Projekt PROI:

Firma testująca gry komputerowe

Autorzy:

Paweł Kochański

Jakub Proboszcz

1. Założenia projektu

Celem projektu jest zasymulowanie działania firmy testującej gry komputerowe w paradygmacie programowania zorientowanego obiektowo.

Firma ma testerów i managerów - testerzy testują gry im przydzielone, managerzy przetwarzają zlecenia testowania i przydzielają testerów do gier.

Podział gier na kategorie następuje w zależności od ich gatunku.

Przydzielanie gier do pewnych kategorii, od których zależy pula potencjalnych testerów jest wykonywane w pliku wejściowym gier, przez użytkownika - zmienna minTestersAmount obecna w grach określa liczbę testerów (maksymalna ich liczba jest jej dwukrotnością).

Opłaty wstępne za testowanie gier są obliczane tylko na podstawie gier - producent może to policzyć w dowolnym momencie. Rzeczywista opłata jest zwracana producentowi na końcu testowania.

Raport z testowania i wnioski od testerów nie są zareprezentowane, ponieważ przebieg testowania nie jest szczegółowo symulowany (wiadomo, że testerzy pracują nad daną grą) - można przyjąć, że przy wywołaniu funkcji powiadamiającej producenta o zakończeniu testowania takie dane zostają również przekazane.

2. Hierarchia klas

Podział programu na klasy:

Program zawiera dwie hierarchie klas i 13 klas autonomicznych (nie licząc wyjątków).

Hierarchia gier zawiera klasy AbstractGame, Game, Puzzle, RolePlayingGame, InfiniteGame oraz CompetitiveGame.

AbstractGame definiuje interfejs publiczny gry. Poza czysto wirtualnymi metodami zawiera również publiczne stałe: ID oraz referencja na producenta, oraz protected konstruktor, aby obiekty klas potomnych mogły te pola zainicjalizować.

Game definiuje ogólną grę, której gatunek jest nieokreślony.

Puzzle, RolePlayingGame, InfiniteGame, CompetitiveGame reprezentują różne gatunki gier, z różnymi polami i różnymi sposobami obliczania ich czasu i ceny testowania.

Hierarchia pracowników zawiera klasy AbstractWorker, Worker, Manager oraz Tester.

AbstractWorker definiuje interfejs publiczny pracownika. Poza czysto wirtualnymi metodami zawiera również publiczne stałe ID, oraz protected konstruktor, aby obiekty klas potomnych mogły to pole zainicjalizować.

Worker definiuje ogólnego pracownika, nie jest wiadome czym się on zajmuje.

Tester reprezentuje pracownika, który testuje przydzieloną do niego grę.

Manager reprezentuje pracownika, który przetwarza zlecenia testowania gier i przydziela gry do testerów.

Klasy autonomiczne to wszystkie pozostałe elementy symulacji.

Producer reprezentuje producenta gier, który produkuje gry i zleca ich testowanie.

ProducerDatabase jest bazą danych producenta, która przechowuje wyprodukowane gry i status ich testowania za pomocą podklasy ProducerDatabase::Record.

Address reprezentuje adres producenta. Z racji swojej natury (obiektów tej klasy nie trzeba rozróżniać, można je dowolnie kopiować) nie posiada swojego unikalnego identyfikatora.

TestingCompany reprezentuje firmę testującą gry komputerowe - przechowuje wszystkich pracowników i wszystkich testerów w dwóch kolekcjach typu std::vector oraz informacje o przetestowanych grach i płatnościach za nie w kolekcji std::vector obiektów podklasy TestingCompany::Record.

TestingDatabase reprezentuje bazę danych nieprzetestowanych gier powyższej firmy. Przechowuje ona kolejkę (std::queue) nieprzetworzonych zleceń testowania (obiektów podklasy TestingDatabase::Request), kolekcję std::deque obecnie nietestowanych gier (dokładniej obiektów std::unique_ptr<TestingRecord>) (gry z przetworzonych zleceń trafiają

na koniec, gry tymczasowo wycofane z testowania na początek) oraz kolekcję std::list obecnie testowanych gier (dokładniej obiektów std::unique_ptr<TestingRecord>).

