the players would be unsynchronized until they send the next update request, removing the illusion of real-time updating. Players were not able to perform these requests rapidly either, due to web servers written in *ASP .NET* working with the *TCP* protocol. Because of the above issues, this approach was not used.
2. Players would continually send requests to the server, which would reply with the current game or menu state every few seconds, or sooner if an event occurred. This would allow the server to immediately inform the players if an event occurred and players would feel like they were updated in real time, even with the usage of *TCP* protocols and the slow response time that entails. This approach would be more challenging. Because the server would be responsible for understanding if an event had occurred, it had to be responsible for detecting events, be able to sleep and recheck for events later and it had to use multithreading to keep multiple players informed simultaneously. However, the usage of the delay method instead of the sleep method would drastically reduce the load on the server, due to any threads that were waiting for the time to pass were performing other work (keeping other players updated and responding to their requests). This is the approach that was eventually used.

The server had to be able somehow store user's images and retrieve them from a database. The usage of a cloud server was considered but not performed, because it was considered important to keep as much control over the application's functionality as possible. This was particularly difficult to implement, due to the necessity of retrieving player's images at any point in time, as well as keeping the service that retrieved the images separated from the rest of the code. The solution to this problem ended up being similar to the gatekeeper cloud design pattern:

```
[HttpGet("{imageName}")]
// GET: ImagesController/FindImage/Player1Image.jpg
public async Task<ActionResult> FindImage(string imageName)
{
    var imageUrlInDatabase =
```

```
        webHostEnvironment.WebRootPath +
        "/Images/" +
        imageName;

    string? contentType;
    new FileExtensionContentTypeProvider().
        TryGetContentType(imageUrlInDatabase, out contentType);
    byte[] imageBytes =
        System.IO.File.ReadAllBytes(imageUrlInDatabase);

    if (contentType == null)
    {
        return UnprocessableEntity();
    }

    return File(imageBytes, contentType);
}
```

Another problem that needed to be solved was the models themselves. The complexity of the models in the database made them impractical for sending and receiving through the internet. Smaller models needed to be used, to reduce data sent and decrease the complexity of messages that were sent and received. For this purpose, thin models were created. These models described models in the database in a simpler way and were easier to understand by the programmers and the applications. The usage of thin models reduces the network bandwidth used by the application. Due to conversion of data models being common to almost all models, a separate mechanism was created to convert all kinds of data models to other data models. This mechanism was completely separated from the data models themselves and was called *ModelConverter<K, V>*. This mechanism describes how any model of type *K* can be converted to any kind of model of type *V*. Every new conversion possible is created as a new class than inherits from this one:

```
public abstract class ModelConverter<K, V>
{
    public abstract V getConvertedModel(K model);

    public List<V> getConvertedModel(List<K> models)
    {
        List<V> convertedModelList = new List<V>();
        models.ForEach(model =>
        {
            convertedModelList.Add(getConvertedModel(model));
        });
        return convertedModelList;
    }
}
```

Below is one of the conversion classes that is used by the web server:

```
public class AnswerToThinConverter
{
    public ThinAnswer visitAnswerForConversion(SingleChoiceAnswer answer)
    {
        return new ThinAnswer
```

```
        {
            ChosenIndex = answer.ChosenIndex,
            Type = "Single choice"
        };
    }
}
```

For the real time updating of players, an advanced system needed to be developed in the game server's code. The server had to understand which players were in the menu or in-game and send any events it detected to them. The players only get events from their own game or party menu and not have access to other player's event data. This was necessary to reduce the data sent and to keep players from accessing the private information of players in other parties. It was decided that the web server would keep a record of players and the parties they were in, in memory and notify players in the case of an event. When the players left the party or quit the game, the server would stop sending updates.

Players do not remain in the server's memory longer than they need to. Whenever players make a request to try to hook to the game's server for events, the server sends the game session or menu session state after a few seconds, or sooner, if an event occurs, as mentioned before. After the server responds with the current game or menu state, the players are immediately removed from the server's memory. The controllers are very strict with storing players in memory, in order to prevent the server's memory from filling completely.

The controllers are responsible for storing players in memory, with the usage of static dictionaries. Every controller responsible for transmitting events to players contains a dictionary. This dictionary stores the players, who take the role of observers when they send an event request and are stored in it. After that, they are notified when an event occurs, similarly to the *Observer* pattern. More precisely, the *LobbyMigrationController* and the *GameController* contain the necessary methods for updating players and observes in their state. The classes that check if an event occurred are separate and the controllers communicate with these classes through their interface to detect events and transmit information to the players, before removing the players from their dictionary.

Conclusion

The creation of a quiz game and the connection between the mobile applications and the web server provided many more problems that may be noted at first glance. The solution of these problems allows players to communicate with other players within the confines of the quiz game and enter a game session together, while keeping updated, almost in real time. A great amount of the functionality is made by the author, allowing this application to exist and possibly work for a long time without being rendered unusable by third parties. This quiz game could also be used in an educational context, where students could be playing, answering questions, viewing their grades and learning together. Some additions that could be made to this game are the creation of more robust synchronization systems that are less prone to errors, more meaningful ways for players to communicate and more information about other players in the same party could be shown in the game's menu.

One of this application's weaknesses are its lack of security. Players can theoretically send false information to the server quite easily, or send information on behalf of other players. No identification of the users is done in the functionality of this software. As such, it is an extremely important field that should be studied in any further iterations or new versions of this software. This would prevent hacking attempts and allow players to be certain that they are in control of the data that they send to the web server.

Bibliography

Alepis, E., 2009. *Συναισθηματική ευφυΐα σε αντικειμενοστρεφή πολυτροπικά συστήματα διεπαφής για κινητή και ηλεκτρονική μάθηση.,* s.l.: s.n.

Alepis, E. & Patsakis, C., 2017. *The All Seeing Eye: Web to App Intercommunication for Session Fingerprinting in Android.* s.l., Springer.

Alepis, E., Virvou, M. & Tsihrintzis, G. A., 2010. Object oriented architecture for affective multimodal e-learning interfaces.. *Intelligent Decision Technologies.*

Efthymios, A. & Virvou, M., 2014. *Object-Oriented User Interfaces for Personalized Mobile Learning..* s.l.:Springer.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object=Oriented Software.* s.l.:Pearson Education (US).

Kastanas, G. & Sakkopoulos, E., 2022. *Mobile student information services: A case study in Greek Open University.* s.l., IEEE.

Mildenberger, T., 2024. *Peer-to-Peer vs. Dedicated Server – A Comparison.* [Online]
Available at: https://contabo.com/blog/peer-to-peer-vs-dedicated-servers/
[Accessed 15 7 2024].

Smyth, N., 2023. *Jetpack Compose 1.3 Essentials.* s.l.:Payload Media.

Virvou, M. & Alepis, E., 2004. *Mobile versus desktop facilities for an e-learning system: users' perspective.* s.l., IEEE.