Programmentwurf: Wer ist der Spion?

Lorenz Scherrer

Advanced Software

Matrikelnummer: **8809469**

Kurs: TINF21B3

Abgabedatum 16.12.2022

Inhaltsverzeichnis

Kapitel 1

Beschreibung der Funktionalität: Wer ist der Spion?

Die entwickelte App ist eine digitale Umsetzung des beliebten Gesellschaftsspiels "Wer ist der Spion?". Ziel der App ist es, den Spaß und die Spannung des Spiels auf Mobilgeräte zu bringen und es den Benutzern zu ermöglichen, jederzeit und überall zu spielen. Die Hauptfunktionen der App umfassen:

- **Spielrunden erstellen:** Benutzer können neue Spielrunden erstellen und Mitspieler einladen, entweder aus ihrer Kontaktliste oder durch Generierung eines Einladungslinks.
- **Rollen zuweisen:** Die App weist automatisch jedem Spieler eine Rolle zu, entweder als normaler Spieler mit Kenntnis des geheimen Themas oder als Spion, der das Thema nicht kennt.
- **Fragen stellen und Antworten geben:** Die Spieler können sich gegenseitig Fragen stellen, um Hinweise auf das geheime Thema zu erhalten, während der Spion versucht, das Thema zu erraten, ohne entdeckt zu werden.
- **Punkte vergeben und Spielstatistiken anzeigen:** Die App verfolgt die Spielrunden und Punkteverteilung, um den Spielfortschritt zu verfolgen, und zeigt den Spielern ihre Statistiken an.

1.1 Beschreibung des Kundennutzens:

Die App bietet den Benutzern eine unterhaltsame und interaktive Möglichkeit, das Spiel "Wer ist der Spion?ßu spielen, ohne physische Karten oder Zubehör zu benötigen. Der Kundennutzen umfasst:

- **Flexibilität:** Spieler können das Spiel jederzeit und überall spielen, indem sie einfach ihre Mobilgeräte verwenden.
- **Soziale Interaktion:** Das Spiel fördert die soziale Interaktion zwischen den Spielern, da sie sich gegenseitig Fragen stellen und versuchen, den Spion zu entlarven.
- **Spaß und Spannung:** Die App bietet den Spielern die Möglichkeit, den Nervenkitzel des Spiels zu erleben, während sie versuchen, das geheime Thema zu erraten oder ihre wahre Identität als Spion zu verbergen.

1.1.1 Verwendete Technologie

Die Anwendung wird in Kotlin entwickelt, einer modernen und leistungsstarken Programmiersprache, die sich besonders gut für die Entwicklung von Android-Apps eignet. Weitere Technologien und Frameworks umfassen:

- **Back-End:** Kotlin
- **Front-End-Framework:** Kotlin-Ktor Client
- **Entwicklungsumgebung:** IDE oder Code-Editor

1.1.2 Ausführbare Datei

Man kann das Projekt als Jar kompilieren und ausführen lassen. Unter diesem Commit-Hash unter build/libs/SpyGame.jar

Kapitel 2

Clean Architecture

Die Clean Architecture ist ein Architekturmuster, das darauf abzielt, Softwareprojekte gut strukturiert, wartbar und testbar zu gestalten. Ein zentrales Thema bei der Entwicklung von Software ist die Wahl zwischen der Verwendung von Libraries und Frameworks. In diesem Kontext wird argumentiert, dass es oft vorteilhafter ist, auf Libraries zurückzugreifen anstatt auf Frameworks zu setzen.

2.1 Kriterien für nachhaltige Architektur

- besitzt einen technologieunabhängigen Kern: der Domain und Applikation ist nur aus Kotlin Code ohne Bibliotheken.
- behandelt jede Abhängigkeit als temporäre Lösung
- Unterscheidet zwischen zentralem (*langlebigem*) und peripherem (*kurzlebigem*) Sourcecode
- **Metapher:** Die Zwiebel - alle Änderungen sollen den Kern nicht verändern

Peripherie: Hier werden die kurzlebigen Bibliotheken an den Kern angebunden. Abhängigkeiten gehen nur vom peripheren Code in den Kern, nie umgekehrt.

2.2 Dependency Inversion

:Interfaces verwenden damit der Kern Code nicht verändert werden muss. Es kann auch anders gelöst werden die Programmiersprache braucht keine interfaces. Die Kern-Klasse muss die andere nicht kennen. Schalter(**Kern**), Schalterbar(**Plugin**), Lampe/Ventilator(**Bibliotheken**) So in der Theorie.

Um die Dependency Inversion zu erreichen, werden wir eine Abstraktion einführen, die es uns ermöglicht, WordRepository von der konkreten GameLogic-Klasse zu entkoppeln. Wir erstellen eine Schnittstelle für die GameLogic-Klasse und lassen ImplWordRepsoitory stattdessen von dieser Schnittstelle abhängen.

Listing 2.1: Example for Dependency Inversion

```
1 package domain
2
3 class GameLogic(private val wordRepository: WordRepository, private val dataRepository: DataRepository, pr
4     var users = mutableListOf<User>()
```

```
5
6       fun initializeGame(numberofSpies:Int, numberofUsers:Int){
7           val numberofPlayers = numberofUsers-numberofSpies
8           dataRepository.setUser(userRepository.createUsers(numberofSpies, numberofPlayers))
9           dataRepository.setWord(wordRepository.getRandomWord(wordRepository.loadWords()))
10          this.users = dataRepository.getAllUsers().toMutableList()
11      }
12
13      fun displayOneRole() {
14          if (users.isEmpty()) { return }
15
16          val randomUser = userRepository.selctRandomUser(users)
17          users.remove(randomUser)
18          randomUser.displayRole()
19          if (randomUser is Player) {
20              printPlayerWord(randomUser)
21          }
22      }
23
24      private fun printPlayerWord(player: Player) {
25          println("Word: ${dataRepository.getWord()?.name}")
26      }
27
28
29      fun endGame(){
30          userRepository.displayAllUserRoles(dataRepository.getAllUsers().toMutableList())
31      }
32  }
33
34  package domain
35
36  interface WordRepository {
37      fun loadWords():List<Word>
38      fun createWord(wordId: Int, word: String): Word
39      fun getRandomWord(words: List<Word>): Word
40  }
41
42  package applikation
43
44  import domain.Player
45  import domain.Spy
46  import domain.User
47  import domain.UserRepository
48
49
50  class ImpUserRepository : UserRepository{
51      override fun createSpy(spyId: Int): Spy {
52          return Spy(spyId,"Spy $spyId")
53      }
54
55      override fun createPlayer(playerId: Int): Player {
56          return Player(playerId, "Player $playerId")
57      }
58
59      override fun createUsers(numberOfSpies: Int, numberOfPlayers: Int): List<User> {
60          val users = mutableListOf<User>()
61          for (i in 1..numberOfSpies) {
62              users.add(createSpy(i))
63
64          }
65          for (i in 1..numberOfPlayers) {
66              users.add(createPlayer(i))
67          }
68          return users
69      }
70
71
72
73      override fun displayAllUserRoles(userList: List<User>) {
74          userList.forEach { user ->
75              println("${user.username}: ")
76              user.displayRole()
77          }
78      }
79
80      override fun selctRandomUser(userList: MutableList<User>): User {
```

```
81          val randomIndex = (0..<userList.size).random()
82          return userList.removeAt(randomIndex)
83      }
84  }
```

Der Applikation Code ruft Code aus der Adapter Layer auf ohne es zu merkenCommit-Hash.

Die GameLogic-Klasse kennt also nicht die konkreten Implementierungen der Methoden in den Repositories. Stattdessen verwendet sie die definierten Methoden des Interfaces. Dies ermöglicht eine hohe Flexibilität und Austauschbarkeit der Implementierungen. Zum Beispiel kann die GameLogic-Klasse problemlos mit einer anderen Implementierung des UserRepository arbeiten, solange diese die gleichen Methoden des UserRepository-Interfaces implementiert.

Durch die Verwendung von abstrakten Interfaces und die Abhängigkeit von ihnen anstelle von konkreten Implementierungen wird das Prinzip der Dependency Inversion erreicht, was eine lockerere Kopplung und eine bessere Testbarkeit der GameLogic-Klasse ermöglicht.

Hier sieht man noch die Dependency Inversion bildlich dargestellt 2.1.