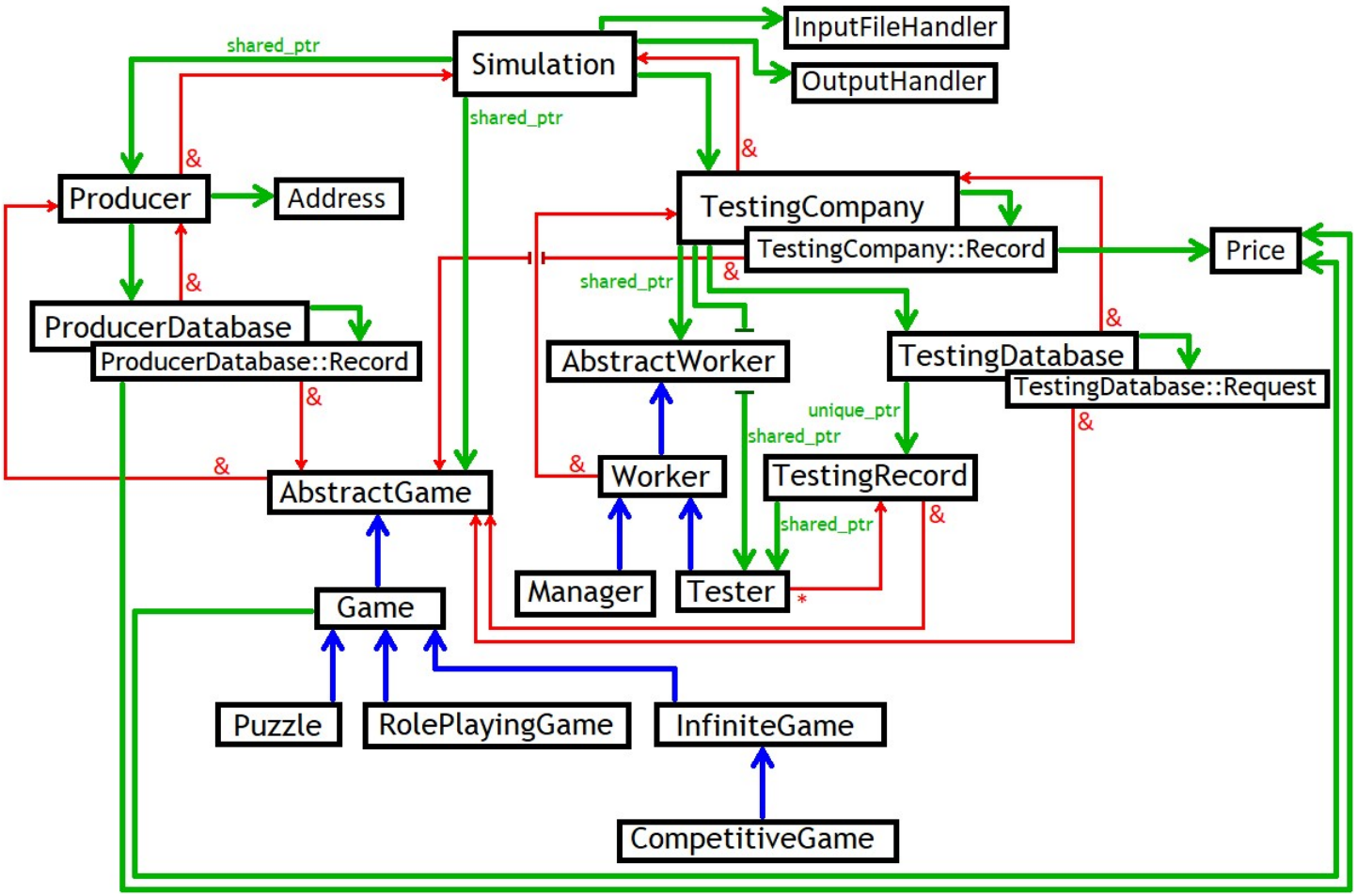
Price reprezentuje cenę w polskiej walucie PLN. Z racji swojej natury (obiektów tej klasy nie trzeba rozróżniać, można je dowolnie kopiować) nie posiada swojego unikalnego identyfikatora.

