Projekt PROI - Sklep z płytami winylowymi

Wykonanie: Bartosz Han, Igor Matynia

Table of Contents

1 Założenia

W naszym projekcie przyjęliśmy następujące założenia co do działania symulacji:

- W sklepie znajduje się jedna duża kolejka prowadząca do K stanowisk ze sprzedawcami
- Po wejściu do sklepu klient od razu wchodzi na koniec kolejki
- Klient ma zdefiniowaną ilość czasu po jakiej się nudzi i wychodzi ze sklepu
- Akcje które może wykonać klient podczas bycia obsługiwanym w stanowisku to:
  - Kupno
  - Zamówienie
  - Zapytanie o płytę - zapytanie, czy płyta jest dostępna w sklepie, czy jest dostępna na zamówienie
- Ilość akcji, a także ich rodzaje, oraz informacja, jakich płyt dotyczą, jest losowana
- Losowany jest także czas wejścia i czas po jakim znudzi się klient
- Klient może tylko tylko jedną akcję dotyczącą danej płyty (z jednym wyjątkiem opisanym niżej)
- Jeśli klient kupuje lub zamawia płytę, musi za nią zapłacić

- Jeśli klient chce kupić płytę, której nie ma sklepie, po próbie jej kupna klient automatycznie ją zamówi (będzie stworzona nowa akcja)
- Po wykonaniu akcji które chciał zrobić klient, wychodzi on od razu do sklepu

2 Działanie programu

Zanim rozpocznie się symulacja sklepu z płytami winylowymi, następuje pobranie danych plików typu JSON dotycząca: klientów, którzy odwiedzą sklep, oraz informacji o wszystkich płytach, które ‘istnieją’ w naszej symulacji. Następnie generowane są instancje: VinylDatabase, w której zawarte są wszystkie płyty (obiekty, które dziedziczą po klasie bazowej Disc, zawiera m.in. authora (obiekt klasy Author), cenę, tytuł, oraz typ dysku) wraz z ilością każdej z nich (ilość poszczególnych płyt jest losową liczbą od zera do określonej stałej), oraz VinylRecordShop, który będzie odpowiedzialny za naszą symulację. W naszej bazie danych musi być co najmniej jeden rekord o jakiejkolwiek płycie. Nasz sklep (Vinyl Record Shop) tworzy zadaną w argumencie wywołania ilość obiektów typu AssistantBooth (budki, w których, będą obsługiwani klienci), a także wektor klientów, który pobiera z pliku. Każdy klient otrzymuje losowo: czas, w którym wejdzie on do sklepu (czas ten jest z zakresie działanie sklepu), czas jaki zamierza spędzić on w sklepie, oraz ilość akcji, jaką zamierza on wykonać w sklepie. Klient posiada także stan, w jakim się znajduje (czy jeszcze nie wszedł do sklepu, jest w kolejce do sklepu, jest obsługiwany, jest znudzony, lub wyszedł już ze sklepu; stan jest reprezentowany za pomocą enuma).

Zanim rozpoczniemy symulację, w terminalu możemy nieco zmodyfikować wyświetlanie naszej symulacji (więcej o terminalu możemy znaleźć w podpunkcie 3.3). Po wpisaniu komendy “run” rozpoczynamy naszą symulację.

W czasie symulacji, nasz sklep co tick aktualizuje zarówno każdą budkę, jak i każdego klienta, podając im wartość liczbową ticku. Gdy wartość tick będzie równa wartości czas, w jakim klient a wejść do sklepu, ten wchodzi do niego i od razu wchodzi do kolejki, która jest w sklepie.

Każda budka ma informację, czy jest ona wolna. Gdy są klienci w kolce, a także jest jakaś wolna budka, klient wchodzi do niego (jest przekazywany pointer do niego do budki). Wówczas następuje jego obsługa.