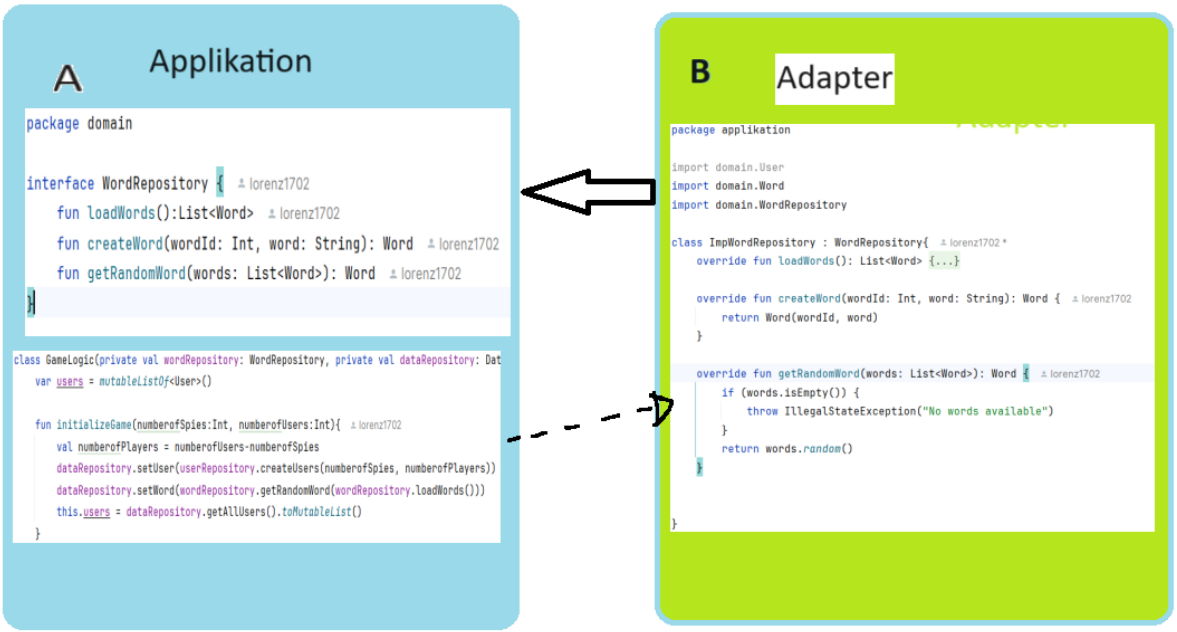


Abbildung 2.1: Dependency Inversion

2.3 Struktur der Clean Architecture

2.3.1 Domain Code

In diesem Bereich wird die Benutzer-Entität definiert.

2.3.2 Applikationscode

Hier werden die Anwendungsfälle (Use Cases) implementiert, die die Kernfunktionalität der Anwendung umsetzen.

2.3.3 Plugin

Adapter dienen als Vermittler zwischen der Daten und der inneren Schicht der Anwendung. Sie übermitteln Daten und führen Aufrufe der inneren Schicht aus. Des Weiteren konvertieren sie Datentypen.

2.3.4 Adapter

Die Adapter-Schicht, die den Ktor-Server implementiert, zeichnet sich durch ihre hohe Austauschbarkeit und Flexibilität aus. Diese Schicht fungiert als Schnittstelle zwischen dem Anwendungskern und dem Ktor-Framework, das für die Verarbeitung von HTTP-Anfragen und -Antworten verantwortlich ist.

Durch die Verwendung des Adaptermusters kann die Implementierung des Ktor-Servers leicht ausgetauscht werden, ohne dass dies Auswirkungen auf den Rest der Anwendung hat. Dies ermöglicht es, verschiedene Implementierungen des Servers zu verwenden, je nach den spezifischen Anforderungen oder Präferenzen des Projekts.

Die Austauschbarkeit der Adapter-Schicht bietet auch die Möglichkeit, verschiedene Technologien oder Frameworks zu evaluieren und zu testen, ohne dass dies größere Änderungen am Anwendungskern erfordert. Dies fördert die Flexibilität und Erweiterbarkeit der Anwendung und erleichtert die Wartung und Weiterentwicklung im Laufe der Zeit.

Funktionsweise von Serveraufrufen

Die Funktionalität der Serveraufrufe soll eine Schnittstelle bereitstellen, um Daten zwischen dem Server und der Anwendungslogik auszutauschen.

2.3.5 Framework vs. Library

Problem: Das Problem von Abhängigkeiten ist, dass sie sich immer ändern, je nach den Frameworks, mit denen man arbeitet. Die verwendeten Frameworks ändern sich alle 5-10 Jahre. Wenn man nicht alle 5 Jahre alles neu machen möchte, muss man die Abhängigkeiten auflösen.

Lösung: Entscheidungen am besten möglichst spät treffen. **Minimalziel**: Entscheidungen müssen revidierbar sein können $\Longrightarrow$ mit möglichst geringen negativen Folgen.

Im Gegensatz dazu bieten Libraries nur spezifische Funktionen oder Werkzeuge, ohne eine vorgegebene Struktur aufzuerlegen. Dies ermöglicht es Entwicklern, die Bibliotheken selektiv und flexibel einzusetzen, je nach den Anforderungen ihres Projekts. Sie bleiben weniger anfällig für Änderungen in der Technologielandschaft, da sie weniger in die Infrastruktur des Projekts eingebettet sind.

Durch die Verwendung von Libraries statt Frameworks wird die Abhängigkeit umgekehrt: Statt dass das Projekt von der Funktionsweise des Frameworks abhängt, sind die Libraries von der Implementierung des Projekts abhängig. Dies erhöht die Kontrolle über den Code und erleichtert es, Änderungen vorzunehmen oder Technologien auszutauschen, wenn dies erforderlich ist.

In der Clean Architecture wird daher empfohlen, Libraries anstelle von Frameworks zu verwenden, um die Flexibilität, Wartbarkeit und Testbarkeit der Software zu verbessern. Diese Herangehensweise ermöglicht es, die Architektur des Projekts sauber zu halten und die Geschäftslogik von technischen Details zu trennen, was letztendlich zu einer robusteren und ska-

lierbareren Software führt.

Um die Dependency Rule der Clean Architecture zu befolgen, habe ich meine Libraries und Frameworks sehr weit nach außen in die äußersten Schichten meines Projekts verschoben. Konkret habe ich die Implementierungsdetails in die Plugin- und Adapter-Schicht ausgelagert. Dadurch wird die innere Geschäftslogik meiner Anwendung von externen Abhängigkeiten isoliert, was die Wartbarkeit und Flexibilität erhöht.

Durch die Anwendung der Clean Architecture in meinem Ktor Kotlin Server-Projekt strebe ich danach, einen klaren und gut strukturierten Code zu entwickeln, der leicht zu warten, zu testen und zu erweitern ist. Die klare Trennung der Verantwortlichkeiten und die Verwendung von Libraries anstelle von Frameworks unterstützen dieses Ziel und tragen zu einer langfristig erfolgreichen Entwicklung meines Projekts bei.

Commit-Hash

Kapitel 3

Programming Principles

3.1 SOLID

Single responsibility principle

Die Single Responsibility Principle (SRP) besagt, dass eine Klasse nur einen Grund zur Änderung haben sollte, was bedeutet, dass sie nur eine Verantwortlichkeit haben sollte. Lassen Sie uns die bereitgestellte Spy-Klasse analysieren:

```
1    class Spy(
2        override val id: Int,
3        override var username: String
4    ) : User {
5        override fun displayRole() {
6            println("Ich bin ein Spion")
7        }
8    }
```

Diese Klasse scheint einen bestimmten Benutzertyp, einen SSpion", zu repräsentieren. Sie implementiert das `User`-Interface und bietet eine spezifische Implementierung für die Funktion `displayRole()` an, die Ïch bin ein Spionäusgibt.

Basierend auf dem SRP sind die Verantwortlichkeiten dieser Klasse:

1. Repräsentation eines Benutzertyps (Spion).
2. Bereitstellung einer Implementierung für die Funktion `displayRole()` spezifisch für die Spion-Rolle.

Da sich die Klasse auf das Verhalten eines Spion-Benutzers konzentriert, scheint sie dem Single Responsibility Principle zu entsprechen. Sie hat einen klaren Zweck und nur einen Grund zur Änderung: wenn das Verhalten oder die Attribute eines Spion-Benutzers geändert werden müssen. Wenn jedoch das Verhalten der `displayRole()`-Funktion aus verschiedenen Gründen, die nicht spezifisch für den Spion-Benutzer sind, häufig geändert werden könnte, wäre es möglicherweise besser, diese Verantwortung an eine separate Klasse zu delegieren. Insgesamt erfüllt die Klasse, solange die Verantwortlichkeiten der Klasse fokussiert und zusammenhängend bleiben, das SRP.

Open/Closed Principle

Listing 3.1: Open/Closed Principle

```
package domain

interface User {
    val id: Int
    var username: String
    fun displayRole()
}


class Spy(
    override val id: Int,
    override var username: String
) : User {
    override fun displayRole() {
        println("I am a Spy")
    }
}


// Player class
class Player(
    override val id: Int,
    override var username: String
) : User {
    override fun displayRole() {
        println("I am a Player")
    }
}
```

Es erfüllt das Open/Closed-Prinzip (OCP), da die Interfaces `User` und `DisplayRoleUseCase` offen für Erweiterungen sind, aber geschlossen für Modifikationen.

- Interface `User`: Definiert die Eigenschaften und Methoden, die für alle Benutzer gelten sollen. Durch Implementierung können verschiedene Benutzertypen erstellt werden, die das `User`-Interface verwenden, ohne das Interface selbst zu ändern. Neue Benutzertypen können hinzugefügt werden, indem neue Klassen erstellt werden, die das `User`-Interface implementieren, ohne das Interface selbst zu ändern.