TestingRecord reprezentuje rekord bazy danych TestingDatabase. Przechowuje referencję na grę, wskaźniki do przydzielonych do niej testerów i informacje o stanie testowania.

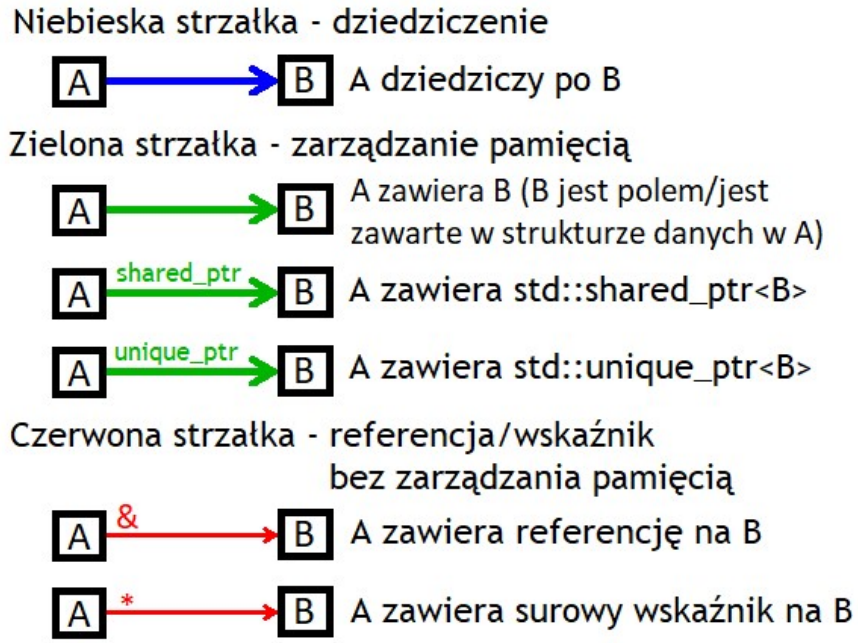
InputFileHandler wczytuje dane z plików, a OutputHandler wypisuje informacje o przebiegu symulacji do pliku oraz do konsoli.

Simulation zarządza symulacją jako całością - jej metoda simulate() zawiera główną pętlę symulacji. Nie zawiera unikalnego identyfikatora - nie trzeba rozróżniać symulacji, bo jest tylko jedna (nawet jeśli utworzono by kilka, nie wchodziłyby ze sobą w interakcję)

Schemat relacji między klasami programu:



Objaśnienia oznaczeń:



Na schemacie, dla lepszej czytelności, nie zostały oznaczone liczne referencje do obiektu klasy OutputHandler. Są one zawarte w klasach: Producer, ProducerDatabase, TestingCompany, Worker (i klasy potomne), TestingDatabase, TestingRecord.

3. Opis działania symulacji

Argumenty wywołania programu

liczba testerów - liczba całkowita nieujemna

liczba managerów - liczba całkowita nieujemna

nazwa pliku z danymi producentów

nazwa pliku z danymi gier

nazwa pliku z danymi testerów

nazwa pliku z danymi managerów

nazwa pliku wyjściowego

liczba godzin, ile ma trwać symulacja

Dla przykładowych plików wywołanie programu może wyglądać tak:

./PROI_22L_206_FirmaTestujacaGryKomputerowe 9 2 producers.txt games.txt testers.txt managers.txt simulationlog.txt 50

Do kompilacji testów powinien zostać odkomentowany define w pliku nagłówkowym outputhandler.h, który zablokuje wypisywanie statusu symulacji na ekran.

Format plików wejściowych

Plik z danymi producentów (na przykładzie):

Piekarnia Pawełek	- nazwa producenta
Legionów	- ulica, przy której mieści się siedziba główna firmy
66	- numer domu (siedziby głównej firmy)
0	- numer mieszkania (siedziby głównej firmy) (0 jeśli brak)
Wołomin	- miasto (siedziby głównej firmy)
05-200	- kod pocztowy powyższego miasta

Producentów w takim formacie może być dowolna ilość, nie mniejsza od argumentu linii poleceń określającego liczbę producentów.

