UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

Testare unitară în kotlin

Echipă: Năstase Nicolae Marius

Mircea Nae Ștefan

Marcu Ioan

2024

1. Scop Proiectului

Am preluat o aplicație de pe GitHub (un joc consolă) căreia i-am implementat o nouă funcționalitate de verificare a parolei unui utilizator. Scopul acestui proiect este de a dezvolta și implementa o soluție robustă pentru verificarea acestor noi funcționalități aduse. Am aplicat metode de testare structurale detaliate, pentru a evalua fiecare componentă a funcționalității în parte. Utilizând tehnici precum acoperirea la nivel de instrucțiune, ramuri, condiții și decizii, asigurând o analiză în detaliu a codului. Aceste teste sunt esențiale pentru identificarea și corectarea oricăror deficiențe sau vulnerabilități în mecanismul de verificare a parolelor, contribuind astfel la întărirea securității întregului program. Prin integrarea acestor practici riguroase de testare, proiectul ales nu numai că îmbunătățește fiabilitatea soluției dezvoltate, dar și crește încrederea utilizatorilor în protecția datelor lor personale.

2. Resurse folosite

Pentru a asigura calitatea și fiabilitatea proiectului nostru în Kotlin, ne-am bazat pe diverse resurse și metode de testare, una dintre acestea fiind „mutation testing-ul”, un proces recunoscut pentru eficiența sa în identificarea slăbiciunilor unui set de teste. Acest tip de testare implică crearea deliberată a unor versiuni modificate (mutanți) ale codului sursă pentru a verifica capacitatea testelor existente de a identifica și respinge aceste modificări. Utilizarea mutation testing-ului este esențială în contextul în care dorim să evaluăm nu doar prezenta erorilor, ci și robustețea testelor noastre.

2.1 Pitest

Pitest este un instrument esențial în ceea ce privește asigurarea calității testelor în cadrul proiectului nostru. Ca o bibliotecă de testare a mutanților pentru Java și Kotlin, Pitest oferă o metodă puternică de evaluare a setului de teste al unei aplicații. Cum funcționează? Pitest analizează codul sursă al aplicației noastre și generează variante mutate ale acestuia. Apoi, testează acești mutanți, evaluând dacă setul nostru de teste detectează modificările efectuate. Dacă o mutație este detectată de testele noastre, aceasta indică că acoperirea testelor este suficient de cuprinzătoare pentru a captura acea modificare. În schimb, dacă o mutație nu este detectată, acest lucru sugerează că există zone din codul nostru care nu sunt acoperite adecvat de testele noastre. Prin urmare, Pitest oferă o perspectivă valoroasă asupra calității testelor noastre, identificând zonele de îngrijorare și sugerând îmbunătățiri pentru setul de teste.

2.2 Motivul pentru alegerea plug-inului de *pitest*

Alegerea Pitest pentru proiectul nostru a fost justificată de nevoia de a evalua calitatea testelor noastre într-un mod exhaustiv și automatizat. Întrucât aplicația noastră crește în complexitate, devine din ce în ce mai important să ne asigurăm că setul nostru de teste este robust și că acoperă eficient toate scenariile posibile. Pitest ne permite să facem acest lucru, oferind o metodă automatizată de identificare a zonelor din cod care nu sunt testate adecvat. Astfel, Pitest nu numai că ne ajută să identificăm potențiale probleme în testele noastre, dar și ne sugerează modalități de îmbunătățire a acoperirii testelor noastre. Prin urmare, alegerea Pitest a fost esențială pentru asigurarea calității și stabilității aplicației noastre pe termen lung.

2.3 Biblioteci folosite pentru scrierea testelor

În efortul nostru de a asigura o testare exhaustivă și eficientă a codului nostru, am utilizat o serie de biblioteci de testare:

- JUnit 4: JUnit rămâne unul dintre cele mai populare și respectate framework-uri de testare pentru Java. Alegerea JUnit pentru proiectul nostru a fost motivată de popularitatea sa, stabilitatea și ușurința în utilizare;
- MockK: Pentru a crea și gestiona mock-uri în testele noastre, am folosit biblioteca MockK. MockK ne permite să simulăm comportamentul componentelor din aplicația noastră, permițându-ne să izolăm și să testăm unitar diferite părți ale codului nostru;
- Biblioteci de testare kotlin: Deoarece proiectul nostru implică și cod Kotlin, am folosit bibliotecile de testare Kotlin standard pentru a asigura compatibilitatea și funcționalitatea adecvată a testelor noastre scrise în Kotlin.

```
11  dependencies {
12      // Kotlin test library
13      testImplementation(kotlin("test"))
14
15      // junit 4
16      testImplementation("junit:junit:4.13.2")
17
18      // Mockito for Kotlin
19      testImplementation("io.mockk:mockk:1.12.0")
20  }
```

2.4 Utilizare JUnit - motivație

Există mai multe motive pentru care am optat pentru JUnit ca bibliotecă principală de testare:

- Stabilitate și fiabilitate: JUnit este o bibliotecă bine stabilită și fiabilă, utilizată pe scară largă în industria software;

- Ușurință în utilizare: JUnit oferă o sintaxă simplă și intuitivă pentru scrierea și rularea testelor, permițând dezvoltatorilor să se concentreze mai mult pe logica testelor lor decât pe detalii tehnice;
- Suport extins și comunitate activă: JUnit beneficiază de un suport extins și o comunitate activă de dezvoltatori, asigurând actualizări regulate și remedieri rapide ale problemelor.