- Klasse `Spy`: Stellt eine spezifische Implementierung eines Benutzers dar (einen Spion). Die Klasse erweitert das `User`-Interface und bietet eine spezifische Implementierung für die `displayRole()`-Methode an. Die Implementierung der `displayRole()`-Methode erweitert die Funktionalität des `User`-Interfaces, ohne es zu ändern.

Liskov substitution principle

Das Liskov Substitutionsprinzip (LSP) ist ein Grundsatz der objektorientierten Programmierung, der besagt, dass Subtypen sich genauso verhalten müssen wie ihre Basistypen. Hier sind einige wichtige Aspekte des LSP:

- Subtypen müssen sich wie ihre Basistypen verhalten: Das bedeutet, dass ein Objekt eines Subtyps anstelle seines Basistyps verwendet werden kann, ohne dass sich das erwartete Verhalten ändert.
- Erweiterung, keine Einschränkung: Ein Subtyp sollte die Funktionalität des Basistyps erweitern können, aber niemals einschränken. Das heißt, die Methoden des Basistyps sollten von Subtypen entweder überschrieben oder erweitert werden, aber nicht entfernt oder in ihrer Funktionalität beschränkt werden.
- Vererbung als "behaves-likeRelation: Statt die Beziehung zwischen Basistyp und Subtyp als rein ïs-aBeziehung zu betrachten, sollten wir sie eher als "behaves-likeRelation verstehen. Das bedeutet, dass ein Subtyp das Verhalten des Basistyps nachahmen sollte.

- Vererbung ist nicht immer die beste Lösung: Obwohl Vererbung ein leistungsfähiges Werkzeug ist, um Code zu organisieren und Funktionalität wiederzuverwenden, ist es nicht immer die beste Wahl. In einigen Fällen kann die Komposition von Objekten über Vererbung bevorzugt werden, insbesondere wenn es um die Zusammenstellung verschiedener Verhaltensweisen geht.

Das LSP ist ein wichtiges Prinzip, das sicherstellt, dass die Vererbungshierarchie eines Systems konsistent und leicht zu verwenden ist. Es fördert eine klare Strukturierung des Codes und trägt zur Wartbarkeit und Erweiterbarkeit von Software bei.

Überall wo man die Elternklasse verwendet soll man auch die Unterklassen verwenden können. Ohne das es zu Problemen kommt. Wenn ich gerade Tier Klasse verwende soll ich an der Stelle auch Hunde verwenden können.

Listing 3.2: Liskov Substitution Principle

```
1
2  abstract class AbstractWordRepository : WordRepository {
3
4      abstract fun getWords(): Array<String>
5
6      override fun loadWords(): List<Word> {
7          val wordsArray = getWords()
8          val words = mutableListOf<Word>()
9          for ((index, word) in wordsArray.withIndex()) {
10             words.add(createWord(index + 1, word))
11         }
12         return words
13     }
14
15     override fun createWord(wordId: Int, word: String): Word {
16         return Word(wordId, word)
17     }
18
19     override fun getRandomWord(words: List<Word>): Word {
20         if (words.isEmpty()) {
21             throw IllegalStateException("No words available")
22         }
23         return words.random()
24     }
25 }
26
27 class KlimacticWordRepository : AbstractWordRepository() {
28     override fun getWords(): Array<String> {
29         return arrayOf(
30             "Arctic",
31             "Antarctic",
32             "Tundra",
33             "Taiga",
34             "Temperate Forest",
35             "Tropical Rainforest",
36             "Grassland",
37             "Desert",
38             "Savanna",
39             "Mediterranean"
40         )
41     }
42 }
43
44 class SportWordRepository : AbstractWordRepository() {
45     override fun getWords(): Array<String> {
46         return arrayOf(
47             "Fussball",
48             "Basketball",
49             "Tennis",
50             "Volleyball",
51             "Schwimmen",
52             "Laufen",
53             "Leichtathletik",
54             "Boxen",
55             "Handball",
56             "Rugby",
```

```
            "Golf",
            "Hockey",
            "Tischtennis",
            "Badminton",
            "Radfahren",
            "Skifahren",
            "Snowboarden",
            "Klettern",
            "Tauchen",
            "Yoga"
        )
    }
}
```

Der vorliegende Code erfüllt das Liskov-Substitutionsprinzip, da die Unterklassen KlimacticWordRepository und SportWordRepository die Verträge der Superklasse AbstractWordRepository einhalten und diese nicht verletzen.

Das Liskov-Substitutionsprinzip besagt, dass Objekte einer abgeleiteten Klasse (Unterklasse) als Ersatz für Objekte ihrer Basisklasse (Superklasse) verwendet werden können, ohne dass sich das erwartete Verhalten des Programms ändert.

Im vorliegenden Code:

1. Die Unterklassen KlimacticWordRepository und SportWordRepository erweitern die abstrakte Klasse AbstractWordRepository.
2. Sie implementieren die abstrakte Methode getWords(), die in der Superklasse definiert ist. Dadurch wird sichergestellt, dass beide Unterklassen das erwartete Verhalten der Superklasse bereitstellen.
3. Die abstrakte Klasse AbstractWordRepository definiert eine Template-Methode loadWords(), die von den Unterklassen verwendet wird, um Wörter zu laden. Die Unterklassen können die Methode getWords() implementieren, um ihre eigenen spezifischen Wörter zurückzugeben, während der Algorithmus zum Laden der Wörter in der Superklasse definiert ist.
4. Da die Unterklassen die Verträge der Superklasse einhalten und das erwartete Verhalten bereitstellen, können sie problemlos anstelle der Superklasse verwendet werden, ohne das Verhalten des Programms zu ändern.

Somit erfüllt der vorliegende Code das Liskov-Substitutionsprinzip.

Interface Segregation Principle

Das Interface Segregation Principle wurde hier schon angewendet 5.2.

Dependency Inversion principle

Das Dependency Inversion Principle wurde hier schon angewendet 2.2.

3.2 GRASP

3.2.1 Low Coupling

Low Coupling bedeutet, dass Klassen oder Module in einem System so wenig wie möglich voneinander abhängig sind. Hier sind einige wichtige Punkte, die dies verdeutlichen:

- **Geringe bzw. lose Kopplung**: Dies bedeutet, dass eine Klasse wenig über die Implementierung anderer Klassen oder Module wissen muss, um ihre Aufgaben zu erfüllen.
- **Kopplung**: Es ist ein Maß für die Abhängigkeit einer Klasse von anderen Klassen oder Modulen. Je höher die Kopplung, desto stärker sind die Verbindungen zwischen den verschiedenen Teilen des Systems.
- **Geringe Kopplung unterstützt:**
  - **Leichte Anpassbarkeit**: Da Änderungen in einer Klasse oder einem Modul weniger wahrscheinlich Auswirkungen auf andere Teile des Systems haben.
  - **Gute Testbarkeit**: Tests können isoliert durchgeführt werden, da weniger Abhängigkeiten zwischen den Komponenten bestehen.
  - **Verständlichkeit**: Durch weniger Kontext oder Abhängigkeiten ist der Code einfacher zu verstehen und zu warten.
  - **Erhöhte Wiederverwendbarkeit**: Lose gekoppelte Komponenten können einfacher in anderen Systemen wiederverwendet werden, da sie weniger Abhängigkeiten haben.

Ein niedriger Kopplungsgrad ist daher ein Ziel in der Softwareentwicklung, da er zu einem flexibleren, besser wartbaren und erweiterbaren System führt. Das Erreichen einer geringen Kopplung erfordert oft die Anwendung verschiedener Designprinzipien und -muster wie Dependency Injection, Interfaces und Abstraktion.

Positive Beispiel

Ein polymorher Aufruf über ein Interface ermöglicht eine geringe Kopplung zwischen verschiedenen Klassen oder Modulen.

Listing 3.3: Polymorher Aufruf über ein Interface

```
var users = mutableListOf<User>()
...
users.remove(randomUser)
randomUser.displayRole()
...
```

In diesem Code 3.4 wird die Methode displayRole() aufgerufen, die Teil des User-Interfaces ist. Je nachdem, ob randomUser ein Spy oder ein Player ist, wird eine unterschiedliche Implementierung dieser Methode ausgeführt. Dies ermöglicht eine lose Kopplung, da die aufrufende Klasse (GameLogic in diesem Fall) nicht wissen muss, welche spezifische Implementierung von User verwendet wird.

Die Implementierungen des User-Interfaces sind wie folgt definiert3.1.

Durch den Einsatz von Polymorphismus über Interfaces wird eine geringe Kopplung erreicht, da die Implementierungsdetails der User-Klassen von der aufrufenden Klasse entkoppelt sind. Dies erleichtert die Wartbarkeit und Erweiterbarkeit des Codes, da Änderungen an den Implementierungen von User keinen direkten Einfluss auf die aufrufende Klasse haben.

