31243
Paulina Hudzik, IS(s) grupa P1
20.11.2024
NSI4

Testy oprogramowania – manualne

Testy manualne stanowią kluczowy etap w procesie weryfikacji oprogramowania, pozwalając na ocenę jego działania z perspektywy użytkownika końcowego oraz analizę pod kątem kluczowych wymagań jakościowych. Ich głównym celem jest identyfikacja błędów, ocena stabilności oraz zapewnienie, że aplikacja spełnia oczekiwania użytkowników i biznesowe założenia projektu.

Kategorie testów i ich znaczenie:

1. **Niezawodność**

Testy niezawodności sprawdzają, czy aplikacja działa poprawnie w różnych warunkach, zarówno standardowych, jak i wyjątkowych. Ich celem jest zapewnienie, że oprogramowanie nie zawiedzie w krytycznych momentach.

2. **Efektywność**

Testy efektywności oceniają, jak szybko aplikacja wykonuje operacje, minimalizując czas przetwarzania. Pozwala to zapewnić płynność działania i komfort użytkowania.

3. **Bezpieczeństwo**

Testowanie bezpieczeństwa polega na analizie aplikacji pod kątem ryzyka i potencjalnych zagrożeń, takich jak błędna obsługa danych wejściowych czy nieautoryzowany dostęp. Celem jest ochrona użytkowników i danych.

4. **Jakość**

Jakość jest mierzona zgodnością z najlepszymi praktykami programistycznymi. Testy tej kategorii oceniają strukturalną poprawność kodu i jego przejrzystość, co ułatwia późniejsze utrzymanie aplikacji.

5. **Użyteczność**

Testy użyteczności skupiają się na ocenie, czy aplikacja jest intuicyjna i łatwa w obsłudze. Mają kluczowe znaczenie dla poprawy doświadczenia użytkownika końcowego.

6. **Pielęgnowalność**

Pielęgnowalność odnosi się do łatwości wprowadzania zmian w kodzie. Testy te zapewniają, że oprogramowanie można rozwijać i modyfikować bez ryzyka destabilizacji działania.

7. **Wydajność**

Testy wydajności mają na celu ocenę, czy aplikacja działa optymalnie pod dużym obciążeniem, np. podczas wykonywania złożonych obliczeń.

8. **Przeciążenia i obciążenia**

Testy te badają zachowanie aplikacji w warunkach ekstremalnych, takich jak intensywne użytkowanie lub równoczesne operacje wielowątkowe. Ich celem jest identyfikacja ograniczeń i punktów krytycznych aplikacji.

Dlaczego stosujemy testy manualne?

Testy manualne pozwalają na:

- Wykrycie błędów trudnych do przewidzenia za pomocą automatyzacji.
- Symulację rzeczywistych warunków użytkowania.
- Oceny jakości z perspektywy użytkownika.
- Zbadanie aspektów, których nie można łatwo zautomatyzować (np. subiektywnej oceny użyteczności).

Testy manualne są niezbędnym uzupełnieniem testów automatycznych, szczególnie w projektach, gdzie ważna jest jakość, wydajność i doświadczenie użytkownika.

O testowany programie

Testowana aplikacja została stworzona w ramach przedmiotu "Kryptografia i analiza kodów" jako projekt mający na celu przedstawienie praktycznego zastosowania różnych technik kryptograficznych. Aplikacja pozwala na szyfrowanie i deszyfrowanie tekstów przy użyciu zarówno klasycznych, jak i nowoczesnych algorytmów kryptograficznych. Aplikacja została wyposażona w graficzny interfejs użytkownika wykonany w PyQt5, który umożliwia wprowadzanie danych, konfigurację kluczy szyfrowania oraz przeglądanie wyników.

Główne funkcjonalności aplikacji obejmują:

- Klasyczne szyfry:
  - Szyfr transpozycyjny,
  - Szyfr monoalfabetyczny,
- Szyfrowanie symetryczne:
  - AES (Advanced Encryption Standard),
  - DES (Data Encryption Standard),
- Szyfrowanie asymetryczne:
  - RSA (Rivest-Shamir-Adleman),
- Protokół wymiany kluczy:
  - Diffie-Hellman.

