Mateusz WASILEWSKI

Projekt: Automatyczne pokładowanie pasażerów samolotu

Architektura projektu

Projekt został podzielony na następujące części:

* Osobne moduły do prawidłowej reprezentacji danych (klasy)
* Moduł (klasa) bazy danych odpowiedzialna za przechowywanie danych
* Moduły umożliwiające wczytywanie do bazy danych plików csv
* Moduły interfejsu użytkownika
  + ui.py – moduł zawierający właściwy interfejs użytkownika
  + console\_ui.py – moduł zawierający przykładowy interfejs wykorzystujący argparse
    - przyjmowane argumenty:
      * OPERATION – należy podać jedną z następujących wartości:
        + flights
        + planes
        + tickets
        + passengers
        + boarding\_pass
        + flight\_params
        + check\_gate
      * --id
        + argument opcjonalny dla parametrów flights, planes, tickets, passengers pozwalający wybranie konkretnego rekordu z bazy danych
        + argument konieczny dla prawidłowego działania parametrów boarding\_pass, flight\_params, check\_gate
* Moduł zawierający definicje wyjątków pojawiających się w czasie działania programu
* Moduł uruchamiający program
* Moduły z testami odpowiednich metod, funkcjonalności

Projekt został podzielony na następujące klasy:

* Klasa Flight
  + odpowiada za prawidłową reprezentację danych lotów
* Klasa Plane
  + klasa dziedziczy po klasie Flight
  + odpowiada za prawidłową reprezentację danych samolotów
  + przed utworzeniem obiektu klasy Plane konieczne jest utworzenie obiektu klasy Flight z tym samym numerem samolotu
  + obiekt nie przechowuje informacji o typie samolotu – ta generowana jest podczas wypisywania informacji o samolocie – przyjęte kryteria podziału są inspirowane faktycznym podziałem samolotów na typy (informacja znaleziona na stronie: <https://www.wyjatkowyprezent.pl/blog/najpopularniejsze-rodzaje-samolotow-co-warto-o-nich-wiedziec/>) – przyjęte typy samolotów to:
    - continental – do 100 miejsc dla pasażerów
    - narrow-body – do 200 pasażerów
    - wide-body – możliwość zabrania ponad 200 pasażerów
* Klasa Ticket
  + klasa dziedziczy po klasie Flight
  + odpowiada za prawidłową reprezentację danych biletów
  + przed utworzeniem obiektu klasy Ticket konieczne jest utworzenie obiektu klasy Flight z tym samym numerem samolotu
* Klasa Passenger
  + odpowiada za prawidłową reprezentację danych pasażerów
  + przed utworzeniem obiektu klasy Passenger konieczne jest utworzenie obiektu klasy Ticket z tym samym numerem identyfikacyjnym biletu
* Klasa Database
  + odpowiada za przechowywanie i zarządzanie danymi – wszystkimi wyżej wymienionymi klasami
* Klasa UserInterface
  + odpowiada za komunikację z użytkownikiem za pomocą menu konsolowego

Metodologia tworzenia kodu oraz testy

Podejściem projektowym było TDD – test driven development. Poza plikami ui.py oraz console\_ui.py zaczynałem od tworzenia testów a dopiero tworzony był kod. Te dwa wyjątki wynikają z charakteru znajdujących się tam funkcjonalności, które nie są oczywiste do testowania na sposób testów jednostkowych – w tych przypadkach po każdych istotnych zmianach kodu, każdej znaczącej implementacji testowałem różne warianty zachowania użytkownika po uruchomieniu programu.