Negativ Beispiel

Listing 3.4: Abhängigkeit von direkter Implementierung

```

class GameLogic(private val wordRepository: WordRepository, private val dataRepository: DataRepository, pr
```

```
3     var users = mutableListOf<User>()
4
5
6     fun choseCategories(chosenCategories: Boolean){
7         if (chosenCategories){
8             dataRepository.setWords(SportWordRepository().loadWords())
9             return
10        }
11
12        dataRepository.setWords(KlimacticWordRepository().loadWords())
13    }
```

n der Klasse GameLogic gibt es eine hohe Kopplung zwischen der Implementierung und den konkreten Implementierungen der Repositories (WordRepository, DataRepository und UserRepository). Dies liegt daran, dass die Klasse direkt auf die konkreten Implementierungen von SportWordRepository und KlimacticWordRepository zugreift, um die Wörter zu laden.

Das direkte Erstellen von Instanzen von SportWordRepository und KlimacticWordRepository innerhalb der Methode choseCategories erhöht die Kopplung, da die Klasse stark von den konkreten Implementierungen abhängig ist. Dadurch wird die Flexibilität der Klasse beeinträchtigt, da Änderungen an diesen Implementierungen Auswirkungen auf die GameLogic-Klasse haben könnten.

3.2.2 High Cohesion

High Cohesion ist ein Prinzip des Software-Designs, das sich darauf konzentriert, dass Klassen oder Module eng zusammenarbeiten und sich auf eine klare und spezifische Aufgabe konzentrieren. Dabei geht es darum, dass die Elemente einer Klasse oder eines Moduls stark miteinander verbunden sind und zusammenarbeiten, um eine gemeinsame Verantwortung zu erfüllen.

Einige wichtige Punkte, die High Cohesion charakterisieren, sind:

1. Aufgabenorientierung: Jede Klasse oder jedes Modul sollte eine klare Aufgabe oder Verantwortung haben und sich darauf konzentrieren, diese effizient zu erfüllen.
2. Zusammengehörigkeit: Die Elemente innerhalb einer Klasse oder eines Moduls sollten eng miteinander verbunden sein und zusammenarbeiten, um das übergeordnete Ziel zu erreichen.
3. Wenige externe Abhängigkeiten: Klassen oder Module mit hoher Kohäsion sollten nur wenige oder keine externen Abhängigkeiten haben. Sie sollten weitgehend unabhängig von anderen Teilen des Systems sein, um ihre Aufgabe effektiv erfüllen zu können.
4. Klar abgegrenzte Funktionalität: Jede Klasse oder jedes Modul sollte eine klar definierte Funktionalität haben, ohne sich in verschiedene Richtungen zu verzweigen oder mehrere Aufgaben gleichzeitig zu erfüllen.
5. Einfache Wartbarkeit und Erweiterbarkeit: Durch die klare Abgrenzung der Verantwortlichkeiten und die engen Verbindungen zwischen den Elementen ist der Code leichter wartbar und erweiterbar. Änderungen in einem Teil des Systems haben weniger Auswirkungen auf andere Teile.

Insgesamt führt High Cohesion zu einem klar strukturierten und gut organisierten Code, der einfacher zu verstehen, zu warten und zu erweitern ist. Es trägt dazu bei, die Komplexität des Systems zu reduzieren und die Qualität der Software zu verbessern.

Beispiel

Listing 3.5: High Cohesion

```
1
2  class ImpUserRepository : UserRepository{
3      override fun createSpy(spyId: Int): Spy {
4          return Spy(spyId,"Spy $spyId")
5      }
6
7      override fun createPlayer(playerId: Int): Player {
8          return Player(playerId, "Player $playerId")
9      }
10
11     override fun createUsers(numberOfSpies: Int, numberOfPlayers: Int): List<User> {
12         val users = mutableListOf<User>()
13         for (i in 1..numberOfSpies) {
14             users.add(createSpy(i))
15
16         }
17         for (i in 1..numberOfPlayers) {
18             users.add(createPlayer(i))
19         }
20         return users
21     }
22
23
24
25     override fun displayAllUserRoles(userList: List<User>) {
26         userList.forEach { user ->
27             println("${user.username}: ")
28             user.displayRole()
29         }
30     }
31
32     override fun selctRandomUser(userList: MutableList<User>): User {
33         val randomIndex = (0..<userList.size).random()
34         return userList.removeAt(randomIndex)
35     }
36 }
```

Der vorliegende Code3.5 erfüllt das High Cohesion-Prinzip. Hier sind die Gründe:

1. **Single Responsibility Principle (SRP)**: Die `ImpUserRepository`-Klasse hat eine klare Verantwortung, nämlich die Erstellung von Benutzern und die Anzeige ihrer Rollen. Dies entspricht dem SRP, da sie nur eine Aufgabe hat.
2. **Fokus auf Benutzeroperationen**: Die Klasse konzentriert sich auf Operationen im Zusammenhang mit Benutzern, einschließlich der Erstellung von Spielern und Spionen, der Anzeige ihrer Rollen und der Auswahl eines zufälligen Benutzers. Dadurch bleibt der Fokus der Klasse auf einer bestimmten Domäne, was die Kohäsion erhöht.
3. **Kohäsive Methoden**: Alle Methoden in der Klasse haben einen klaren Zusammenhang zu Benutzeroperationen. Die Methoden sind eng miteinander verbunden und dienen einem gemeinsamen Zweck, was die Kohäsion erhöht.
4. **Wiederverwendbarkeit**: Die Klasse ist darauf ausgelegt, wieder verwendbar zu sein, da sie unabhängig von anderen Teilen des Systems ist und ihre Funktionen isoliert sind.

Insgesamt erfüllt der Code das High Cohesion-Prinzip, da er eine klare Verantwortung hat, sich auf spezifische Benutzeroperationen konzentriert und kohäsive Methoden aufweist.

Negativ Beispiel

Listing 3.6: High Cohesion

```
1 class GameLogic(private val wordRepository: WordRepository, private val dataRepository: DataRepository, pr
2         var users = mutableListOf<User>()
3
4
```

```
5           fun choseCategories(chosenCategories:Boolean){
6               if (chosenCategories){
7                   dataRepository.setWords(SportWordRepository().loadWords())
8                   return
9               }
10
11              dataRepository.setWords(KlimacticWordRepository().loadWords())
12          }
13
14      fun initializeGame(numberofSpies:Int, numberofUsers:Int){
15          val numberofPlayers = numberofUsers-numberofSpies
16          dataRepository.setUser(userRepository.createUsers(numberofSpies, numberofPlayers))
17          dataRepository.setWord(wordRepository.getRandomWord(dataRepository.getAllWords()))
18          this.users = dataRepository.getAllUsers().toMutableList()
19      }
20
21      fun displayOneRole() {
22          if (users.isEmpty()) { return }
23
24          val randomUser = userRepository.selctRandomUser(users)
25          users.remove(randomUser)
26          randomUser.displayRole()
27          if (randomUser is Player) {
28              printPlayerWord(randomUser)
29          }
30      }
31
32      private fun printPlayerWord(player: Player) {
33          println("Word: ${dataRepository.getWord()?.name}")
34      }
35  }
```

In diesem Code 3.6 zeigt sich eine geringe Kohäsion, da verschiedene Aufgaben in einer Klasse zusammengefasst sind, die möglicherweise nicht eng miteinander verbunden sind. Hier sind einige Beispiele:

- choseCategories-Methode: Diese Methode ist für die Auswahl von Kategorien zuständig, was sich auf die Spielkonfiguration bezieht. Sie greift jedoch auf externe Repositories wie SportWordRepository und KlimacticWordRepository zu, um Wortlisten zu laden. Dies führt zu einer geringen Kohäsion, da die Auswahl von Kategorien und das Laden von Wortlisten zwei unterschiedliche Aufgaben sind und nicht eng miteinander verbunden sind.
- initializeGame-Methode: Diese Methode ist für die Initialisierung des Spiels zuständig, was eine andere Aufgabe ist als die Auswahl von Kategorien oder das Laden von Wortlisten. Auch hier gibt es eine geringe Kohäsion, da verschiedene Aspekte des Spiels in derselben Klasse zusammengefasst sind.
- displayOneRole-Methode: Diese Methode ist für das Anzeigen der Rolle eines Spielers während des Spiels verantwortlich. Obwohl dies eine Aufgabe im Kontext des Spiels ist, gibt es eine Trennung zwischen dem Anzeigen der Rolle und der Auswahl von Kategorien oder der Initialisierung des Spiels.

Um die Kohäsion zu verbessern, könnten diese verschiedenen Aufgaben in separaten Klassen oder Modulen organisiert werden, die jeweils eine klar definierte Verantwortlichkeit haben. Dadurch würde der Code klarer strukturiert und besser wartbar sein.