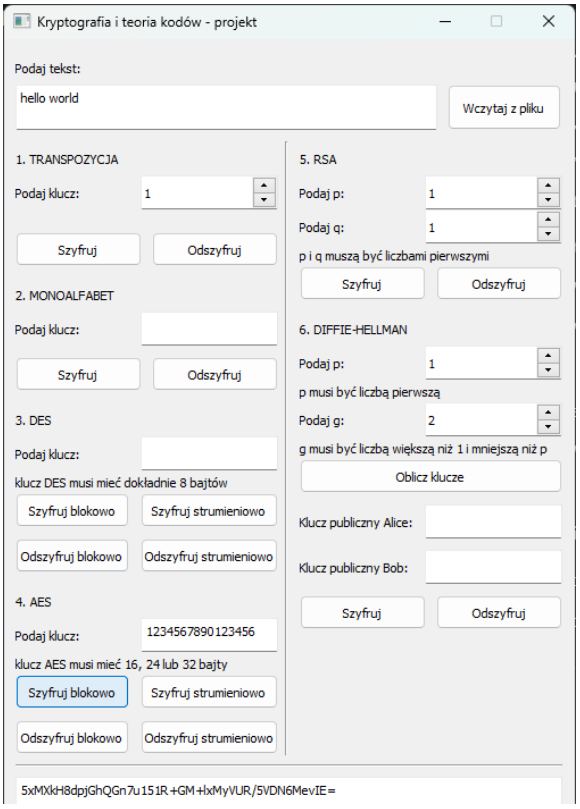
1. Test niezawodności

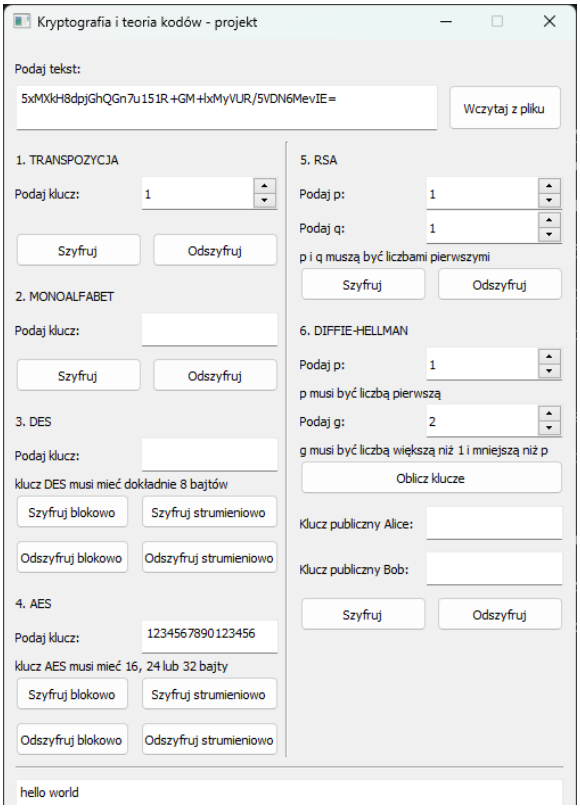
Test 1: Szyfrowanie tekstu 'hello world' algorytmem AES w trybie blokowym
Cel: Zweryfikowanie, czy algorytm AES poprawnie szyfruje i odszyfrowuje tekst w trybie blokowym.
Kroki testowe:

- W polu tekstowym wprowadź tekst: hello world.
- W polu klucza wprowadź klucz: 1234567890123456.
- Kliknij przycisk "Szyfruj blokowo".
- Wprowadź zaszyfrowany tekst do pola wejściowego.
- Kliknij przycisk "Odszyfruj blokowo".

Oczekiwany wynik:

- Po szyfrowaniu w polu wyjścia pojawi się zaszyfrowany tekst.
- Po odszyfrowaniu w polu wyjścia zostanie wyświetlony oryginalny tekst.



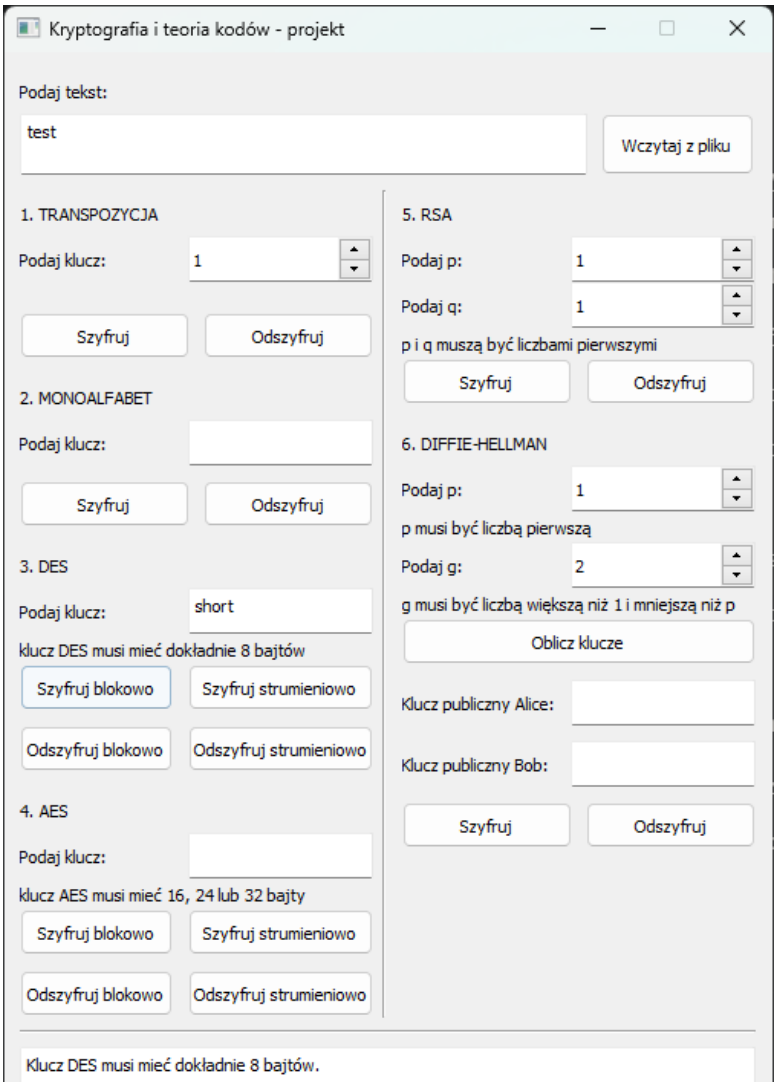


Wynik testu: Udany. Aplikacja poprawnie zaszyfrowała i odszyfrowała tekst w trybie blokowym AES. Wynik odszyfrowania był identyczny z tekstem pierwotnym. Test potwierdził niezawodność algorytmu w standardowych warunkach.

Test 2: Próba szyfrowania z niepoprawnym kluczem dla DES
Cel: Sprawdzenie, czy aplikacja poprawnie obsługuje nieprawidłowy klucz DES.
Kroki testowe:

- W polu tekstowym wprowadź tekst: hello world.
- W polu klucza wprowadź klucz: short (mniej niż 8 znaków).
- Kliknij przycisk "Szyfruj blokowo" (DES).

Oczekiwany wynik: W polu wyjścia pojawi się odpowiedni komunikat o błędnym kluczu.



Wynik testu: Udany. Aplikacja poprawnie wykryła nieprawidłowy klucz o długości 5 znaków i zablokowała operację szyfrowania. Komunikat błędu był czytelny i zgodny z oczekiwaniami. Test potwierdził niezawodność mechanizmu walidacji kluczy dla algorytmu DES.

