Dokument Testowania

Spis Treści

Wstęp

Cel Dokumentu

Celem tego dokumentu jest przedstawienie planu testowania dla UJPoly gry bazującej na Monopoly. Dokument opisuje strategię testowania, środowisko testowe, przypadki testowe oraz metryki używane do oceny jakości oprogramowania.

Zakres Testowania

Testowanie obejmuje weryfikację funkcjonalności gry Monopoly, w tym inicjalizację gry, liczbę graczy oraz tury gry dla jednego i wielu graczy.

Opis Testowanego Systemu

Testowany system to UJPoly zmodyfikowana wersja gry Monopoly, zaimplementowana przy użyciu języka Python i biblioteki Pygame. Gra obsługuje wizualizację, komunikację między komponentami oraz mechanikę gry dla wielu graczy.

Cel i Zakres Testowania

Główne Cele Testowania

- **Weryfikacja:** poprawnej inicjalizacji gry.
- **Sprawdzenie:** czy gra obsługuje poprawną liczbę graczy.
- **Testowanie:** logiki poszczególnych komponentów gry.
- **Testowanie mechaniki:** gry dla różnych liczby graczy.

Zakres Testów

- **Testy jednostkowe** dla komponentów gry.
- **Testy integracyjne** dla sprawdzenia współdziałania różnych komponentów.
- **Smoke testy** dla szybkiej weryfikacji stabilności aplikacji.
- **Testy manualne** przeprowadzane przez testera oraz grupę deweloperów.

Strategia Testowania

Ogólne Podejście

Testy będą przeprowadzane przy użyciu biblioteki *pytest* dla automatyzacji testowania. Testy będą obejmować zarówno testy jednostkowe, jak i testy integracyjne. Smoke testy będą używane do szybkiej weryfikacji kluczowych funkcji aplikacji.

Testy mutacyjne będą przeprowadzane przy użyciu biblioteki *mutmu*t dla automatyzacji testów. Biblioteka *mutmut* korzysta z biblioteki *pytest.*

Testy manualne będą przeprowadzane przez testera oraz grupę deweloperów gry. Testy będą obejmować elementy interakcji gry z użytkownikiem.

Typy Testów

- **Testy jednostkowe:** Sprawdzenie pojedynczych funkcji i metod.
- **Testy integracyjne:** Weryfikacja współdziałania między różnymi komponentami gry.
- **Testy mutacyjne:** Wykorzystywane w celu wykrycia potencjalnie nowych defektów.
- **Smoke testy:** Szybka weryfikacja kluczowych funkcji aplikacji, takich jak inicjalizacja i tury gry.
- **Testy manualne:** Wykorzystane w celu wykrycia defektów, luk w użytkowaniu interfejsu przez użytkownika.

Narzędzia Używane do Testowania

- **pytest** - framework do automatyzacji testów.
- **unittest** – moduł używany do tworzenia mock'ów, ich aplikacji
- **mutmut** - biblioteka używana do tworzenia testów mutacyjnych
- **pygame** - biblioteka używana do tworzenia gry.

Skrypty używane podczas testowania:

- *python -m **pytest** ./tests*
- *python ./**tests**/mutacyjne/**uruchom_mutacyjne.py***

Środowisko Testowe

Opis Środowiska

- **System operacyjny:** Windows 11 Home
- **Python:** Wersja 3.10
- **Pygame:** Wersja 2.5.2
- **Mutmut:** Wersja 2.5.0
- **Pytest:** Wersja 8.2.0

Dane Testowe

- Predefiniowane nazwy graczy (np. "test_1", "test_2")
- Predefiniowane zdarzenia w grze
- Konfiguracje ekranu (szerokość i wysokość)

Plan Testów

Harmonogram Testów

- **Faza 1:** Testy jednostkowe – po wprowadzeniu zmian przez developerów oraz po każdym większym wydaniu (czyt. połączeniu gałęzi na main w git'cie).
- **Faza 2:** Testy integracyjne – Po każdym większym wydaniu (czyt. połączeniu gałęzi na main w git'cie).
- **Faza 3:** Smoke testy - 1 dzień

Kryteria Wejścia i Wyjścia

Wejście: Kod źródłowy gry, środowisko testowe skonfigurowane.