3.3 DRY

DRY (Don't Repeat Yourself) ist ein Prinzip der Softwareentwicklung, das darauf abzielt, Redundanz im Code zu vermeiden und eine klare, unzweideutige Darstellung von Wissen im System sicherzustellen. Hier sind einige wichtige Punkte, die DRY verdeutlichen:

- Das Prinzip lautet "Mache alles einmal und nur einmal", was bedeutet, dass jede Information oder Funktionalität im System nur an einem einzigen Ort vorhanden sein sollte.
- Änderungen sollten an einer einzigen Stelle vorgenommen werden können, und diese Änderungen sollten sich auf alle relevanten Teile des Systems auswirken.
- DRY betrifft nicht nur den Code selbst, sondern auch andere Artefakte wie Skripte, Testpläne und Dokumentationen.
- DRY ist nicht dasselbe wie das Singleton-Muster. Es interessiert sich nicht für die Anzahl von Objekten zur Laufzeit, sondern darauf, dass jedes Stück Wissen eine einzige, unzweideutige Repräsentation im System hat.

Das Motto für DRY besagt, dass jedes Wissensaspekt eine einzige, unzweideutige, autoritative Repräsentation innerhalb eines Systems haben sollte. Dabei ist mechanische Duplikation erlaubt, solange die Originalquelle klar definiert ist.

Hier findet sich Code 5.2 der gegen DRY verstößt.

Dieser Code verstößt gegen das Prinzip "Don't Repeat Yourself"(DRY), weil es zwei separate Funktionen gibt (loadKlimacticWords und loadSportWords), die im Wesentlichen denselben Code enthalten, um eine Liste von Wörtern zu erstellen. Der einzige Unterschied zwischen den beiden Funktionen besteht in den Arrays von Wörtern, die sie verwenden. Durch die Wiederholung des Codes entsteht Redundanz und erhöhter Wartungsaufwand. Wenn sich die Art und Weise ändert, wie die Wörter geladen werden sollen, müssten beide Funktionen geändert werden, was das Risiko von Fehlern erhöht.

Um das DRY-Prinzip anzuwenden, könnten Sie eine einzige Funktion erstellen, die eine Liste von Wörtern basierend auf einem übergebenen Array von Wörtern erstellt. Auf diese Weise würde der Code an einer einzigen Stelle definiert und gewartet werden, was die Wartbarkeit und Lesbarkeit des Codes verbessert und das Risiko von Inkonsistenzen reduziert.

Es wird dann mit dem Liskov Substitutionsprinzip behoben 3.1.

Kapitel 4

Test

Es wurden im späteren Verlauf Änderungen durchgeführt. Es wurde ein Branch angelegt in dem die alle Test vorhanden sind TestBranch.

Menschen sind die Architekten der digitalen Welt, aber sie sind auch fehlbar. Jeder, der Software entwickelt, weiß, dass Fehler unvermeidlich sind. Und diese Fehler können teuer sein - in Geld, Zeit und sogar in Bezug auf Vertrauen und Nerven. Die Kosten eines Fehlers steigen mit der Zeit seiner Existenz. Selbst wenn ein Fehler sofort erkannt und behoben wird, hinterlässt er dennoch seine Spuren.

In Deutschland sind Softwaretests gesetzlich vorgeschrieben, und das aus gutem Grund. Das Nichttesten wird als "grob fahrlässigängesehen. Tests helfen dabei, gewollte Funktionen von zufälligen Features zu unterscheiden. Legacy Code, also Code ohne Tests, kann zu einem schwerwiegenden Problem werden.

Einige Entwickler setzen Tests als Werkzeug in ihrer täglichen Arbeit ein. Testgetriebene Entwicklung ist eine Methode, bei der Tests vor dem eigentlichen Code geschrieben werden. Dies stellt sicher, dass jede Funktionalität von Tests begleitet wird und somit die Qualität und Stabilität der Software gewährleistet wird. Letztendlich sind Tests ein unverzichtbarer Bestandteil des Entwicklungsprozesses, der dazu beiträgt, dass Software fehlerfrei und vertrauenswürdig ist.

4.1 Unit Test

Unit-Tests, auch bekannt als Komponententests, sind eine grundlegende Form von Tests in der Softwareentwicklung, die darauf abzielen, die korrekte Implementierung einer einzelnen Komponente, oft einer Methode oder Funktion, sicherzustellen. Bei Unit-Tests wird nur der relevante Teil des Systems gestartet, während alle anderen Teile durch Stellvertreter ersetzt werden. Dies ermöglicht es, isoliert auf die Funktionalität der Komponente zu fokussieren.

Die Durchführung von Unit-Tests erfolgt in der Regel mithilfe eines Testframeworks, das Methodenaufrufe der zu testenden Komponente simuliert. Dabei werden die Rückgabewerte der Methoden überprüft, um sicherzustellen, dass sie den erwarteten Ergebnissen entsprechen.

Das Hauptziel von Unit-Tests besteht darin, die korrekte Implementierung der einzelnen Komponente sicherzustellen. Dabei liegt der Fokus auf der Prüfung der Funktionalität und des Verhaltens der Komponente unter verschiedenen Bedingungen. Unit-Tests testen jedoch nicht das Zusammenspiel zwischen verschiedenen Komponenten oder die Performance des Systems, sondern konzentrieren sich ausschließlich auf die Funktionalität der einzelnen Einheiten.

Hier wurden jetzt 10 Unit Test verwendet um die GameReository Klasse zu test. Zuvor wurde der zu testende Code nicht vollständig überprüft. Von den 10 Unit tests fallen vier durch. Damit haben die Unit tests ihr Ziel erfüllt die Korrektheit der Implementierung zu überprüfen.

Listing 4.1: Test and the wrong code

```
    @Test
    fun createUsers() {
        // Arrange
        val users = listOf(
            Player(1, "Player 1"),
            Player(2, "Player 2"),
            Spy(3, "Spy 1")
        )
        val game = InMemoryGameRepository()
        game.createUsers(1,2)

        assertEquals(users, game.getAllUser())

override fun createUsers(numberOfSpies: Int, numberOfPlayers: Int) {
    for (i in 1..numberOfSpies) {
        -createSpy(i)
        +this.addSpy(createSpy(i))
        }
    for (i in 1..numberOfPlayers) {
        -createPlayer(i)
        +this.addPlayer(createPlayer(i))
    }
}

}
```

In diesem Beispiel hat der Unit-Test einen entscheidenden Sinn gezeigt. Er ist durchgefallen, und dadurch wurde entdeckt, dass die Liste keine Benutzer enthielt, weil vergessen wurde, die Benutzer in der Liste zu speichern. Dieser Fehler wurde aufgedeckt, weil der Test die erwartete Funktionalität mit der tatsächlichen Implementierung verglich. Dies verdeutlicht die Bedeutung von Unit-Tests, um sicherzustellen, dass jede Komponente einer Software gemäß den Anforderungen funktioniert.4.1

4.2 ATRIP-Regeln

Fünf Regeln für gute Unit Tests

Automatic - Eigenständig

Automatic bedeutet, dass Tests eigenständig ablaufen müssen, ohne dass manuelle Eingriffe erforderlich sind. Dies bedeutet, dass keine Dialoge weggeklickt werden müssen und keine manuellen Werteeingaben erfolgen dürfen. Die Tests müssen in der Lage sein, ihre Ergebnisse selbst zu überprüfen. Dabei ist für jeden Test nur das Ergebnis "bestandenöder "nicht bestandenßulässig. Wenn ein Test nicht besteht, wird dies als "gebrochen"bezeichnet. Durch die Einhaltung dieser Regeln wird eine Testdurchführung ohne zusätzliches Wissen ermöglicht, was wiederum die Automatisierung der Tests erleichtert.Commit-Hash Alle Tests sind automatisch ausführbar.

Thorough - Gründlich

Tests sollten nicht nur oberflächlich sein, sondern durchdringend und gründlich, um sicherzustellen, dass alle relevanten Aspekte einer Anwendung ordnungsgemäß funktionieren. Das Konzept

von "thoughïn Bezug auf Tests bedeutet, dass sie alles Notwendige abdecken, um die Anforderungen und Rahmenbedingungen der Anwendung zu erfüllen. Dies wird in diesem Commit erfüllt in dem die Core Funktionen gestestet werden die Missions Kritisch sind.Commit-Hash

Repeatable - Wiederholbar

Der Test der diese Funktion getestet hatte war nicht gut wiederholbar der unterschiedliche Ergebnisse geliefert hat, da die Funktion einen Zufall enthällt. Jetzt werden die Rahmenbedingungen getestet um sicherzustellen das die Funktion korrekt abläuft.