2.4 Configurarea Pitest

Configurația Pitest în cadrul proiectului nostru a fost realizată în mod meticulos pentru a asigura o evaluare eficientă și relevantă a calității testelor noastre. Iată o prezentare detaliată a configurării specifice a Pitest pentru proiectul nostru:

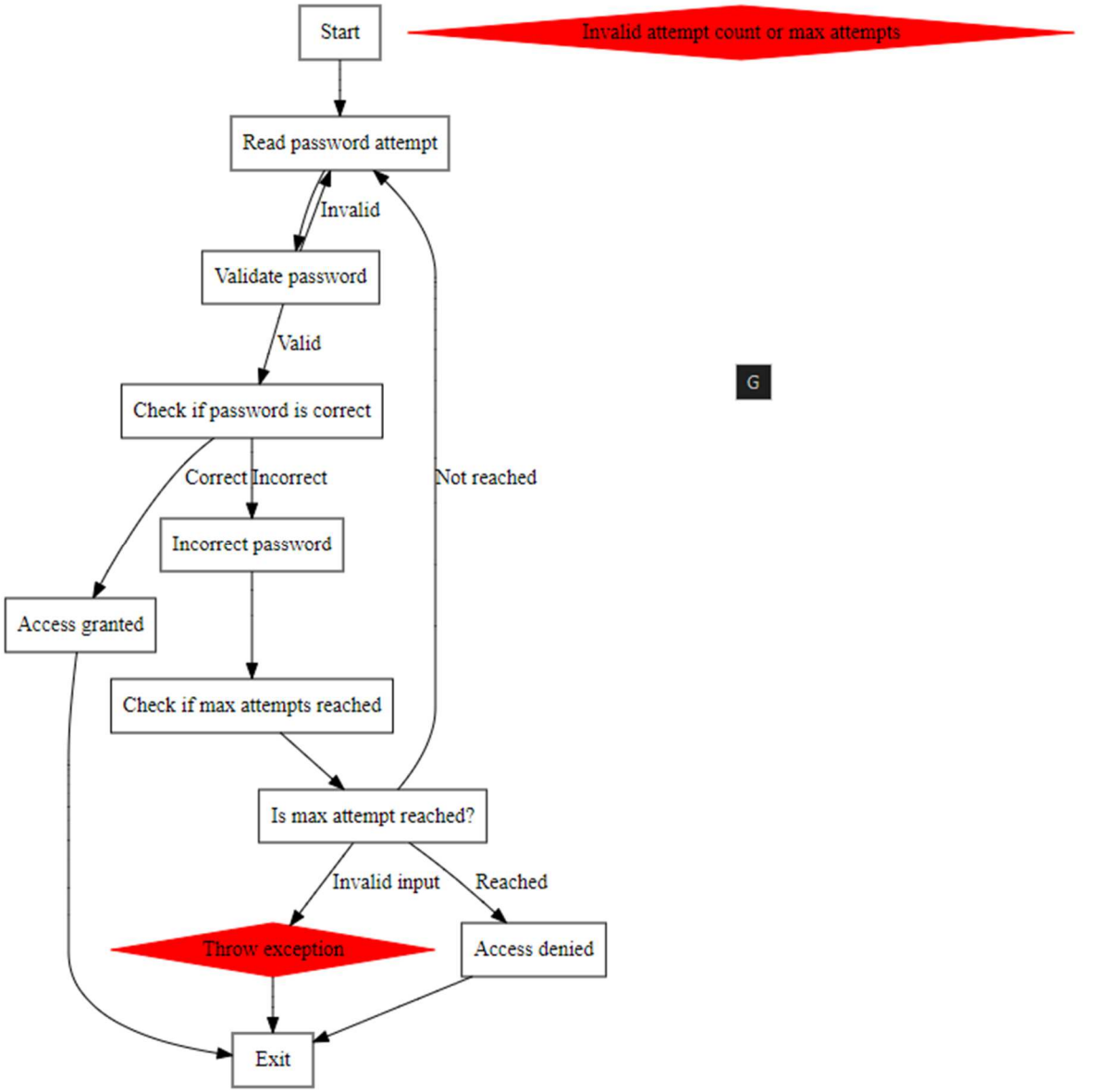
- Clase țintă (Target Classes): Am specificat clasele din proiectul nostru care trebuie mutate și testate pentru a evalua acoperirea testelor;
- Teste țintă (Target Tests): Am definit teste specifice care trebuie rulate împotriva claselor mutate pentru a evalua acoperirea testelor;
- Formatul rapoartelor de ieșire (Output Formats): Am configurat formatele în care vor fi generate rapoartele de ieșire ale testelor mutate, permițând o analiză ușoară și o interpretare adecvată a rezultatelor;
- Limitele de mutare și de acoperire (Mutation and Coverage Thresholds): Am stabilit pragurile de mutație și de acoperire pentru a asigura o evaluare corectă a calității testelor noastre. Această configurație detaliată și personalizată a Pitest este esențială pentru asigurarea unei evaluări adecvate a calității testelor noastre, contribuind la îmbunătățirea generală a stabilității și fiabilității aplicației noastre.

```
33  pitest {
34      targetClasses.set(listOf("model.SecurityProcessor"))
35      targetTests.set(listOf("AutoGeneratedSecurityProcessorTests"))
36      threads.set(4)
37      outputFormats.set(listOf("HTML", "XML"))
38      timestampedReports.set(false)
39      mutationThreshold.set(16)
40      coverageThreshold.set(24)
41      testPlugin.set("junit4")
42  }
```