Wyjście: Raport z wynikami testów, zgłoszone defekty.

Zasoby Potrzebne do Przeprowadzenia Testów

- Testerzy: 1 osoba
- Komputery z odpowiednim środowiskiem testowym.

Specyfikacja Testów

Przypadki Testowe

1. **Test Inicjalizacji Gry**
    - **Identyfikator:** test_Gra.py
    - **Opis:** Weryfikacja poprawnej inicjalizacji komponentów gry.
    - **Warunki Początkowe:** Brak
    - **Kroki do Wykonania:** Uruchomienie metody inicjalizacyjnej, odpowiednich konstruktorów.
    - **Oczekiwany Wynik:** Komponenty gry powinny być poprawnie zainicjalizowane, gra powinna się włączyć.

Wyniki Testów

Sposób Dokumentowania Wyników Testów

Wyniki testów będą dokumentowane w formie raportów HTML generowanych przez bibliotekę mutmut oraz wersji papierowej zgłaszanej do testera. Raporty będą zawierać

informacje o liczbie testów przeprowadzonych, liczbie testów zakończonych sukcesem oraz liczbie testów zakończonych niepowodzeniem.

Format Raportów z Testów

Raporty będą generowane w formacie HTML oraz tekstowej, zawierającym szczegółowe informacje o każdym teście, wyjściu z oprogramowania do testowania.

Procedury Raportowania Błędów

Błędy będą zgłaszane i śledzone przy użyciu systemu komunikacji z deweloperami gry ze względu na małą liczbę osób w zespole oraz zamieszczaniu błędów na platformie GitHub w zakładce Issues. Większe błędy będzie mieć przypisany unikalny identyfikator, opis, kroki do reprodukcji oraz priorytet, błędy mniejsze przy ocenie przez zespół programistów będą naprawiane w chwili ich wykrycia przy spotkaniu online zwołanym w trybie pilnym.

Role i Odpowiedzialności

Zespół Testowy

- **Tester 1:** Odpowiedzialny za testy jednostkowe, testy mutacyjne, testy dymne(smoke) oraz za całokształt przeprowadzonych testów, raportowanie do kierownika projektu o wykrytych oraz naprawionych błędach.

Zarządzanie Defektami

Proces Zgłaszania, Śledzenia i Zarządzania Defektami

- Zgłaszanie defektów przez system zarządzania defektami.
- Przypisanie defektów do testera.
- Śledzenie statusu defektów (otwarty, w trakcie, zamknięty).

Narzędzia Używane do Zarządzania Defektami

- GitHub

- Discord

Kryteria Priorytetyzacji Defektów

- **Krytyczne:** Błędy blokujące działanie aplikacji.
- **Wysokie:** Błędy wpływające na kluczowe funkcjonalności.
- **Średnie:** Błędy wpływające na mniej istotne funkcje.
- **Niskie:** Błędy kosmetyczne.

Ryzyka i Zarządzanie Ryzykiem

Identyfikacja Potencjalnych Ryzyk

- Niezgodność środowiska testowego z produkcyjnym.
- Opóźnienia w harmonogramie testów.
- Brak testera w poszczególnych dniach.
- Brak zasobów testowych.

Plany Łagodzenia Ryzyk

- Regularne aktualizacje środowiska testowego.
- Monitorowanie postępu testów i dostosowywanie harmonogramu.
- Planowanie z kilkudniowym wyprzedzeniem czasu przeprowadzania testów.
- Współpraca z zespołem programistów w celu szybkiego rozwiązywania problemów.

Metryki i Raportowanie

Metryki Zbierane Podczas Testów

- Liczba przeprowadzonych testów.
- Liczba testów zakończonych sukcesem.
- Liczba testów zakończonych niepowodzeniem.
- Czas trwania testów.
- Liczba zgłoszonych defektów.

Sposoby i Częstotliwość Raportowania Wyników

- Co wydanie raport postępu.
- Raporty końcowe po zakończeniu faz testowych.