Testy niezawodności potwierdziły, że aplikacja jest w stanie prawidłowo realizować swoje podstawowe funkcje szyfrowania i odszyfrowywania danych, a także reagować na błędy użytkownika. Jest to kluczowe dla aplikacji kryptograficznej, ponieważ:

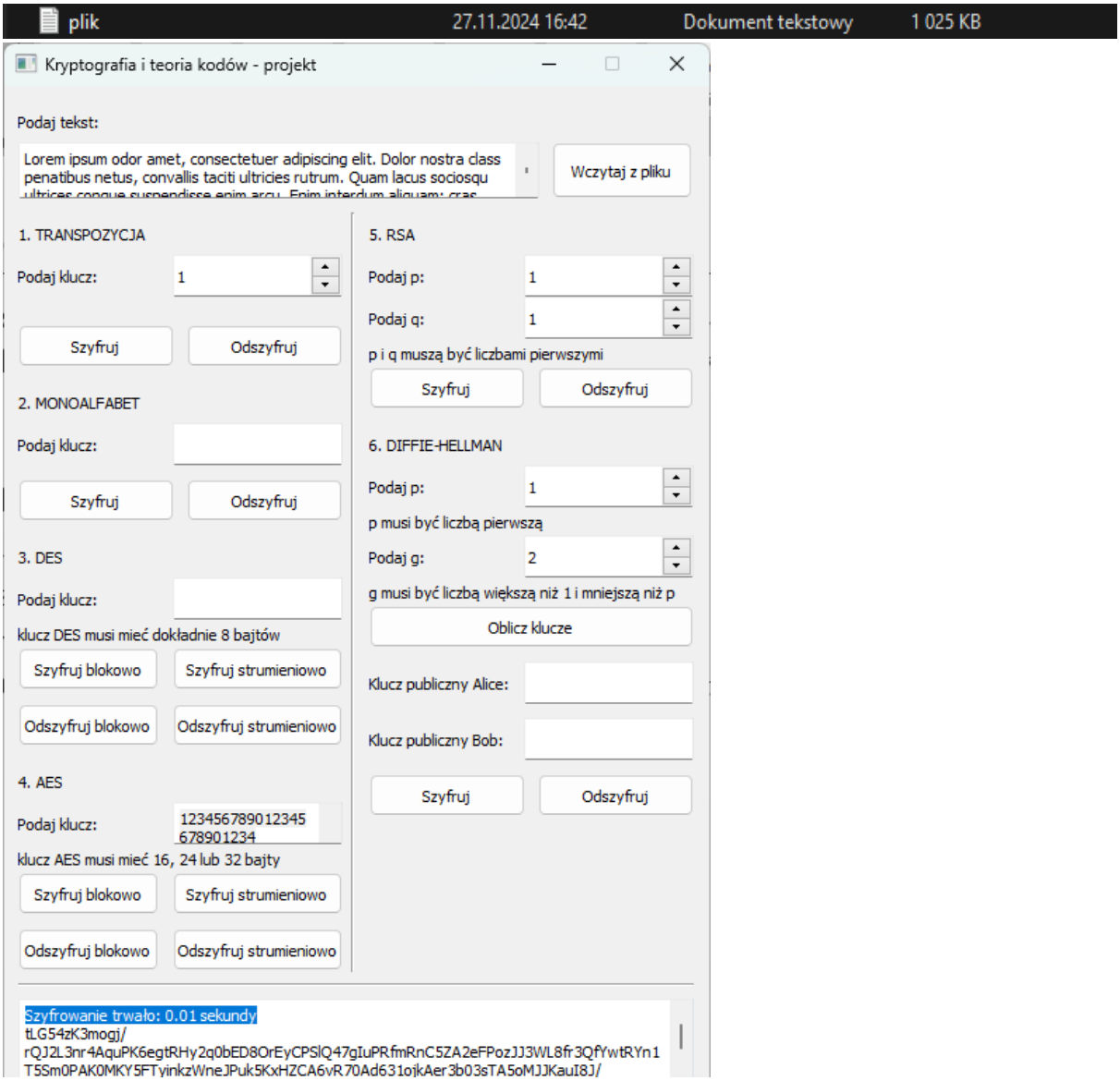
- Bezpieczeństwo danych: Niezawodność operacji szyfrowania gwarantuje, że dane użytkownika są chronione przed utratą integralności.
- Stabilność: Aplikacja działała poprawnie i stabilnie zarówno w warunkach prawidłowych, jak i w przypadku błędnych danych wejściowych.
- Użyteczność: Jasne komunikaty o błędach zwiększają komfort użytkownika, pomagając mu zrozumieć wymagania dotyczące danych wejściowych.

2. Test efektywność

Test 3: Pomiar czasu szyfrowania tekstu o długości 1 MB algorytmem AES
Cel: Sprawdzenie efektywności algorytmu AES przy pracy z dużymi danymi.
Kroki testowe:

- Przygotuj plik tekstowy o długości 1 MB.
- Wczytaj plik do aplikacji za pomocą funkcji "Wczytaj z pliku".
- W polu klucza wprowadź klucz: 12345678901234567890123 4.
- Kliknij przycisk "Szyfruj blokowo".
- Zmierz czas trwania operacji.

Oczekiwany wynik: Czas szyfrowania nie przekracza 1 sekundy dla tekstu.



Wynik testu: Udany. Algorytm AES wykazuje bardzo wysoką efektywność i stabilność przy przetwarzaniu danych o dużych rozmiarach. Wynik (0.01 sekundy) pokazuje, że aplikacja jest dobrze zoptymalizowana i gotowa do pracy z dużymi plikami w środowisku produkcyjnym.