Plik z danymi gier (na przykładzie):

W przykładzie jest po jednej grze każdego typu. Ich ilość nie ma znaczenia, poza tym, że musi być w sumie co najmniej jedna gra.

infinite	- typ gry (od tego zależą następujące dane gry)
The Binding Of Isaac: Repentance	- nazwa
200000	- rozmiar plików gry w KB
2	- skomplikowanie gry (skala od 0 do 2)
4	- minimalna liczba testerów dla gry
2	- głębokość nieskończonej gry (skala od -1 do 2)
false	- czy kod źródłowy gry jest dostępny dla testujących
200	- cena gry dla użytkowników (pełne złote)
99	- cena gry dla użytkowników (grosze)

puzzle	- typ gry (od tego zależą następujące dane gry)
Helltaker	- nazwa
40000	- rozmiar plików gry w KB
1	- skomplikowanie gry (skala od 0 do 2)
1	- minimalna liczba testerów dla gry
1	- trudność gry (skala od -1 do 2)
90	- czas przejścia gry w minutach
true	- czy kod źródłowy gry jest dostępny dla testujących
0	- cena gry dla użytkowników (pełne złote)
0	- cena gry dla użytkowników (grosze)

roleplaying	- typ gry (od tego zależą następujące dane gry)
Wiedźmin 3: Dziki Gon	- nazwa
300000	- rozmiar plików gry w KB
2	- skomplikowanie gry (skala od 0 do 2)
5	- minimalna liczba testerów dla gry
480	- czas przejścia gry w minutach
900	- czas pełnego przejścia gry w minutach
false	- czy kod źródłowy gry jest dostępny dla testujących
250	- cena gry dla użytkowników (pełne złote)
0	- cena gry dla użytkowników (grosze)

competitive	- typ gry (od tego zależą następujące dane gry)
League of Legends	- nazwa
150000	- rozmiar plików gry w KB
2	- skomplikowanie gry (skala od 0 do 2)
6	- minimalna liczba testerów dla gry
1	- głębokość nieskończonej gry (skala od -1 do 2)
2000	- liczba graczy, których ma móc obsłużyć serwer gry
false	- czy kod źródłowy gry jest dostępny dla testujących
0	- cena gry dla użytkowników (pełne złote)
0	- cena gry dla użytkowników (grosze)

normal	- typ gry (od tego zależą następujące dane gry)
The Normalest Game	- nazwa
50000	- rozmiar plików gry w KB
2	- skomplikowanie gry (skala od 0 do 2)
3	- minimalna liczba testerów dla gry
true	- czy kod źródłowy gry jest dostępny dla testujących
0	- cena gry dla użytkowników (pełne złote)
0	- cena gry dla użytkowników (grosze)

Plik z danymi testerów (na przykładzie):

Paweł	- imię testera
Piekarski	- nazwisko testera

Testerów w takim formacie może być dowolna ilość, nie mniejsza od argumentu linii poleceń określającego liczbę testerów.

Plik z danymi managerów (na przykładzie):

Paweł	- imię managera
Piekarski	- nazwisko managera

Managerów w takim formacie może być dowolna ilość, nie mniejsza od argumentu linii poleceń określającego liczbę managerów.

Przebieg symulacji

Utworzenie obiektu klasy Simulation zostają otwarte pliki wejściowe i wyjściowy. Właściwą symulację rozpoczyna wywołanie funkcji simulate utworzonego obiektu.

Najpierw wczytywane są dane z plików wejściowych producentów, managerów i testerów. Potem uruchamiana jest pętla symulacji. W jej trakcie:

O minięciu godziny powiadamiany jest każdy z producentów - mają oni szansę na utworzenie nowej gry (symulacja załaduje ją z pliku) oraz na wysłanie prośby o testowanie. Jeżeli licznik czasu od końca testowania dojdzie do końca, producent może również zapłacić za testowanie.

Następnie o minięciu godziny powiadamiana jest firma testująca. Wykonuje ona następujące czynności:

Wywołuje pracowników, aby wykonali swoją pracę. Testerzy pracują bezpośrednio nad swoimi grami (jeśli mają je przydzielone), a praca Managerów jest „zbierana” przez firmę w zmiennej effort.