Podsumowanie

Kluczowe Wnioski

- Testy potwierdziły poprawną inicjalizację i działanie kluczowych funkcji gry.
- Zidentyfikowano i zgłoszono defekty, które wymagają naprawy przed wdrożeniem.

Następne Kroki

- Naprawa zgłoszonych defektów.
- Przeprowadzenie testów regresji po naprawie defektów.

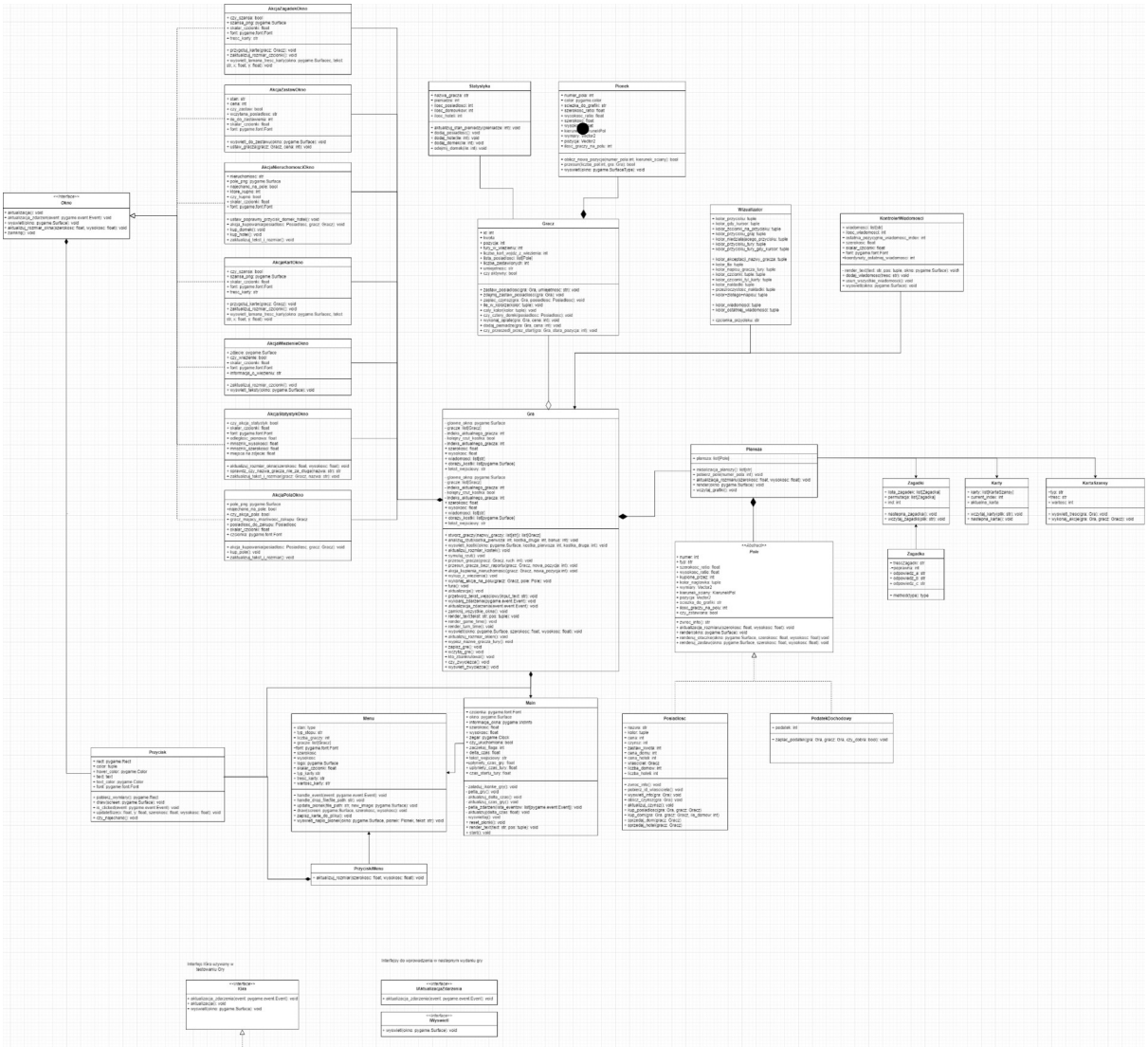
- Przygotowanie aplikacji do wdrożenia produkcyjnego.