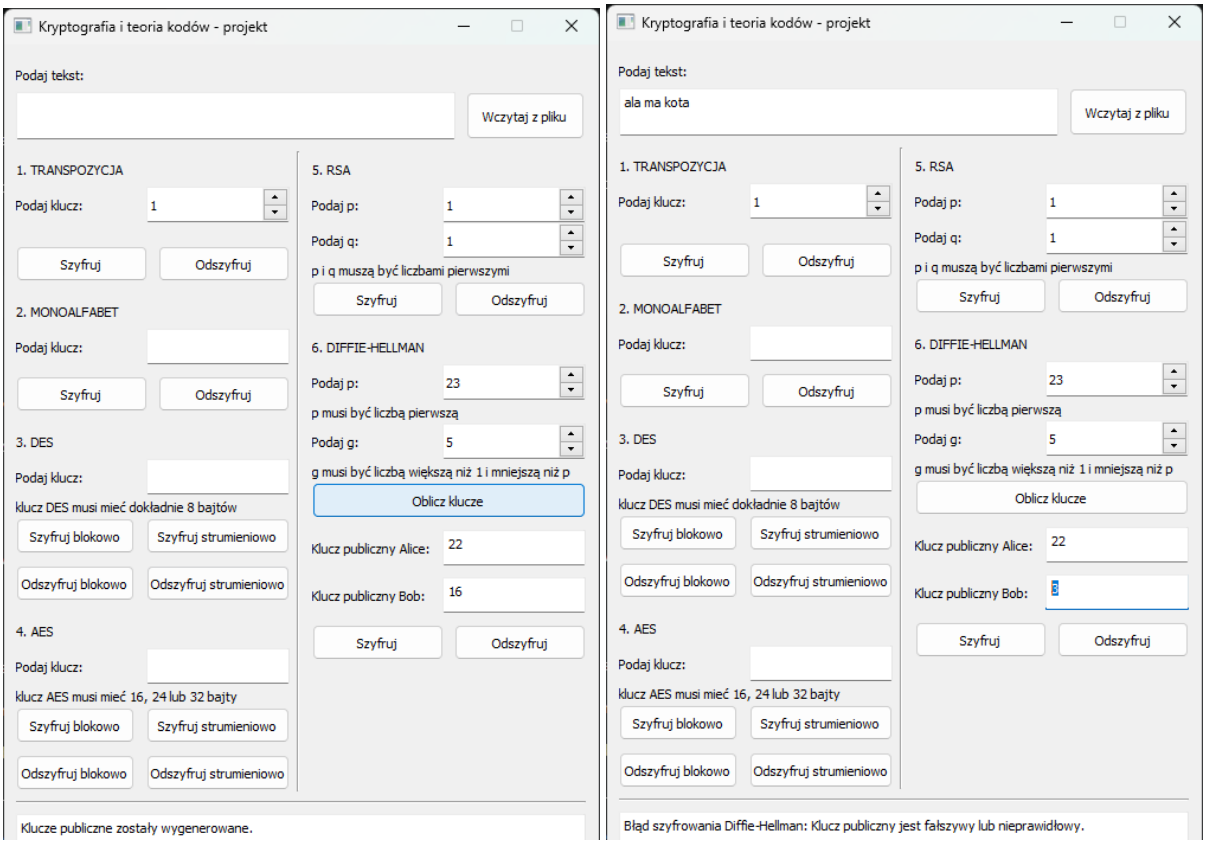
Test efektywności wykazał, że aplikacja spełnia wymagania wydajnościowe. Jest to kluczowe dla aplikacji kryptograficznej, ponieważ zapewnia możliwość szyfrowania dużych danych w czasie rzeczywistym, co jest istotne w zastosowaniach praktycznych, takich jak przesyłanie dużych plików czy przetwarzanie danych w systemach produkcyjnych.

3. Test bezpieczeństwa

Test 4: Próba deszyfrowania Diffie-Hellman z fałszywym kluczem publicznym
Cel: Weryfikacja, czy aplikacja obsługuje nieprawidłowe klucze w protokole Diffie-Hellman.
Kroki testowe:

- Wygeneruj klucz publiczny Alice i Bob za pomocą aplikacji.
- Ręcznie zmodyfikuj klucz publiczny Bob na fałszywy.
- Spróbuj odszyfrować wiadomość za pomocą Diffie-Hellman.

Oczekiwany wynik: W polu wyjścia pojawi się komunikat błędu.



Wynik testu: Udany. Aplikacja poprawnie rozpoznała fałszywy klucz publiczny w protokole Diffie-Hellman i uniemożliwiła operację odszyfrowywania. Test potwierdził, że mechanizmy walidacji danych w aplikacji są poprawnie zaimplementowane i wspierają bezpieczeństwo aplikacji kryptograficznej.

Test ten ma kluczowe znaczenie dla bezpieczeństwa aplikacji. Pokazał, że aplikacja jest odporna na błędy w kluczach publicznych i potrafi odpowiednio reagować na sytuacje, w których używane są nieprawidłowe dane.

4. Test jakości

1. Zgodność z najlepszymi praktykami programistycznymi:
Zalety:

- Kod aplikacji jest modularny – każda funkcjonalność (algorytm szyfrowania) została zaimplementowana w osobnej klasie, co jest zgodne z zasadami Single Responsibility Principle (SRP).
- Walidacja danych wejściowych (np. długości kluczy dla DES i AES) jest obecna, co świadczy o dbałości o poprawność danych.
- Czytelne komentarze i opisy funkcji (docstringi) pomagają w zrozumieniu działania kodu.

Wady:

- Kod GUI zawiera duplikaty funkcji, co narusza zasadę DRY (Don't Repeat Yourself).
- Brak interfejsu lub klasy bazowej dla wspólnych funkcji szyfrowania i odszyfrowywania prowadzi do zwiększonego nakładu pracy w przypadku wprowadzania zmian.

2. Strukturalna poprawność kodu:
Zalety:

- Każdy algorytm szyfrowania ma swoją własną implementację w osobnym module, co zwiększa czytelność i ułatwia utrzymanie.
- Klasy są dobrze nazwane i odpowiadają swojemu przeznaczeniu (np. AESCipher, DESCipher), co ułatwia ich zrozumienie i ponowne użycie.

Wady:

- Brakuje wspólnej struktury lub interfejsu dla wszystkich algorytmów, co utrudnia standaryzację i reużywalność kodu w GUI.

3. Przejrzystość kodu i łatwość utrzymania:
Zalety:

- Kod jest dobrze skomentowany, a docstringi szczegółowo opisują działanie funkcji, co ułatwia zrozumienie i rozwój aplikacji przez nowych programistów.
- Aplikacja została zaprojektowana w sposób modularny, co ułatwia dodawanie nowych funkcji (np. ROT13 został bezproblemowo zintegrowany).

Wady:

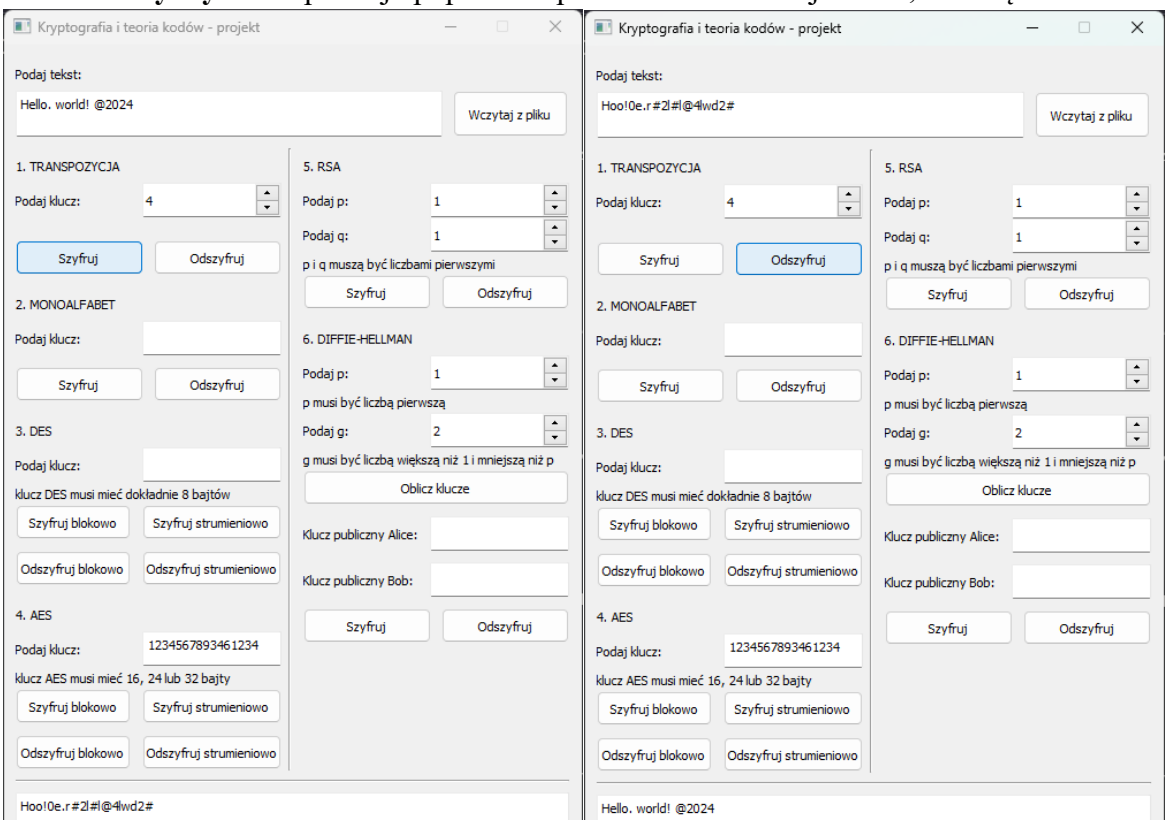
- Duplikaty kodu w GUI (np. osobne funkcje szyfrowania i odszyfrowywania dla każdego algorytmu) utrudniają utrzymanie i zwiększają ryzyko błędów podczas przyszłych zmian.
- Brakuje mechanizmu centralnego logowania błędów, co mogłoby pomóc w diagnozowaniu problemów i utrzymaniu jakości.

5. Test użyteczność

Test 5: Test wprowadzania danych do GUI
Cel: Sprawdzenie, czy interfejs użytkownika umożliwia poprawne wprowadzanie danych.
Kroki testowe:

- Wprowadź tekst zawierający znaki specjalne: Hello, world! @2024.
- Wprowadź klucz o mieszanej wielkości liter: MyKey2024.
- Kliknij "Szyfruj" dla dowolnego algorytmu.

Oczekiwany wynik: Aplikacja poprawnie przetwarza dane wejściowe, bez błędów.



Wynik testu: Udany. Aplikacja poprawnie obsługuje różnorodne dane wejściowe, co potwierdza jej użyteczność i niezawodność w typowych scenariuszach użytkowania. Interfejs użytkownika jest intuicyjny i wspiera płynne wprowadzanie oraz przetwarzanie danych, zwiększając komfort użytkowania.

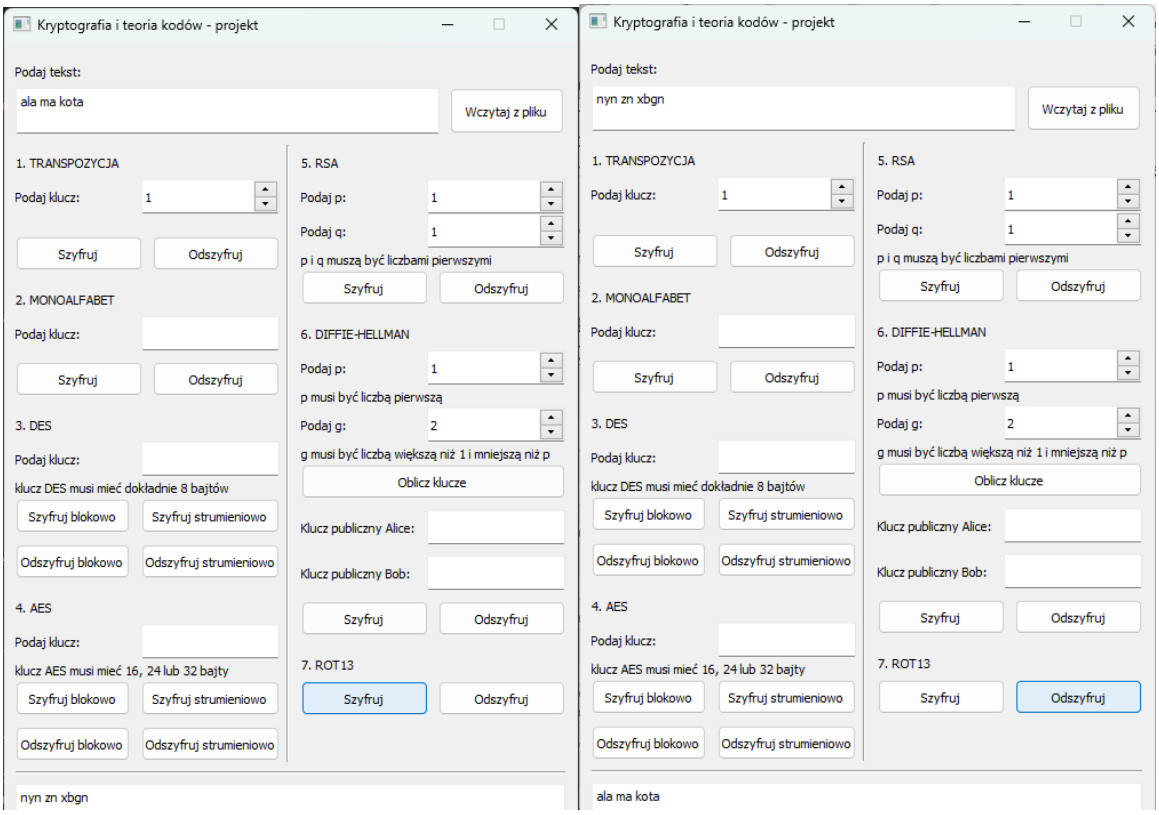
Test potwierdził, że aplikacja jest użyteczna i dobrze przystosowana do pracy z różnorodnymi danymi wejściowymi, co zwiększa jej praktyczne zastosowanie. Umiejętność poprawnego przetwarzania tekstów zawierających znaki specjalne i różne formaty kluczy świadczy o wysokiej jakości implementacji.