Wywołuje bazę danych testowania, by sprawdziła, czy któraś gra została przetestowana. Jeśli została, testerzy do niej przydzieleni są zwalniani i powiadomienie o tym jest przekazywane do firmy i producenta.

Następnie za cenę pracy managerów (effort) firma próbuje przydzielić testerów do gier. Jeśli to spowoduje, że testowanie się rozpocznie (liczba testerów przekroczyła minimalną), baza danych to zarejestruje.

Następnie pozostały effort z danej godziny zostaje zużyty na przetwarzanie zleceń testowania w bazie danych. Ukończone zlecenia są dodawane do obecnie testowanych lub czekających na testowanie gier.

Potem następuje przejście do kolejnej godziny, czyli kolejnej iteracji pętli w funkcji simulate.

4. Wybrane elementy biblioteki STL

Elementy używane do obsługi strumieni i komunikacji z użytkownikiem:

- **cerr** – wyjście standardowe błędu użyte do wypisania informacji o błędach w funkcji main

- **cout** – wyjście standardowe użyte do wypisywania informacji dla użytkownika

- **endl** – manipulator przejścia do nowej linii

- **ostream** – ogólny typ strumienia wyjściowego, użyty do przeładowywania operatora <<

- **stringstream** – strumień wyjściowy używany do testowania operatora <<

- **this_thread::sleep_for** – odczekanie określonego czasu po wypisaniu wiadomości, aby użytkownik mógł przeczytać wypisaną wiadomość

Elementy użyte przy wykorzystywaniu zmiennych tekstowych (string):

- **string** – podstawowy typ zmiennej składającej się z wielu znaków

- **stoi** – konwersja ciągu znaków na liczbę całkowitą

- **to_string** – konwersja zmiennej na ciąg znaków

- **string::npos** – pokazuje że nie znaleziono danych znaków w danym ciągu znaków

Klasy do definiowania i obsługi wyjątków:

- **invalid_argument** – wyjątki sygnalizujące niepoprawny argument

- **out_of_range** – wprowadzona liczba nie mieści się w typie danych używanym w programie

- **logic_error** – błędy logiki programu

- **exception** - ogólny wyjątek wykorzystany przy łapaniu dowolnego wyjątku

Kolekcje obiektów:

- **vector** – uniwersalna kolekcja elementów w postaci ciągłej tablicy o zmiennym rozmiarze, ale stałym adresie pierwszego elementu

- **queue** – kolekcja działająca na zasadzie FIFO (first in – first out)

- **deque** – (double-ended queue, kolejka z dwoma końcami) - kolekcja pozwalająca na efektywne dodanie i usunięcie elementów z początku i końca

- **list** – kolekcja umożliwiająca efektywne dodawanie i usuwanie elementów z dowolnego jej miejsca, kosztem braku szybkiego dostępu do dowolnego jej elementu po indeksie

- **find** – znajdowanie wybranego elementu w danej kolekcji

- **find_if** – znajdowanie elementu spełniającego podany warunek w danej kolekcji

Generowanie liczb losowych:

- **chrono::system_clock::now().time_since_epoch().count()** – odliczenie czasu od 1 stycznia 1970 w celu wygenerowania seed'a do generatora liczb losowych

- **mt19937** – wybrany generator liczb losowych

„Inteligentne" wskaźniki:

- **shared_ptr** – „inteligentny" wskaźnik do danego obiektu mogący występować w wielu kopiach. Element jest usuwany dopiero po usunięciu wszystkich wskazujących na niego shared_ptr

- **make_shared** – tworzy „inteligentny" wskaźnik shared_ptr do danego obiektu

- **unique_ptr** – „inteligentny" wskaźnik do danego obiektu mogący występować tylko w jednej kopii. Kiedy przestaje istnieć, wykonuje usunięcie obiektu, na który wskazuje

- **make_unique** – tworzy „inteligentny" wskaźnik unique_ptr do danego obiektu

- **move** – przesuwanie elementów (konkretnie wskaźników typu unique_ptr) na inne miejsce w pamięci, celem uniknięcia ich kopiowania

Zarządzanie plikami:

- **ofstream** – strumień pliku wyjściowego

- **ifstream** – strumień pliku wejściowego

- **getline** – zapisanie wiersza (sekwencji znaków zakończonej znakiem nowej linii) z pliku wejściowego do zmiennej w programie