Z testów został wygenerowany raport – jest to plik report.html

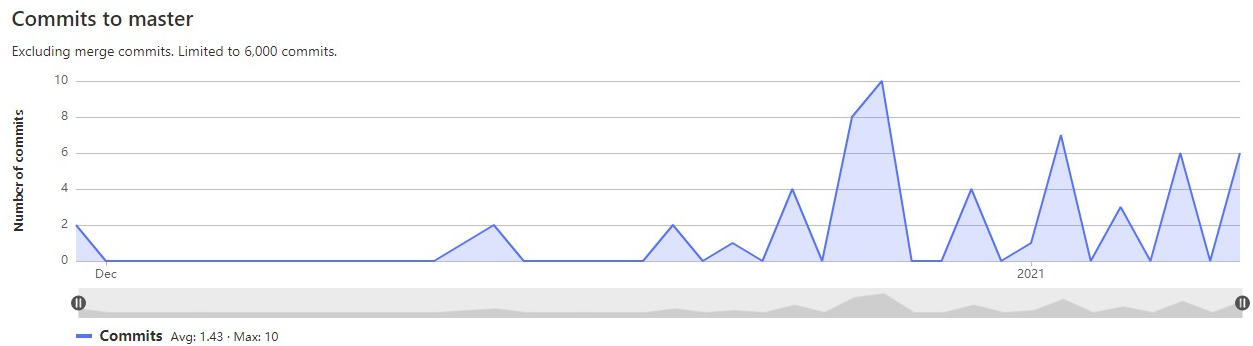
Po każdej istotnej zmianie kopia kodu wysyłana była do repozytorium – stworzony projekt na wydziałowym gitlabie:

<https://gitlab-stud.elka.pw.edu.pl/mwasilew/pipr-project.git>

Czas pracy:

Czas pracy był liczony z zaokrągleniami do 5 minut.

|  |  |  |  |
| --- | --- | --- | --- |
| Data | Czas pracy własnej [minuty] | Czas kosultacji [minuty] | Łączny czas pracy [minuty] |
| 14.12.2020 | 0 | 20 | 20 |
| 15.12.2020 | 0 | 0 | 0 |
| 16.12.2020 | 60 | 0 | 60 |
| 17.12.2020 | 100 | 0 | 100 |
| 18.12.2020 | 0 | 0 | 0 |
| 19.12.2020 | 0 | 0 | 0 |
| 20.12.2020 | 60 | 0 | 60 |
| 21.12.2020 | 0 | 0 | 0 |
| 22.12.2020 | 40 | 20 | 60 |
| 23.12.2020 | 100 | 0 | 100 |
| 24.12.2020 | 30 | 0 | 30 |
| 25.12.2020 | 40 | 0 | 40 |
| 26.12.2020 | 0 | 0 | 0 |
| 27.12.2020 | 180 | 0 | 180 |
| 28.12.2020 | 0 | 0 | 0 |
| 29.12.2020 | 270 | 35 | 305 |
| 30.12.2020 | 440 | 0 | 440 |
| 31.12.2020 | 0 | 0 | 0 |
| 01.01.2021 | 0 | 0 | 0 |
| 02.01.2021 | 145 | 0 | 145 |
| 03.01.2021 | 0 | 0 | 0 |
| 04.01.2021 | 15 | 0 | 15 |
| 05.01.2021 | 235 | 20 | 255 |
| 06.01.2021 | 0 | 0 | 0 |
| 07.01.2021 | 205 | 0 | 205 |
| 08.01.2021 | 0 | 0 | 0 |
| 09.01.2021 | 185 | 0 | 185 |
| 10.01.2021 | 10 | 0 | 10 |
| 11.01.2021 | 175 | 15 | 190 |
| razem |  | | 2400 |



*Wniosek 1. z czasu pracy: Jak widać wykres czasu pracy w sporej części odzwierciedla wykres liczby commitów w repozytorium.*

*Wniosek 2. z czasu pracy: Możliwe jest zrealizowanie projektu w jednym tygodniu roboczym.*

Wnioski:

Co bym zrobił inaczej?

* Jeszcze wcześniej zacząłbym projekt – właściwy kod zacząłem pisać w połowie / pod koniec grudnia – teraz zacząłbym od razu po otrzymaniu tematu projektu

Czego się nauczyłem?

* Planowania swojej pracy – chodzi nie tylko o czas, ale również o myślenie krok albo kilka kroków do przodu, np. żeby nie trzeba było później wprowadzać poprawek