Listing 4.2: Repetable

```
override fun DisplayOneRole() {

        if (users.isEmpty()) {
            println("No more users left.")
            return
        }

        val randomIndex = (0 until users.size).random()
        val randomUser = users.removeAt(randomIndex)

        println("${randomUser.username}:")
        randomUser.displayRole()
        if (randomUser is Player) {
                println("Word: ${Game.getWord()?.name}")  // Print the word if the user is a player and no
        }
}
@Test
fun displayOneRole() {
    // Arrange
    gameRepository = mock()
    impCoreFunktions = ImpCoreFunktions(gameRepository)
    val player1 = Player(1, "Player 1")
    val player2 = Player(2, "Player 2")
    val spy1 = Spy(3, "Spy 1")
    val usersList = mutableListOf<User>(player1, player2, spy1)

    impCoreFunktions.users = usersList.toMutableList()

    // Mocking gameRepository to return a word
    `when`(gameRepository.getWord()).thenReturn(Word(1,"TestWord"))

    // Act
    impCoreFunktions.DisplayOneRole()

    // Assert
    assertEquals(usersList.size - 1, impCoreFunktions.users.size)
}
```

In dem wird getestet, ob die Größe der Liste um eins verkleinert wurde.4.2

Independent - Unabhängig voneinander

Tests müssen jederzeit in beliebiger Zusammenstellung und Reihenfolge funktionsfähig sein. Das bedeutet, dass keine impliziten Abhängigkeiten zwischen den Tests bestehen sollten, wie beispielsweise persistente Datenstrukturen wie Datenbanken oder Dateisysteme. Idealerweise sollte jeder Test genau einen Aspekt der Komponente überprüfen. Im Falle eines Fehlers sollte ein Test fehlschlagen, nicht hunderte, und die Ursache für den Testausfall sollte möglichst direkt ableitbar sein. Dies ermöglicht eine effiziente Fehlersuche und -behebung während des Testprozesses.

Hier wird die StartGame()-Methode gemockt, sodass, falls die StartGame()-Methode einen Fehler hat, nur der Test für die StartGame()-Methode abbricht und nicht auch die anderen. Siehe 4.3.

Listing 4.3: Independent

```
fun displayOneRole() {
    // Arrange
    gameRepository = mock()
    impCoreFunktions = ImpCoreFunktions(gameRepository)
    val player1 = Player(1, "Player 1")
    val player2 = Player(2, "Player 2")
    val spy1 = Spy(3, "Spy 1")
    val usersList = mutableListOf<User>(player1, player2, spy1)

    impCoreFunktions.users = usersList.toMutableList()

    // Mocking gameRepository to return a word
    `when`(gameRepository.getWord()).thenReturn(Word(1,"TestWord"))

    // Act
    impCoreFunktions.DisplayOneRole()
    }
```

Profressional - Mit Sorgfalt hergestellt

Test Code ist ebenso produktionsrelevant wie der eigentliche Code. Dennoch unterliegt Test Code selbst nicht automatischen Tests. Fehler in Tests können schwerwiegend sein und kostenintensive Reparaturen nach sich ziehen. Daher sollte Test Code so klar und verständlich wie möglich sein. Es ist nicht ratsam, die Anzahl der Tests rein quantitativ zu erhöhen, ohne deren Qualität zu berücksichtigen. Ebenso sind Tests für irrelevante Aspekte nicht zielführend und sollten vermieden werden.

Listing 4.4: Profressional

```
@Test
fun endGame() {

    // Arrange
    gameRepository = mock()
    impCoreFunktions = ImpCoreFunktions(gameRepository)
    val player1 = Player(1, "Player 1")
    val player2 = Player(2, "Player 2")
    val spy1 = Spy(3, "Spy 1")
    val usersList = mutableListOf<User>(player1, player2, spy1)

    impCoreFunktions.users = usersList.toMutableList()

    // Act
    impCoreFunktions.EndGame()

    // Assert
    verify(gameRepository, times(1)).userdisplaythereRole()
}
```

Lesbarkeit und Verständlichkeit: Der Test ist gut strukturiert und dokumentiert, was er überprüft und welche Interaktionen erwartet werden. Dadurch ist der Test leicht verständlich und kann von anderen Entwicklern einfach nachvollzogen werden .

4.3 Mocked Test

Mock-Objekte, oft einfach als "Mocks"bezeichnet, sind einfache Stellvertreter für ëchteÖbjekte in der Softwareentwicklung. Sie dienen dazu, Abhängigkeiten während eines Tests zu ersetzen, ähnlich wie ein Licht-Double in der Filmindustrie nur die beleuchtungsrelevanten Eigenschaften eines echten Stars besitzt.

Die Verwendung von Mocks ermöglicht die Isolation von Klassen während des Tests. Um eine Klasse isoliert testen zu können, müssen ihre Abhängigkeiten durch Mock-Objekte ersetzt wer-

den. Diese Mocks bieten eine Minimalumsetzung der notwendigen Funktionalität, auch bekannt als Fakes, die "gut genug"für den Testeinsatz sind.

Obwohl selbst programmierte Mocks zeitaufwendig sein können, lohnt sich der Einsatz von Mock-Tools, die speziell für diesen Zweck entwickelt wurden. Mocks müssen für ihren jeweiligen Einsatzzweck "trainiert"werden und durchlaufen dabei drei Phasen: das Einlernen (Training-Phase), das Abspielen (Einsatz-Phase) und die Überprüfung (Verifikation-Phase). Ein Beispiel dafür findet man hier 4.2 und in diesem Commit-Hash

Die verify-Methode in Mockito wird verwendet, um sicherzustellen, dass bestimmte Interaktionen mit Mock-Objekten während des Tests aufgetreten sind. Sie ermöglicht es, zu überprüfen, ob bestimmte Methoden eines Mock-Objekts aufgerufen wurden und mit welchen Argumenten.

Im Kontext des zuvor bereitgestellten Unit-Tests wird verify verwendet, um sicherzustellen, dass die StartGame-Funktion der ImpCoreFunktions-Klasse bestimmte Methoden des GameRepository-Interfaces mit den erwarteten Argumenten aufruft. Zum Beispiel:

```
1
2 verify(gameRepository).createUsers(numberOfSpys, numberOfUsers - numberOfSpys)
```

Diese Zeile überprüft, ob die createUsers-Methode des GameRepository-Mockobjekts mit den angegebenen Argumenten (numberOfSpys und numberOfUsers - numberOfSpys) während der Ausführung der StartGame-Funktion aufgerufen wird.

Zusammenfassend ist verify eine Mockito-Methode, die verwendet wird, um Methodenaufrufe auf Mock-Objekten zu überprüfen, um sicherzustellen, dass das zu testende System sich wie erwartet verhält.

Durch die Verwendung von Mock-Objekten und der verify-Methode von Mockito wird die Interaktion mit dem gameRepository überprüft, ohne tatsächliche Änderungen am GameRepository vorzunehmen. Dies stellt sicher, dass der Test unabhängig von externen Ressourcen ist und jederzeit wiederholbar ist. Darüber hinaus ist der Test leicht verständlich, da er klar angibt, welches Verhalten überprüft wird und welche Interaktionen erwartet werden. Somit trägt dieser Test zur Stabilität und Qualität des Codes bei, indem er sicherstellt, dass die EndGame-Funktion wie erwartet funktioniert.

4.4 Code Coverage

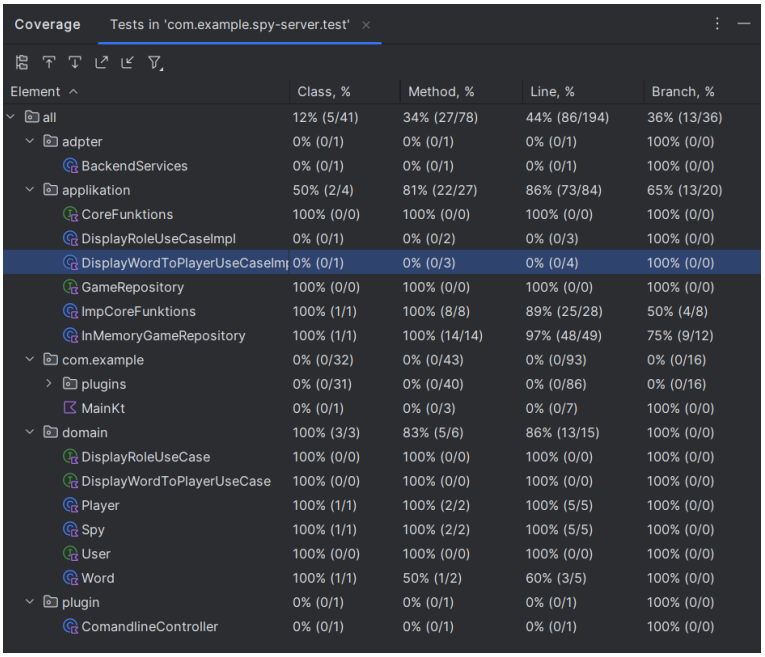
Testabdeckung, auch bekannt als Code Coverage, ist ein Maß dafür, wie viel von Ihrem Quellcode während der Ausführung Ihrer Tests abgedeckt wird. Es gibt verschiedene Arten von Testabdeckung, von denen die beiden gängigsten Line Coverage und Branch Coverage sind.

Die Line Coverage misst, welche Zeilen Ihres Codes von Ihren Tests durchlaufen werden. Eine Zeile gilt als abgedeckt, wenn sie während der Testausführung mindestens einmal ausgeführt wird.

Die Branch Coverage hingegen betrachtet die Verzweigungen oder Entscheidungspunkte in Ihrem Code. Sie misst, wie viele Verzweigungspunkte durch Ihre Tests abgedeckt werden, indem sie sowohl den Wahrheitswert true als auch false erreichen.