3. Graful de flux – CFG

Graful de flux de control pentru clasa ”SecurityProcessor” a fost conceput să reprezinte fiecare etapă logică a verificării unei parole prin intermediul unui proces clar definit. Această

structură de graf ajută la vizualizarea fluxului operațional al programului și la identificarea punctelor cheie pentru testare, facilitând astfel o acoperire structurală eficientă a codului. Elementele grafului, cum ar fi nodurile și arcele, corespund deciziilor, condițiilor și posibilelor căi de execuție ale programului, permițând analiza detaliată a comportamentului acestuia în diverse scenarii.



4. Teste unitare în kotlin

Construcția testelor pentru SecurityProcessor a fost ghidată de principiile testării structurale, având la bază diferite nivele de acoperire a codului, cum ar fi acoperirea la nivel de instrucțiune, de ramură (branch), și de condiție. Aceste teste au fost meticulos create pentru a evalua și a valida toate căile posibile de execuție și punctele de decizie din cod, astfel încât

să se asigure detectarea și corectarea eficientă a oricăror defecte sau comportamente neașteptate ale aplicației.

Clasa SecurityProcessor:

```
class SecurityProcessor(private val inputReader: InputReader) {
    fun checkPassword() { val correctPassword = "KotlinSecure123!"
        var attempt: String
        var attemptCount = 0
        val maxAttempts = 3

        println("Please enter your password (you have $maxAttempts attempts, password must include a number, an uppercase letter, and a special character):")

        do {
            attempt = inputReader.readLine() ?: ""
            if (validatePassword(attempt)) {
                if (attempt == correctPassword) {
                    println("Password correct!")
                    return
                } else {
                    println("Incorrect password. Please try again.")
                }
            }
            attemptCount++
            if (attemptCount < maxAttempts) {
                println("${maxAttempts - attemptCount} attempts left.")
            }
        } while (!isMaxAttemptReached(attemptCount, maxAttempts) && attempt != correctPassword)

        if (isMaxAttemptReached(attemptCount, maxAttempts)) {
            println("Maximum attempt limit reached. Access denied.")
        }
    }

    private fun validatePassword(password: String): Boolean {
        if (password.length < 8) {
            println("Password must be at least 8 characters long.")
            return false
        }
        var hasNumber = false
        var hasUppercase = false
```

```
        var hasSpecial = false
        for (ch in password) {
            when {
                ch.isDigit() -> hasNumber = true
                ch.isUpperCase() -> hasUppercase = true
                !ch.isLetterOrDigit() -> hasSpecial = true
            }
        }
        if (!hasNumber || !hasUppercase || !hasSpecial) {
            println("Password must contain at least one number, one uppercase letter, and one special character.")
            return false
        }
        return true
    }

    fun isMaxAttemptReached(attemptCount: Int, maxAttempts: Int): Boolean {
        // Verifică dacă inputurile sunt valide
        if (attemptCount < 0 || maxAttempts <= 0) {
            throw IllegalArgumentException("Attempt count and max attempts must be non-negative and max attempts must be greater than zero.")
        }

        // Verifică dacă numărul de încercări a depășit limita maximă
        return attemptCount >= maxAttempts
    }

}
```

4.1 Acoperirea testelor

i. Acoperirea la nivel de instrucțiune (Statement Coverage)

Scop: Asigură că fiecare linie de cod este executată cel puțin o dată.

- testValidPassword(): Această metodă testează scenariul în care parola introdusă este corectă și corespunde cu parola presetată în sistem. Acest test acoperă cazul de succes al metodei checkPassword, inclusiv verificarea condițiilor de validare prin apelarea validatePassword().
- testInvalidPasswordFormat(): Testează cazul în care parola este prea scurtă, acoperind ramura din validatePassword unde se verifică lungimea parolei.

Aceste teste asigură că porțiuni semnificative din codul metodelor checkPassword și validatePassword sunt executate.

```
33      @Test  Nastase Marius *
34      fun testValidPassword() {
35          every { inputReader.readLine() } returns "KotlinSecure123!"
36          securityProcessor.checkPassword()
37          val output = outputStreamCaptor.toString()
38          assertTrue(output.contains( other: "Password correct!"),  message: "Expected output" +
39                  " to confirm correct password")
40      }
41
42      @Test  new *
43      fun testInvalidPasswordFormat() {
44          every { inputReader.readLine() } returns "short"
45          securityProcessor.checkPassword()
46          val output = outputStreamCaptor.toString()
47          assertTrue(output.contains( other: "Password must be at least 8 characters long"),
48              message: "Expected output to indicate password length error")
49      }
```

ii. Acoperirea la nivel de decizie (Decision/Branch Coverage)

Scop: Toate ramurile deciziilor sunt executate pentru a verifica comportamentele true/false.

- testPasswordWithSpecialCharacters(): Verifică ramura în care parola introdusă este validă din punct de vedere al formatului, dar nu este parola corectă. Acest lucru implică evaluarea condițiilor în care parola este acceptată de validatePassword(), dar respinsă ca nevalidă în checkPassword().
- testPasswordWithNoSpecialCharacter(): Acest test verifică ramura unde parola nu îndeplinește toate criteriile de validare, verificând condițiile în care parolei îi lipsește un caracter special.

```
50
51      @Test  Nastase Marius *
52      fun testPasswordWithSpecialCharacters() {
53          every { inputReader.readLine() } returns "Abcde#123"
54          securityProcessor.checkPassword()
55          val output = outputStreamCaptor.toString()
56          assertTrue(output.contains( other: "Incorrect password. Please try again."),
57              message: "Expected output for incorrect password with valid format")
58      }
59
60      @Test  Nastase Marius *
61      fun testPasswordWithNoSpecialCharacter() {
62          every { inputReader.readLine() } returns "Abcde1234"
63          securityProcessor.checkPassword()
64          val output = outputStreamCaptor.toString()
65          assertTrue(output.contains( other: "Password must contain at least one number, " +
66                  "one uppercase letter, and one special character."),
67              message: "Expected specific validation error for missing special character")
68      }
69
```

iii. Acoperirea la nivel de condiție (Condition Coverage)

Scop: Toate condițiile evaluative dintr-o decizie sunt verificate pentru valorile true și false.