Klient nie posiada żadnego kontenera o akcjach, jakie chce zrobić. Dopiero, gdy jest on w budce, zaczyna się ich generacja (a dokładniej tworzony jest shared_pointer na akcję). Akcja to obiekt klasy Action. Zawiera on informację o płycie, o której chce on wykonać, a także enum DiscAction, informującą jaką akcję chce on wykonać (czy chce płytę kupić, zamówić, czy też zapytać o podaną płytę). Zarówno płyta, jak i akcja dysku są generowane całkowicie losowo. Gdy jest tworzona nowa akcja, dekrementowany jest licznik akcji.

Gdy zostaje przekazana nowa akcja, ustawiane są odpowiednie flagi dotyczące budki, a także ustawiany jest czas trwania akcji. Czas akcji to suma pewnego stałego czasu bazowego, innego dla każdego rodzaju akcji płyty, oraz pewnej losowej liczby ze stałego zakresu. Budka następnie “obsługuje” klienta przez ten zadany czas (przez taką ilość ticków).

Gdy kończy się jedna akcja (wykonuje się metoda finishAction() w budce), jej pointer zostaje przekazany klientowi w wektorze informującym o wszystkich dokonanych przez niego akcjach. Również inne obiekty w symulacji są modyfikowane, gdy następuje koniec akcji:

- Jeśli klient chciał kupić jakąś płytę, to jeśli w bazie danych jest informacja o tym, iż płyta jest na stanie (ilośc płyty jest niezerowa), to jej ilość jest dekrementowana, a do rachunku klienta jest dodawana cena całkowita płyty z akcji (rachunek to pojedyncze pole w klasie Customer, zawierające informację to całkowitej wartości pieniężnej zakupów klienta); w przeciwnym razie, jeśli zadanej płyty nie było na stanie, do budki trafia nowa, określona akcja: “zamów płytę, której nie udało się kupić”.
- Jeśli klient chciał zamówić płytę, to jej wartość całkowita jest dodawana do rachunku klienta.
- Jeśli klient chciał zapytać o płytę, to ani w budce, ani w kliencie (oprócz dodania do wektora wykonywanych akcji) nie dzieje się nic.

Jeśli klient ma jeszcze jakieś akcje do zrobienia (tj. licznik pozostałych akcji nie jest równy zero), to przekazywana jest kolejna akcja.

Klient może wykonać maksymalnie jedną akcję na każdą płytę znajdującą się w bazie danych. Z tego powodu, jego pierwotna ilości akcji może być zmodyfikowana przez budkę tak, żeby ilość akcji klienta nie przekroczyła ilości unikalnych płyt w bazie danych. Jeśli pierwotna ilość akcji będzie przekroczona, ustawiana jest ilość akcji równa ilości płyt w bazie danych.

Klient może wyjść ze sklepu na dwa sposoby:

- Podczas obsługi ilość akcji do przekazania budce spadała do zera - wówczas w następnym ticku sklepu klient opuści tą budkę i wyjdzie ze sklepu.
- Klient tak długo stał w kolejce, że znudził się, i postanowił z niej wyjść, bez bycia obsłużonym.

W celu ukazania, co dzieje się w sklepie, już przy tworzeniu sklepu został utworzony i przekazany sklepowi obiekt klasy EventLog. Log ten zbiera informacje o wszystkim co się wydarzyło w sklepie za pomocą obiektów dziedziczących po klasie Event. Wszystkie te obiekty posiadają w sobie czas (tick) w którym stało się jakieś istotne wydarzenie. Szczegółowe informacje o poszczególnych eventach znajduje się w punkcie 4 dokumentacji, w klasie Event. Eventy są na bieżąco wyświetlane w terminalu, a także zapisywane w pliku zadanym jako argument wywołania (patrz punkt 3.2).

Cała symulacja kończy się po czasie zadanym przez użytkownika. Wówczas sklep “staje w miejscu” - już żadna akcja nie zostaje wykonana, żaden klient nie może wejść ani do sklepu, ani do budki.