5. Zdefiniowane sytuacje wyjątkowe i ich obsługa

W programie zostały zdefiniowane następujące sytuacje wyjątkowe (klasy wyjątków):

- **ConversionError**
    - niepoprawna próba skonwertowania ciągu znaków wczytanego z pliku na dany typ
- **DuplicateGameError**
    - próba dodania do bazy danych producenta gry, która już się tam znajduje
- **EmptyNameException**
    - próba ustawienia nazwy obiektu na składającą się wyłącznie z białych znaków lub pustą
- **EndOfFileError**
    - próba wczytania danych z pliku po dojściu do jego końca
- **FileError**
    - niepowodzenie otwarcia pliku (np. plik wejściowy nie istnieje)
- **GameAlreadyTestedError**
    - próba rozpoczęcia testowania już przetestowanej gry
- **GameNotPresentError**
    - próba zmiany gry w bazie danych producenta, kiedy ona się tam nie znajduje
- **GameNotRequestedError**
    - próba oznaczenia gry w bazie danych producenta jako przetestowanej, kiedy jej testowanie nie zostało nigdy rozpoczęte
- **IncorrectAddressException**
    - próba utworzenia adresu z kodem pocztowym w formacie innym niż dd-ddd (d - cyfra)
- **InvalidFileSize**
    - próba ustawienia rozmiaru plików gry na 0 KB
- **InvalidFullLength**
    - próba ustawienia pełnej długości (czasu potrzebnego na pełne doświadczenie) gry fabularnej na krótszą od bazowej długości tej gry
- **InvalidId**
    - próba ustawienia identyfikatora obiektu na wartość niedozwoloną dla tej klasy obiektów

- **InvalidLength**

    - próba ustawienia bazowej długości gry na 0 minut

- **InvalidPrice**

    - próba ustawienia ceny na nieodpowiednią wartość (ujemna wartość, ujemna liczba groszy)

- **InvalidProducer**

    - próba dodania gry wyprodukowanej przez innego producenta do bazy danych producenta

- **InvalidTestersAmount**

    - próba ustawienia minimalnej liczby testerów wymaganej do przetestowania danej gry na 0, bądź próba ustawienia maksymalnej liczby testerów na mniejszą od minimalnej

- **InvalidTitle**

    - próba ustawienia tytułu gry na składający się wyłącznie z białych znaków lub pusty

- **NoGamesUntestedError**

    - próba pobrania gry do testów, gdy żadna nie jest dostępna.

- **TestingEndedError**

    - próba modyfikacji rekordu już przetestowanej gry

- **TestingNotEndedError**

    - próba odczytania końcowych parametrów nie w pełni przetestowanej gry

- **InvalidHouseNumberError**

    - próba ustawienia na zero wartości numeru domu w adresie

Obsługa sytuacji wyjątkowych:

Wyjątki są łapane w funkcji przeprowadzającej symulację, wypisywane do pliku i na wyjście standardowe cout po czym zgłaszane jest **ShutdownException** sygnalizujące ową sytuację wyjątkową dla programu. ShutdownException łapane jest w funkcji main, co kończy tym samym symulację wypisując informację o tym na wyjście cerr.

Poza tym wyjątek **EndOfFileError** jest wykorzystywany przy tworzeniu producentów by zakończyć ich tworzenie bez potrzeby informowania programu o ich ilości.

6. Podział obowiązków

Paweł Kochański:

Klasy: Address, Worker, Manager, Tester, TestingCompany, InputFileHandler, OutputHandler, Simulation, interfejs AbstractWorker

Wyjątki związane z ww. klasami

Testy jednostkowe ww. klas

Jakub Proboszcz:

Klasy: Price, Game, Puzzle, CompetitiveGame, InfiniteGame, RoleplayingGame, Producer, ProducerDatabase, TestingDatabase, TestingRecord, interfejs AbstractGame

Wyjątki związane z ww. klasami

Testy jednostkowe ww. klas

Wspólnie:

Naprawianie zauważonych błędów i problemów w plikach drugiej osoby

Main.cpp

Sprawozdanie

Testowe pliki wejściowe

Testy integracyjne - debugowanie programu jako całości