6. Test pielęgnowalności

Test 6: Weryfikacja pielęgnowalności kodu przy dodawaniu nowej funkcji.
Cel: Sprawdzenie, czy interfejs użytkownika umożliwia poprawne wprowadzanie danych.
Kroki testowe:

- Dodaj do aplikacji szyfr ROT13 jako nową funkcję.
- Skonfiguruj GUI, aby umożliwiało wybór szyfru ROT13.
- Wprowadź tekst: hello.
- Kliknij przycisk szyfrowania.

Oczekiwany wynik: Tekst hello zostaje zaszyfrowany jako uryyb.



Dodanie nowego pliku rot13.py:

```
def szyfruj(tekst):
    return tekst.translate(str.maketrans(
        "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz",
        "NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm"
    ))

def odszyfruj(tekst):
    return szyfruj(tekst)
```

Dodane metody do pliku main.py:

```
    def szyfruj_rot13(self):
        """
        Szyfruje tekst za pomocą algorytmu ROT13.
        """
        tekst = self.ui.input.toPlainText()
        if not self.sprawdz_puste_pole(tekst):
            return
```

```
        szyfrowany = rot13_szyfruj(tekst)
        self.ui.output.setPlainText(szyfrowany)

    def odszyfruj_rot13(self):
        """
        Odszyfrowuje tekst zaszyfrowany za pomocą algorytmu ROT13.
        """
        tekst = self.ui.input.toPlainText()
        if not self.sprawdz_puste_pole(tekst):
            return
        odszyfrowany = rot13_odszyfruj(tekst)
        self.ui.output.setPlainText(odszyfrowany)
```

Wynik testu: Udany. Algorytm ROT13 został skutecznie dodany do aplikacji, a funkcjonalność szyfrowania działa zgodnie z założeniami. Test potwierdza, że aplikacja jest modularna i łatwa do pielęgnacji, co pozwala na przyszłą rozbudowę o kolejne funkcje i algorytmy.

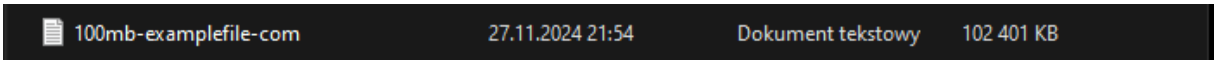
Test potwierdził, że aplikacja jest łatwa w pielęgnacji i rozbudowie, co jest kluczowe dla jej dalszego rozwoju. Dzięki modularnej strukturze kodu, dodanie nowych funkcji (takich jak szyfr ROT13) nie wymagało znaczących modyfikacji istniejących komponentów. To zwiększa elastyczność aplikacji i ułatwia jej skalowanie w przyszłości.