Es ist wichtig zu beachten, dass die Testabdeckung als Bezugspunkt dient und nicht als alleiniges Maß für die Qualität Ihrer Tests betrachtet werden sollte. Eine hohe Testabdeckung deutet zwar darauf hin, dass ein großer Teil Ihres Codes getestet wurde, sagt jedoch nichts über die Qualität der Tests oder die tatsächliche Fehlerabdeckung aus. Dennoch kann die Testabdeckung ein nützliches Werkzeug sein, um Bereiche Ihres Codes zu identifizieren, die möglicherweise

zusätzliche Tests erfordern. Die Testabdeckung zeigt, dass derzeit nur ein kleiner Teil des Codes



Element	Class, %	Method, %	Line, %	Branch, %
all	12% (5/41)	34% (27/78)	44% (86/194)	36% (13/36)
adpter	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
BackendServices	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
applikation	50% (2/4)	81% (22/27)	86% (73/84)	65% (13/20)
CoreFunktions	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
DisplayRoleUseCaseImpl	0% (0/1)	0% (0/2)	0% (0/3)	100% (0/0)
DisplayWordToPlayerUseCaseImp	0% (0/1)	0% (0/3)	0% (0/4)	100% (0/0)
GameRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ImpCoreFunktions	100% (1/1)	100% (8/8)	89% (25/28)	50% (4/8)
InMemoryGameRepository	100% (1/1)	100% (14/14)	97% (48/49)	75% (9/12)
com.example	0% (0/32)	0% (0/43)	0% (0/93)	0% (0/16)
plugins	0% (0/31)	0% (0/40)	0% (0/86)	0% (0/16)
MainKt	0% (0/1)	0% (0/3)	0% (0/7)	100% (0/0)
domain	100% (3/3)	83% (5/6)	86% (13/15)	100% (0/0)
DisplayRoleUseCase	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
DisplayWordToPlayerUseCase	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
Player	100% (1/1)	100% (2/2)	100% (5/5)	100% (0/0)
Spy	100% (1/1)	100% (2/2)	100% (5/5)	100% (0/0)
User	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
Word	100% (1/1)	50% (1/2)	60% (3/5)	100% (0/0)
plugin	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
ComandlineController	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)

Abbildung 4.1: Code Coverage

durch Tests abgedeckt ist.

Insgesamt beträgt die Testabdeckung für alle Klassen 12,2%. Das bedeutet, dass nur etwa 12,2% des Codes durch Tests getestet werden. Dies ist ein niedriger Wert und deutet darauf hin, dass viele Teile des Codes nicht ausreichend getestet wurden.

Eine detaillierte Analyse der Testabdeckung zeigt, dass einige Pakete und Klassen besser abgedeckt sind als andere:

- Das Paket äpplikation"hat eine Testabdeckung von 50% für Klassen, 81,5% für Methoden, 65% für Verzweigungen und 86,9% für Zeilen. Dies deutet darauf hin, dass dieses Paket besser getestet ist als andere, aber es gibt immer noch Bereiche, die verbessert werden könnten.
- Das Paket "domain"hat eine höhere Testabdeckung, mit 100% für Klassen, 83,3% für Methoden und 86,7% für Verzweigungen. Dies deutet darauf hin, dass dieses Paket gründlicher getestet wurde und weniger ungetestete Bereiche aufweist.
- Die anderen Pakete wie ädpter", "com.exampleünd "com.example.plugins"haben eine Testabdeckung von 0%, was darauf hindeutet, dass sie überhaupt nicht getestet wurden.

Insgesamt zeigt die Analyse der Testabdeckung, dass es noch viel Raum für Verbesserungen gibt. Eine höhere Testabdeckung ist wichtig, um die Zuverlässigkeit und Qualität des Codes sicherzustellen und potenzielle Fehler zu identifizieren. Es ist ratsam, die Testabdeckung schrittweise zu verbessern, indem mehr Tests hinzugefügt und ungetestete Bereiche abgedeckt werden.

Hier wurde die Code Coverage durch geführt Commit-Hash

Kapitel 5

Refactoring

5.1 Code smells identifizieren

Lage Class and Long Method

Listing 5.1: Code Smell: Large Class and Long Method

```
1
2  class InMemoryGameRepository : GameRepository {
3
4      private val users = mutableListOf<User>()
5      private val players = mutableListOf<Player>()
6      private val spies = mutableListOf<Spy>()
7      private var words = mutableListOf<Word>()
8      private var word: Word? = null
9
10     // Add players to the repository
11
12
13     fun addPlayer(player: Player) {
14         players.add(player)
15         users.add(player)
16     }
17
18     // Add spies to the repository
19     fun addSpy(spy: Spy) {
20         spies.add(spy)
21         users.add(spy)
22     }
23
24     // Set the secret word
25     override fun setWord(theword: Word) {
26         this.word = theword
27     }
28
29     override fun getWord(): Word? {
30         return this.word
31     }
32
33
34     // Implement methods from the GameRepository interface
35
36     override fun createPlayer(PlayerId: Int): Player {
37         return Player(
38             PlayerId,
39             "Player $PlayerId"
40         )
41     }
42
43     override fun createSpy(SpyId: Int): Spy {
44         return Spy(SpyId,"Spy $SpyId")
45     }
46
47     fun getAllWords(): MutableList<Word> {
```

```
48          return this.words
49      }
50      override fun LoadWords() {
51          val geographicZones = arrayOf(
52              "Arctic",
53              "Antarctic",
54              "Tundra",
55              "Taiga",
56              "Temperate Forest",
57              "Tropical Rainforest",
58              "Grassland",
59              "Desert",
60              "Savanna",
61              "Mediterranean"
62          )
63
64          words.clear() // Clear existing words
65          for ((index, zone) in geographicZones.withIndex()) {
66              words.add(createWord(index + 1, zone))
67          }
68      }
69
70      override fun createWord(WordId: Int, Word: String): Word {
71
72          return Word(WordId, Word)
73      }
74
75      override fun getAllUser(): List<User> {
76          return users.toList()
77      }
78
79
80
81      override fun getRandomWord(): Word {
82          if (words.isEmpty()) {
83              throw IllegalStateException("No words available")
84          }
85          return words.random()
86      }
87
88      override fun createUsers(numberOfSpies: Int, numberOfPlayers: Int) {
89          for (i in 1..numberOfSpies) {
90              this.addSpy(createSpy(i))
91
92          }
93          for (i in 1..numberOfPlayers) {
94              this.addPlayer(createPlayer(i))
95          }
96      }
97
98      override fun userdisplaythereRole() {
99          users.forEach { user ->
100             println("${user.username}: ")
101             user.displayRole()
102         }
103     }
104 }
```

Die Klasse InMemoryGameRepository zeigt mehrere Code Smells auf, die auf potenzielle Probleme im Code hinweisen. Einer dieser Code Smells ist die Größe der Klasse, die zu groß erscheint und möglicherweise mehrere Verantwortlichkeiten beinhaltet. Dies verstößt gegen das Single Responsibility Principle (SRP), das besagt, dass eine Klasse nur für eine Aufgabe verantwortlich sein sollte.

Ein weiterer Code Smell ist die Kombination von Datenverwaltung und Geschäftslogik innerhalb derselben Klasse. Die InMemoryGameRepository Klasse ist nicht nur für die Verwaltung von Daten zuständig, sondern enthält auch Logik, die sich auf das Spiel bezieht. Dies führt zu einer unklaren Trennung der Verantwortlichkeiten und kann die Wartbarkeit und Testbarkeit des Codes beeinträchtigen.

Darüber hinaus verstößt das GameRepository Interface gegen das Interface Segregation Principle (ISP). Das ISP besagt, dass Clients nicht von Methoden abhängig sein sollten, die sie nicht verwenden. Da das GameRepository Interface von der InMemoryGameRepository Klasse implementiert wird, die sowohl Datenverwaltung als auch Spiellogik enthält, kann es sein, dass Clients gezwungen sind, Methoden zu implementieren oder zu verwenden, die für sie nicht relevant sind. Dies führt zu einer Interface-Bloat und Verletzung des ISP.

Duplicated Code

Listing 5.2: Code Smell: Duplicated Code

```
override fun loadKlimacticWords(): List<Word> {
    val words = mutableListOf<Word>()
    val geographicZones = arrayOf(
        "Arctic",
        "Antarctic",
        "Tundra",
        "Taiga",
        "Temperate Forest",
        "Tropical Rainforest",
        "Grassland",
        "Desert",
        "Savanna",
        "Mediterranean"
    )

    words.clear() // Clear existing words
    for ((index, zone) in geographicZones.withIndex()) {
        words.add(createWord(index + 1, zone))
    }
    return words
}

override fun loadSportWords(): List<Word> {
    val words = mutableListOf<Word>()
    val sportWords = arrayOf(
        "Fussball",
        "Basketball",
        "Tennis",
        "Volleyball",
        "Schwimmen",
        "Laufen",
        "Leichtathletik",
        "Boxen",
        "Handball",
        "Rugby",
        "Golf",
        "Hockey",
        "Tischtennis",
        "Badminton",
        "Radfahren",
        "Skifahren",
        "Snowboarden",
        "Klettern",
        "Tauchen",
        "Yoga"
    )
    words.clear() // Clear existing words
    for ((index, zone) in sportWords.withIndex()) {
        words.add(createWord(index + 1, zone))
    }
    return words
}
```

In diesem Code gibt es zwei ähnliche Methoden, `loadKlimacticWords` und `loadSportWords`, die beide eine Liste von Wörtern laden und zurückgeben. Beide Methoden haben folgende Gemeinsamkeiten:

1. Sie erstellen eine leere Liste von Wörtern (`words.clear()`), bevor sie neue Wörter hinzufügen.

2. Sie verwenden eine Schleife, um Wörter aus einem Array von Strings hinzuzufügen.
3. Sie verwenden die `createWord`-Methode, um ein Word-Objekt zu erstellen und es der Liste hinzuzufügen.