- testPasswordWithNoSpecialCharacter și testInvalidPasswordFormat: Aceste teste verifică condițiile individuale din validatePassword (lungime insuficientă, lipsa unui caracter special). Acestea asigură că fiecare condiție din validatePassword este testată atât pentru true, cât și pentru false.

iv. Acoperirea modificată la nivel de condiție/decizie (MC/DC Coverage)

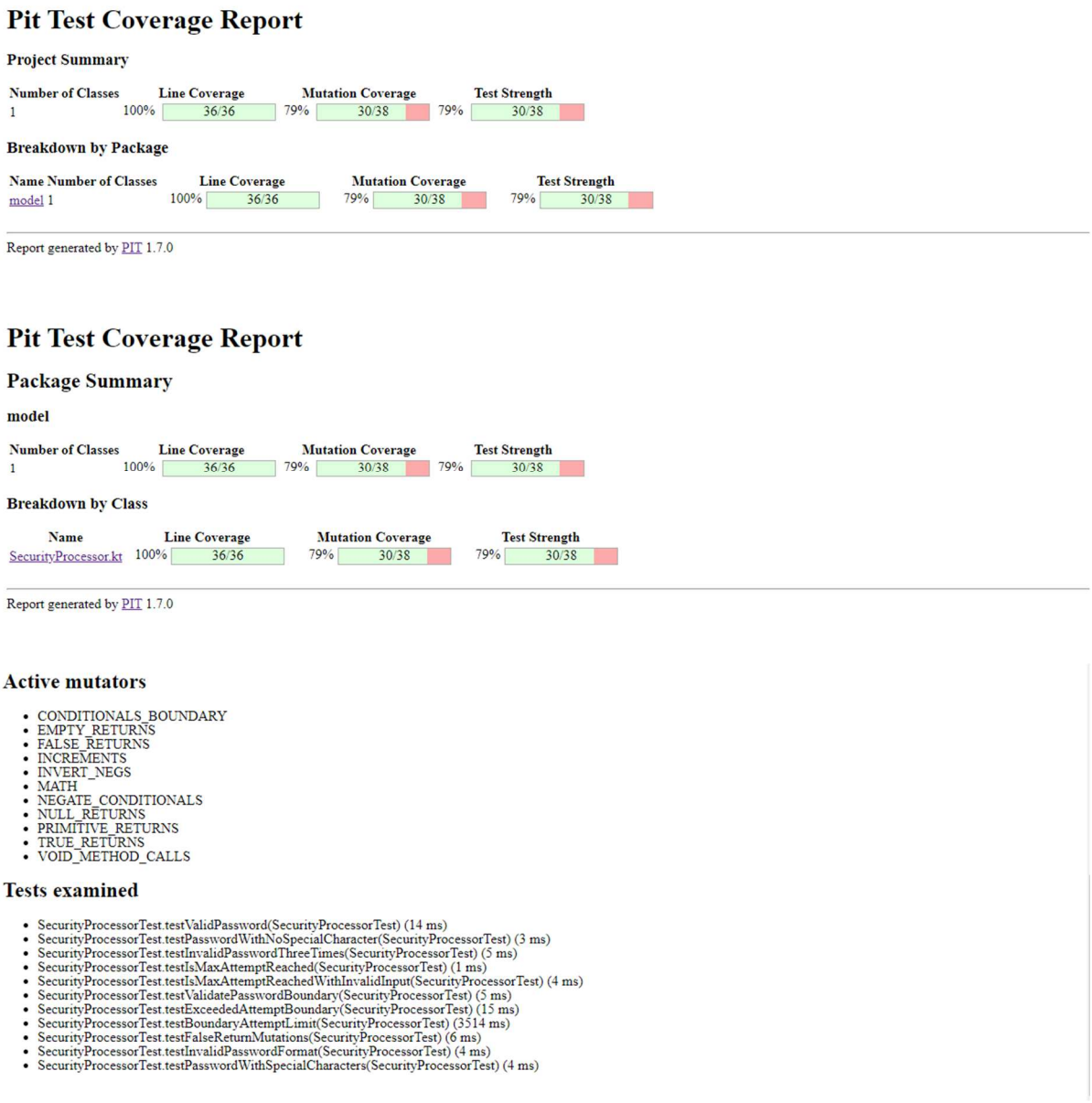
Scop: Fiecare condiție într-o decizie trebuie să determine independent decizia.