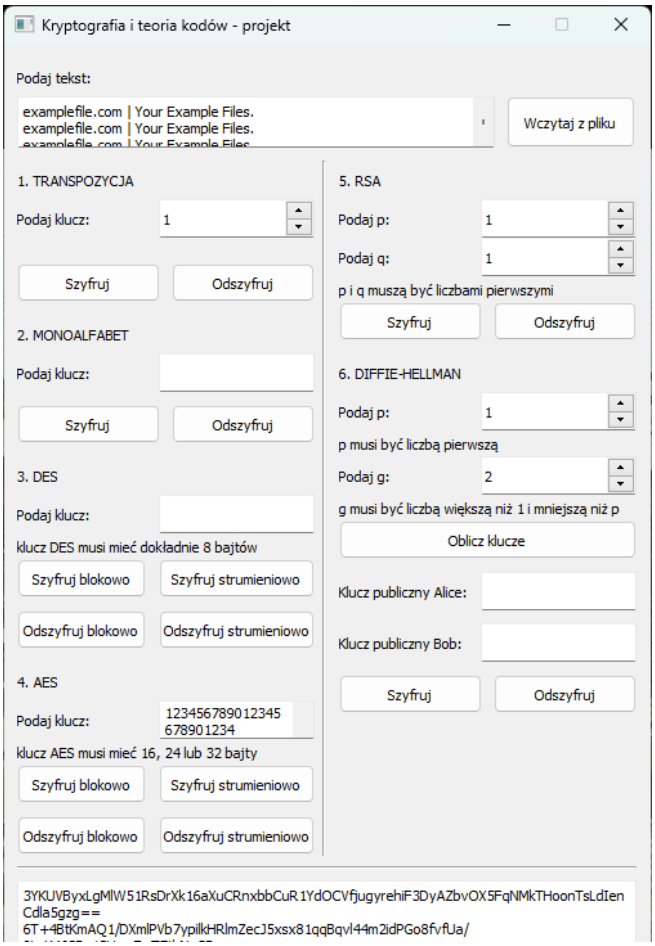
7. Test wydajności

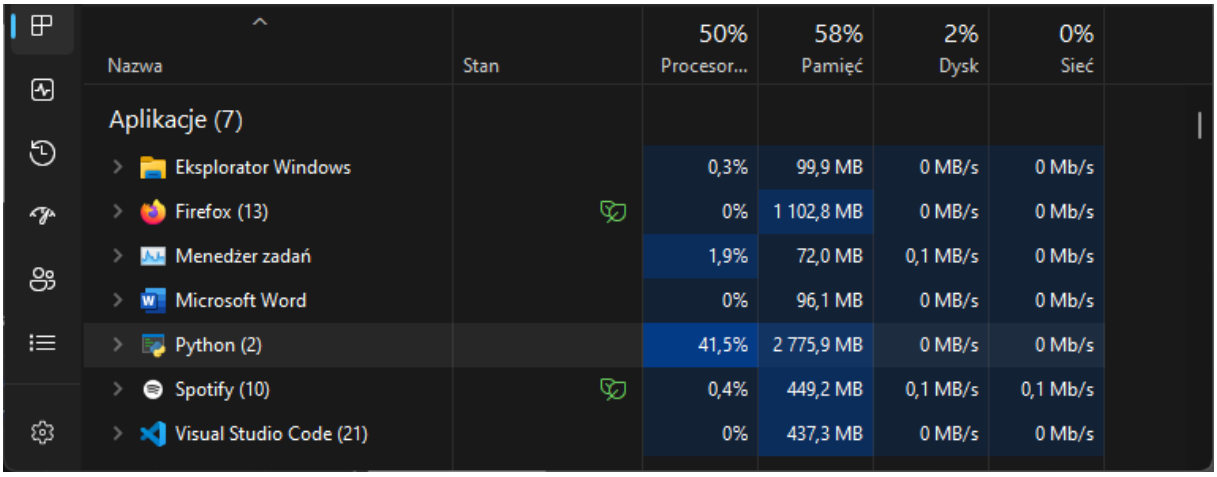
Test 7: Sprawdzenie zużycia pamięci podczas szyfrowania dużego pliku (>100 MB)
Cel: Sprawdzenie, czy interfejs użytkownika umożliwia poprawne wprowadzanie danych.
Kroki testowe:

- Przygotuj plik tekstowy o rozmiarze 100 MB.
- Wczytaj plik do aplikacji.
- Uruchom szyfrowanie algorytmem AES.
- Monitoruj użycie pamięci za pomocą systemowych narzędzi (np. Task Manager).

Oczekiwany wynik: Zużycie pamięci nie przekracza 20% zasobów systemowych.







Wynik testu: Udany. Aplikacja poprawnie zaszyfrowała plik 100 MB, a zużycie pamięci wyniosło około 17% zasobów systemowych, co jest akceptowalne. Test wykazał, że aplikacja jest wydajna, stabilna i dobrze zoptymalizowana do pracy z dużymi danymi.

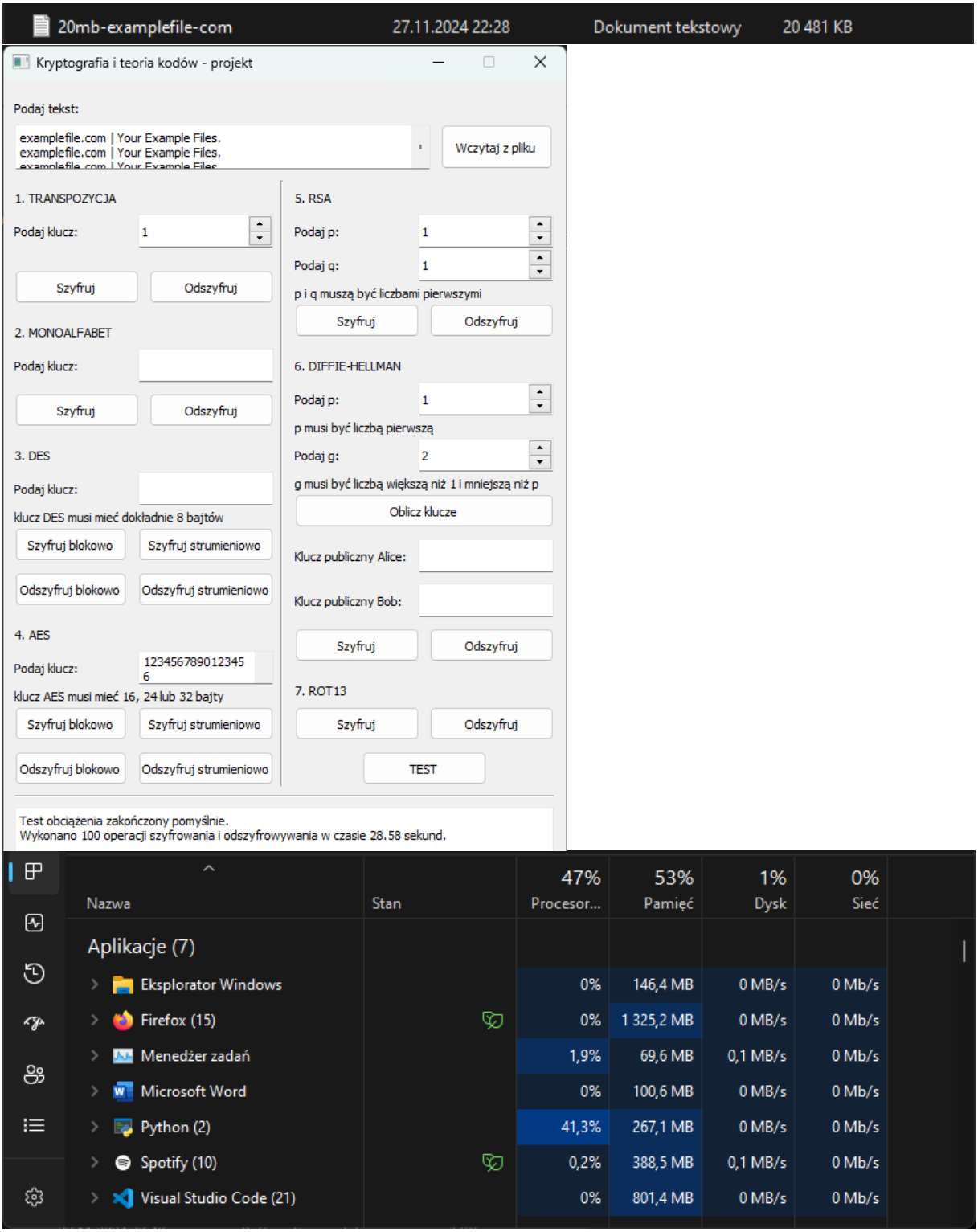
Test potwierdził, że aplikacja jest zdolna do przetwarzania dużych plików wejściowych, co czyni ją odpowiednią do zastosowań praktycznych, takich jak szyfrowanie dużych plików tekstowych czy danych produkcyjnych. Optymalizacja pamięci gwarantuje, że aplikacja może działać na komputerach z przeciętnymi zasobami sprzętowymi bez ryzyka przeciążenia.