Aneksy

Dodatkowe Informacje

- Schematy architektury systemu.
- Przykłady danych testowych.
- Instrukcje konfiguracji środowiska testowego.

Schemat architektury systemu: Diagram UML(zawarty również w dokumentacji).



Przykłady danych testowych:

- **nazwa gracza:** “test1”, “test2”, etc.
- **kwota_startowa**: 2_000, 10_000, 15_000, etc.
- **pozycja_przycisku_x**: 1200, 1300, 500, etc.
- **pozycja_przycisku_y:** 1400, 1550, 991, etc.
- **kolor_przycisku**: pygame.color.THECOLORS[“white”], (120, 131, 221), etc.
- **numer_pola:** list(range(0, 40)), -1, -2, -100

Instrukcje konfiguracji środowiska testowego:

- *pip install –force-reinstall -v “mutmut==2.5.0”*
- *pip install –force-reinstall -v “pygame==2.5.2”*
- *pip install –force-reinstall -v “pytest==8.2.0”*
- *ręczna instalacja pythona w wersji 3.10.**

Przykładowe testy:

- Test komponentu **Plansza**:

```
import pygame
import pytest

from src.Pole import Pole
from src.Plansza import Plansza


class TestPlansza:
def setup_method(self):
pygame.init()

self.plansza = Plansza()

def test_typow_klasy_plansza(self):
assert isinstance(self.plansza.plansza, list)

@pytest.mark.parametrize("numer_pola", list(range(0, 40)))
def test_pobierz_pole_poprawny_zakres(self, numer_pola):
rezultat_pole = self.plansza.pobierz_pole(numer_pola)

assert isinstance(rezultat_pole, Pole)

@pytest.mark.parametrize("numer_pola", [-2, -1, -100, 41, 42, 50, 100, 1000])
def test_pobierz_pole_niepoprawny_zakres(self, numer_pola):
with pytest.raises(Exception):
rezultat = self.plansza.pobierz_pole(numer_pola)

def test_wczytaj_grafiki(self):
path = "graphics/pola/pole_"
extension = ".png"
nr_pola = 0

self.plansza.wczytaj_grafiki()

for pole in self.plansza.plansza:
tekst_sciezki = (path + str(nr_pola) + extension)
nr_pola += 1
```

```
assert pole.sciezka_do_grafiki == tekst_sciezki
```

- Test komponentu **TestAkcjaZastawOkno:**

```
import pygame
import pytest

from src.Wizualizator import Wizualizator
from src.Gracz import Gracz
from src.okno.AkcjaZastawOkno import AkcjaZastawOkno
from unittest.mock import MagicMock


class TestAkcjaZastawOkno:
def setup_method(self):
pygame.init()

mock_gra = MagicMock()
mock_gra.wizualizator = Wizualizator()

self.akcja_zastaw_okno = AkcjaZastawOkno(gra=mock_gra)

def test_typ_zmiennych_zainicjalizowanych(self):
assert isinstance(self.akcja_zastaw_okno.gracz, (type(None), Gracz))
assert isinstance(self.akcja_zastaw_okno.czy_zastaw, bool)

@pytest.mark.xfail(reason="Po ustawieniu gracza, gracz nie moze byc typu None")
def test_czy_gracz_null(self):
event = pygame.event.Event(type=pygame.MOUSEBUTTONDOWN)
self.akcja_zastaw_okno.aktualizacja_zdarzen(event)

assert self.akcja_zastaw_okno.gracz is not None

@pytest.mark.xfail(reason="Gracz nie moze byc typu None, podczas wykonywania akcja na nim")
def test_zastaw_gracz_none(self):
self.akcja_zastaw_okno.gracz = None
self.akcja_zastaw_okno.gracz.zastaw_posiadlosci()

@pytest.mark.skip
def test_klik_event_zastaw(self):
# magic mock here
pass
```