Obwohl die spezifischen Wortlisten unterschiedlich sind, ist die grundlegende Logik zum Laden der Wörter in beiden Methoden sehr ähnlich. Dies führt zu redundantem Code, da die gleichen oder ähnliche Codefragmente an mehreren Stellen im Code vorhanden sind.

Die Präsenz dieses duplizierten Codes führt zu mehreren Problemen:

- Erhöhter Wartungsaufwand: Änderungen müssen an mehreren Stellen im Code vorgenommen werden, was die Wartung erschwert und die Fehleranfälligkeit erhöht.
- Niedrige Kohäsion: Der Code könnte kohäsiver sein, indem er die gemeinsame Logik an einem Ort konsolidiert.
- Wiederverwendbarkeit: Durch die Konsolidierung des Codes an einer Stelle könnte die Wiederverwendbarkeit verbessert werden, da der Code leichter in anderen Teilen des Systems verwendet werden könnte.

Daher handelt es sich hier um einen Fall von duplicated Code, der vermieden werden sollte, indem die gemeinsame Logik in eine separate Methode oder Funktion extrahiert wird, um die Wiederholung zu vermeiden.

5.2 Refactoring the Code Smell

Interface Segregation Principle

Um den Code Smell Large Class zu beheben wird das Interface Segregation Principle angewendet. In dem die Datenverwaltung auszulagern und in separate Interfaces aufzuteilen: UserRepository, WordRepository und DataRepository. Jedes Interface sollte nur die Methoden enthalten, die für die Verwaltung des entsprechenden Datentyps erforderlich sind5.3.

Listing 5.3: Interface Segragation Principle

```
interface DataRepository {
    fun getAllUsers(): List<User>
    fun getAllWords(): List<Word>
    fun getWord(): Word?
    fun setWord(word: Word)
    fun setUser(createUsers: List<User>)
}


interface UserRepository {
    fun createSpy(spyId: Int): Spy
    fun createPlayer(playerId: Int): Player
    fun createUsers(numberOfSpies: Int, numberOfPlayers: Int): List<User>
    fun displayAllUserRoles(userList:List<User>)
}


interface WordRepository {
    fun loadWords():List<Word>
    fun createWord(wordId: Int, word: String): Word
    fun getRandomWord(words: List<Word>): Word
}
```

Zusätzlich kann die Game-Logik mit Dependency Inversion ausgelagert werden in den Domain Code. Die GameLogic-Klasse kann dann diese abstrakten Repositories verwenden, um die Da-

ten zu verwalten und die Game-Funktionen auszuführen. Dies verbessert die Flexibilität und Testbarkeit des Codes diese wird genauer hier beschrieben 2.2.

Auch wurden Funktion und Code entfernt der nicht genutzt wurdeCommit-Hash.

5.2.1 Extract Method

Listing 5.4: Extract Method

```
fun displayOneRole(){
    if (users.isEmpty()) { return }
    val randomIndex = (0..<users.size).random()
    val randomUser = users.removeAt(randomIndex)

    println("${randomUser.username}:")
    randomUser.displayRole()
    if (randomUser is Player) {
        println("Word: ${dataRepository.getWord()?.name}")  // Print the word if the user is a player and
    }
}
```

Die Methode displayOneRole 5.5 in ihrer aktuellen Form hat eine längere und komplexere Struktur, die sie schwerer zu verstehen und zu warten macht. Hier sind die Gründe, warum dies als Code Smell "Longe Method"betrachtet werden kann:

Umfangreiche Funktionalität: Die Methode führt mehrere Aufgaben aus, darunter die Auswahl eines zufälligen Benutzers aus einer Liste, die Anzeige des Benutzernamens und seiner Rolle sowie die Anzeige des Wortes (falls der Benutzer ein Spieler ist). Diese Vielzahl von Aufgaben erhöht die Länge und Komplexität der Methode.

Hohe Zeilenanzahl: Die Methode umfasst mehrere Zeilen Code, die verschiedene Aspekte der Logik behandeln. Dies kann dazu führen, dass Entwickler mehr Zeit benötigen, um den gesamten Code zu verstehen und potenzielle Fehler zu identifizieren.

Schlechte Lesbarkeit: Eine lange Methode kann schwer zu lesen sein, insbesondere wenn sie viele Schritte oder bedingte Anweisungen enthält. Dies kann die Wartbarkeit des Codes beeinträchtigen und die Fehlerbehebung erschweren.

Um den Code Smell "Longe Methodßu adressieren, könnte die Methode in kleinere, spezifischere Methoden aufgeteilt werden, die jeweils eine einzelne Aufgabe ausführen. Dadurch wird der Code modularer, besser lesbar und einfacher zu warten.

Um den Code der Methode displayOneRole zu verbessern und den Code Smell "Longe Methodßu adressieren, können wir die Methode in kleinere, spezifischere Methoden aufteilen, die jeweils eine einzelne Aufgabe ausführen. Hier ist eine refrakturierte Version unter Verwendung der Methode Ëxtract Method":

Listing 5.5: Extract Method

```
fun displayOneRole() {
    if (users.isEmpty()) { return }

    val randomUser = userRepository.selctRandomUser(users)
    users.remove(randomUser)
    randomUser.displayRole()
    if (randomUser is Player) {
        printPlayerWord(randomUser)
    }
}

private fun printPlayerWord(player: Player) {
    println("Word: ${dataRepository.getWord()?.name}")
}

```

```
16 class ImpUserRepository : UserRepository{
17
18 override fun selctRandomUser(userList: MutableList<User>): User {
19     val randomIndex = (0..<userList.size).random()
20     return userList.removeAt(randomIndex)
21 }
22 }
```

In dieser refaktorierten Version wurde die ursprüngliche Methode displayOneRole in zwei kleinere Methoden aufgeteilt:

selectRandomUser(): Diese Methode wählt einen zufälligen Benutzer aus der Liste aus und entfernt ihn dann aus der Liste.

printPlayerWord(player: Player): Diese Methode druckt das Wort des übergebenen Spielers, falls vorhanden.

Durch diese Aufteilung wird der Code modularer, besser lesbar und einfacher zu warten, wodurch der Code Smell "Longe Method"behoben wirdCommit-Hash.

5.2.2 Template Method

In diesem Commit wird die Code duplication 5.2 mit einer Template Method und mit dem Liskov Substitutionsprinzip gelöst 3.1.

In dem refrakturierten Code ist die Verwendung des Template Method-Musters in der Klasse AbstractWordRepository deutlich erkennbar.

Das Template Method-Method definiert das Grundgerüst eines Algorithmus in einer Superklasse, ermöglicht es jedoch den Unterklassen, bestimmte Schritte des Algorithmus zu überschreiben, ohne dessen Struktur zu ändern.

In diesem Fall:

- Die Klasse AbstractWordRepository definiert eine Template-Methode loadWords(), die den Algorithmus zum Laden von Wörtern repräsentiert.
- Die Methode loadWords() enthält gemeinsames Verhalten, das von allen Unterklassen gemeinsam genutzt wird, wie z. B. das Löschen vorhandener Wörter, das Iterieren über ein Array von Wörtern, das Erstellen von Word-Objekten und das Hinzufügen zu einer Liste.
- Die Methode createWord() wird innerhalb der Template-Methode verwendet, um Word-Objekte zu erstellen und bietet eine Möglichkeit für Unterklassen, den Erstellungsprozess bei Bedarf anzupassen.
- Unterklassen (KlimacticWordRepository und SportWordRepository) erweitern die Klasse AbstractWordRepository und überschreiben die Methode loadWords(), um spezifische Implementierungen zum Laden von klimabezogenen und sportbezogenen Wörtern bereitzustellen.
- Durch die Verwendung des Template Method-Musters wird der Gesamtalgorithmus zum Laden von Wörtern in der Superklasse (AbstractWordRepository) definiert, während Unterklassen bestimmte Schritte anpassen können, ohne die Gesamtstruktur des Algorithmus zu ändern.

Daher macht die Anwesenheit einer Methode (loadWords()) in der Superklasse, die die allgemeine Struktur des Algorithmus definiert und Haken für die Anpassung in den Unterklassen

bereitstellt, das Template Method-Method deutlich erkennbar.

Den Code findet man hier unter Liskov Substitutionsprinzip3.1 oder im Commit-Hash.