8. Test przeciążenia i obciążenia

Test 8: Wykonanie 100 operacji szyfrowania/odszyfrowywania jednocześnie
Cel: Ocena stabilności aplikacji pod dużym obciążeniem.
Kroki testowe:

- W pętli wykonaj 100 operacji szyfrowania i odszyfrowywania tekstu.
- Obserwuj działanie aplikacji.

Oczekiwany wynik: Aplikacja działa stabilnie, ewentualnie wolniej, ale bez awarii.



Metoda wykorzystana do testu:

```
def test_obciazenia_aes(self):
        try:
            # Pobierz dane wejściowe od użytkownika
            tekst = self.ui.input.toPlainText()
            if not self.sprawdz_puste_pole(tekst):
                return

            # Pobierz klucz AES od użytkownika
            klucz = self.ui.klucz__aes.toPlainText().encode('utf-8')
            if len(klucz) not in [16, 24, 32]:
                self.ui.output.setPlainText("Klucz AES musi mieć 16, 24 lub 32 bajty.")
                return

            aes_cipher = AESCipher(klucz)

            # Test w pętli
            import time
            start_time = time.time()
            for i in range(100):
                # Szyfrowanie
                szyfrowany = aes_cipher.encrypt_block_mode(tekst)

                # Odszyfrowanie
                odszyfrowany = aes_cipher.decrypt_block_mode(szyfrowany)

                # Weryfikacja poprawności
                if odszyfrowany != tekst:
                      self.ui.output.setPlainText(f"Błąd w iteracji {i + 1}: Odszyfrowany tekst
różni się od oryginału.")
                    return

            # Pomiar czasu operacji
            end_time = time.time()
            elapsed_time = end_time - start_time

            # Wyświetlenie wyniku testu
            self.ui.output.setPlainText(f"Test obciążenia zakończony pomyślnie.\n"
                                        f"Wykonano 100 operacji szyfrowania i odszyfrowywania w
czasie {elapsed_time:.2f} sekund.")
        except Exception as e:
            self.ui.output.setPlainText(f"Błąd podczas testu obciążenia: {str(e)}")
```

Wynik testu: Udany. Czas wykonania 100 operacji w 28.58 sekundy jest satysfakcjonujący, biorąc pod uwagę wielkość pliku wejściowego (20 MB). Aplikacja efektywnie wykorzystuje zasoby systemowe, zużywając umiarkowaną ilość pamięci i procesora. Aplikacja działa stabilnie i responsywnie pod dużym obciążeniem. Test przeciążenia wykazał, że aplikacja jest wydajna, dobrze zoptymalizowana i odpowiednia do pracy z intensywnymi operacjami kryptograficznymi.

Test przeciążenia jest kluczowy dla oceny zdolności aplikacji do pracy w warunkach intensywnego użytkowania. Udane wykonanie testu wskazuje, że aplikacja może być używana w środowisku produkcyjnym, gdzie wykonywane są wielokrotne operacje kryptograficzne w krótkim czasie.

Podsumowanie

Przeprowadzone testy manualne aplikacji kryptograficznej objęły różnorodne aspekty jej działania, w tym niezawodność, efektywność, bezpieczeństwo, jakość kodu, użyteczność, pielęgnowalność oraz wydajność. Testowanie dostarczyło kluczowych obserwacji na temat mocnych stron aplikacji oraz obszarów wymagających dalszych ulepszeń.

Obserwacje i wnioski:

1. Niezawodność:
   - Aplikacja skutecznie realizuje swoje podstawowe funkcje szyfrowania i odszyfrowywania danych przy użyciu różnych algorytmów. Testy niezawodności wykazały stabilność działania zarówno w standardowych warunkach, jak i przy obsłudze błędnych danych wejściowych (np. nieprawidłowe klucze czy puste pola).
   - Obsługa błędów jest intuicyjna, a komunikaty informują użytkownika o problemach w sposób przejrzysty.
2. Efektywność:
   - Testy wydajności potwierdziły, że algorytmy, takie jak AES, działają bardzo szybko, nawet w przypadku dużych danych (1 MB zaszyfrowano w 0.01 sekundy).
   - Aplikacja spełnia założenia efektywności, co czyni ją odpowiednią do pracy w czasie rzeczywistym, np. w systemach przetwarzania dużych ilości danych.
3. Bezpieczeństwo:
   - Mechanizmy walidacji (np. w protokole Diffie-Hellman) skutecznie rozpoznają błędne dane, takie jak fałszywe klucze publiczne. To gwarantuje odporność aplikacji na potencjalne manipulacje danymi.
4. Jakość kodu:
   - Struktura kodu jest modularna, co ułatwia jego utrzymanie i rozwój. Każdy algorytm został zaimplementowany w osobnym module, co zwiększa przejrzystość i wspiera zasadę pojedynczej odpowiedzialności.
   - Można zaobserwować pewne niedoskonałości, takie jak duplikacja kodu w interfejsie użytkownika (GUI), co zwiększa nakład pracy przy wprowadzaniu zmian. Wprowadzenie klasy bazowej dla algorytmów mogłoby znacząco poprawić strukturę aplikacji.
5. Użyteczność:
   - Interfejs użytkownika jest intuicyjny i umożliwia łatwe wprowadzanie danych. Testy z tekstami zawierającymi znaki specjalne oraz różnorodne klucze pokazały, że aplikacja obsługuje szeroką gamę scenariuszy użytkowych.
6. Pielęgnowalność:
   - Test dodania algorytmu ROT13 wykazał, że aplikacja jest łatwa w rozbudowie dzięki modularnej architekturze. Wprowadzanie nowych funkcji jest szybkie i nie wymaga dużych zmian w istniejącym kodzie.

7. Wydajność i stabilność:
    - Testy obciążenia (100 operacji szyfrowania i odszyfrowywania w pętli) potwierdziły stabilność aplikacji nawet pod dużym obciążeniem. Czas wykonania 28.58 sekundy dla operacji na pliku o rozmiarze 20 MB jest akceptowalny.

Znaczenie testowania manualnego:

Testowanie aplikacji pozwoliło na:

- Wykrycie i naprawienie potencjalnych błędów (np. brak walidacji pustych pól).
- Ocenę jakości i stabilności aplikacji z perspektywy użytkownika końcowego.
- Potwierdzenie, że aplikacja spełnia założenia funkcjonalne i wydajnościowe.
- Wskazanie możliwości dalszych ulepszeń, takich jak optymalizacja kodu oraz automatyzacja powtarzalnych testów.