- testIsMaxAttemptReachedWithInvalidInput: Acest test explorează isMaxAttemptReached verificând diferite combinații de intrări care ar trebui să arunce excepții, testând astfel condițiile de verificare a limitelor în mod independent.

```
111     @Test  Nastase Marius
112     fun testIsMaxAttemptReachedWithInvalidInput() {
113         assertFailsWith<IllegalArgumentException> {
114             securityProcessor.isMaxAttemptReached( attemptCount: -1,  maxAttempts: 3)
115         }
116         assertFailsWith<IllegalArgumentException> {
117             securityProcessor.isMaxAttemptReached( attemptCount: 1,  maxAttempts: 0)
118         }
119     }
```

Prin aplicarea acestor tehnici de testare structurală, am reușit să construim un set de teste care nu doar că verifică funcționalitățile de bază ale clasei SecurityProcessor, dar și identifică și gestionează cazurile limită și excepțiile, asigurând astfel robustețea și fiabilitatea soluției noastre. Această abordare detaliată facilitează menținerea și scalarea aplicației, permițând adaptări și îmbunătățiri continue pe măsură ce noi cerințe sau scenarii sunt luate în considerare.

4.2 Raport teste



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
1	100% 36/36	79% 30/38	79% 30/38

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
model	1	100% 36/36	79% 30/38	79% 30/38

Report generated by PIT 1.7.0

Pit Test Coverage Report

Package Summary

model

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
1	100% 36/36	79% 30/38	79% 30/38

Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
SecurityProcessor.kt	100% 36/36	79% 30/38	79% 30/38

Report generated by PIT 1.7.0

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

- SecurityProcessorTest.testValidPassword(SecurityProcessorTest) (14 ms)
- SecurityProcessorTest.testPasswordWithNoSpecialCharacter(SecurityProcessorTest) (3 ms)
- SecurityProcessorTest.testInvalidPasswordThreeTimes(SecurityProcessorTest) (5 ms)
- SecurityProcessorTest.testIsMaxAttemptReached(SecurityProcessorTest) (1 ms)
- SecurityProcessorTest.testIsMaxAttemptReachedWithInvalidInput(SecurityProcessorTest) (4 ms)
- SecurityProcessorTest.testValidatePasswordBoundary(SecurityProcessorTest) (5 ms)
- SecurityProcessorTest.testExceededAttemptBoundary(SecurityProcessorTest) (15 ms)
- SecurityProcessorTest.testBoundaryAttemptLimit(SecurityProcessorTest) (3514 ms)
- SecurityProcessorTest.testFalseReturnMutations(SecurityProcessorTest) (6 ms)
- SecurityProcessorTest.testInvalidPasswordFormat(SecurityProcessorTest) (4 ms)
- SecurityProcessorTest.testPasswordWithSpecialCharacters(SecurityProcessorTest) (4 ms)

5. Teste generate cu IA (ChatGBT)

În această parte a proiectului am folosit inteligența artificială pentru a genera teste unitare pentru aceeași clasă. Testele generate sunt:

```kotlin
class AutoGeneratedSecurityProcessorTests {  new *
    private lateinit var inputReader: InputReader
    private lateinit var securityProcessor: SecurityProcessor

    @Before  new *
    fun setUp() {
        inputReader = mockk(relaxed = true) //
        securityProcessor = SecurityProcessor(inputReader)
    }

    @Test  new *
    fun testInputWithValidPassword() {
        every { inputReader.readLine() } returns "KotlinSecure123!"
        securityProcessor.checkPassword()
        verify(exactly = 1) { inputReader.readLine() }
        // Verificăm că metoda de verificare a fost apelată corect.
    }

    @Test  new *
    fun testInputWithInvalidPasswordFormat() {
        every { inputReader.readLine() } returns "short"
        securityProcessor.checkPassword()
        verify(exactly = 3) { inputReader.readLine() }
        // Testăm logica de maxim trei încercări.
    }

    @Test  new *
    fun testInputWithMaxAttemptLimitReached() {
        every { inputReader.readLine() } returnsMany listOf("wrongPass1", "wrongPass2", "wrongPass3")
        securityProcessor.checkPassword()
        verify(exactly = 3) { inputReader.readLine() }
        // Verificăm că metoda este apelată de trei ori.
    }
}
```

5.1 Raport teste generate de IA

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	94%	34/36	29%	11/38	30%	11/37

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
model	1	94%	34/36	29%	11/38	30%	11/37

Report generated by PIT 1.7.0

Pit Test Coverage Report

Package Summary

model

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	94%	34/36	29%	11/38	30%	11/37

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
SecurityProcessor.kt	94%	34/36	29%	11/38	30%	11/37

Report generated by PIT 1.7.0

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

- AutoGeneratedSecurityProcessorTests.testInputWithMaxAttemptLimitReached(AutoGeneratedSecurityProcessorTests) (7 ms)
- AutoGeneratedSecurityProcessorTests.testInputWithInvalidPasswordFormat(AutoGeneratedSecurityProcessorTests) (2110 ms)
- AutoGeneratedSecurityProcessorTests.testInputWithValidPassword(AutoGeneratedSecurityProcessorTests) (3 ms)

Report generated by PIT 1.7.0

6. Diferențe

Diferențele dintre testele generate automat de AI și cele scrise manual oferă perspective valoroase asupra abordărilor de testare și asupra modului în care acestea pot fi optimizate pentru a acoperi diferite scenarii de utilizare ale codului în mod eficient.

- *Specificații și detalii*: testele scrise manual, tind să fie mai detaliate și să includă validări specifice pentru output-uri și comportamente ale sistemului. Aceste teste validează nu doar faptul că codul rulează, ci și că output-urile sunt corecte în diverse condiții, reflectând o înțelegere profundă a cerințelor sistemului. De exemplu, testele manuale verifică explicit mesajele afișate utilizatorului în diferite scenarii de eroare, cum ar fi introducerea unei parole prea scurte sau lipsa unui caracter special.
- *Acoperirea scenariilor de eșec și manipularea excepțiilor*: testele generate automat se concentrează adesea pe execuția funcțiilor cu valori anticipate, verificând apelurile și interacțiunile dintre componente, cum ar fi numărul de ori în care se apelează o funcție de citire a parolei. Deși aceste teste sunt utile pentru a confirma că sistemul se comportă conform așteptărilor în condiții normale, ele pot neglija testarea adecvată a gestionării erorilor sau a scenariilor de eșec. De exemplu, un test automat poate verifica că readLine() este apelat de trei ori atunci când parolele sunt incorecte, dar poate să nu verifice mesajele specifice afișate utilizatorului sau comportamentele de sistem în urma erorilor.
- *Verificarea interacțiunilor și a stărilor internale*: în testele manuale, este frecvent să se verifice starea internă a sistemului sau a obiectelor pentru a asigura că starea internă corespunde așteptărilor după diverse interacțiuni. Testele generate de AI, se concentrează mai mult pe interacțiunile dintre componentele mock-uite și metodele apelate, cum ar fi verificarea numărului de apeluri la readLine(). Acest stil de testare este util pentru validarea fluxurilor de lucru, dar poate omite verificările detaliate ale stării interne care sunt critice pentru a asigura corectitudinea logică în profunzime.

- *Flexibilitate și Adaptabilitate*: testele manuale pot fi ajustate pentru a reflecta schimbările în logica de business sau cerințele utilizatorilor, permitând testatori să adauge scenarii de test complexe și specifice domeniului de aplicare. Pe de altă parte, testele generate de AI se bazează pe algoritmi care urmăresc șabloane și pot să nu adapteze testele la subtilitățile cerințelor noi sau modificate fără intervenția umană.

Bibliography

https://github.com/Smoothie1-ini/Fighthalla

https://github.com/Smoothie1-ini/Fighthalla

https://pitest.org/

https://chat.openai.com/

https://mockk.io/