3 Instrukcja obsługi

3.1 Kompilacja

Przed kompilacją należy się upewnić że w folderze repozytorium znajduje się folder “bin”. Aby skompilować program należy po pobraniu repozytorium wykonać domyślną opcję makefile, wpisując w terminal komendę

```
$ make
```

W celu kompilacji testów, należy wpisać

```
$ make testing
```

3.2 Uruchamianie

Po skompilowaniu programu należy się upewnić że ma się dostęp do plików json bazy danych płyt oraz klientów. Aby uruchomić z poziomu domyślnego katalogu repozytorium należy wykonać poniższe polecenie:

```
$ bin/main SIMULATION_TIME N_OF_BOOTHS DISC_JSON CUSTOMER_JSON CONFIG_JSON
LOG_FILE [DELAY_MS]
```

Gdzie pola kolejno oznaczają:

SIMULATION → ilość jednostek czasu wykonywania się symulacji

N_OF_BOOTHS → ilość stanowisk sprzedawców w sklepie

DISC_JSON → ścieżka do pliku z bazą danych

CUSTOMER_JSON → ścieżka do pliku z klientami

LOG_FILE → ścieżka do pliku w którym ma być zapisany log symulacji

[DELAY_MS] → opcjonalny argument zmieniający jaka ilość rzeczywistego czasu ma upływać pomiędzy kolejnymi jednostkami czasu w symulacji (w milisekundach, default = 50)

Plik z klientami powinien zawierać json’ową listę obiektów, posiadających pola “name” oraz “surname”. Plik bazy danych natomiast powinien zawierać listę opisów płyt, składających się z pól:

- “genre_id” -> ciąg znaków opisujący gatunek płyty
- “title” -> tytuł
- “price” -> cena (int)
- “author” składający się z:
    - “name”
    - “surname”

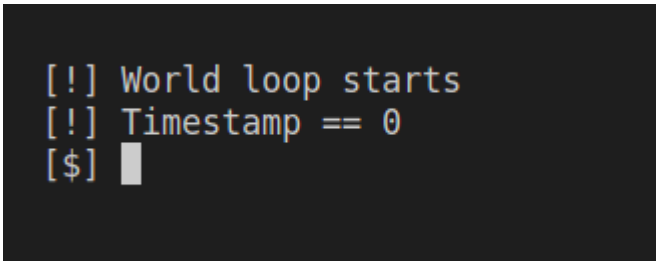
Pliki z przykładowymi danymi znajdują się w folderze “example_data”.

Aby uruchomić testy, należy w terminalu wpisać:

```
$ bin/testing
```

3.3 Obsługa terminalowego interfejsu

Po poprawnym uruchomieniu programu powinien się ukazać poniższy tekst:



W tym momencie możemy już interaktować z symulacją. Timestamp oznacza w której klatce symulacji aktualnie się znajdujemy (a konkretnie która klatka się zasymuluje po włączeniu symulacji). Dostępne komendy można wyświetlić wpisując dowolny ciąg znaków będący nieprawidłową komendą:

```
[!] World loop starts
[!] Timestamp == 0
[$] help
[!] Available commands are:
> status (stat)                   -> shop status
> booth (bth)       [SUBCOMMAND] [PARAMETERS]
> queue (qu)        [SUBCOMMAND] [PARAMETERS]
> database (db)                   -> display database info
> break (br)        TIMESTAMP     -> stop simulation at timestamp
> continue (cont)]                -> continiue untill the next breakpoint or step 1 frame
> run                             -> run til completion
> exit (q)
[$] qu help
[!] Available subcommands for queue are:
> list (ls)               -> prints all customers in queue
> print (p)       [INDEX] -> prints detailed info of a customer
[$] bth help
[!] Available subcommands for booth are:
> list (ls)               -> prints all booths
> print (p)       [INDEX] -> prints detailed info of a booth
[$]
```

Wpisując poszczególne komendy typu status, booth, queue czy database można wypisać na ekran szczegółowe informacje o poszczególnych elementach sklepu i symulacji. Komendy break, continue i run służą uruchamianiu/przerywaniu symulacji. Aby uruchomić symulację by działała do końca należy po prostu wpisać *[$] run*. Aby symulacja się przerwała w danej klatce, należy wpisać *[$]break <numer klatki>* a potem *[$] continue*. W terminalu będą się wyświetlać wydarzenia jeden po drugim w czasie rzeczywistym. Po zakończeniu programu będą też one zapisane do pliku zdefiniowanego przy wywołaniu programu.

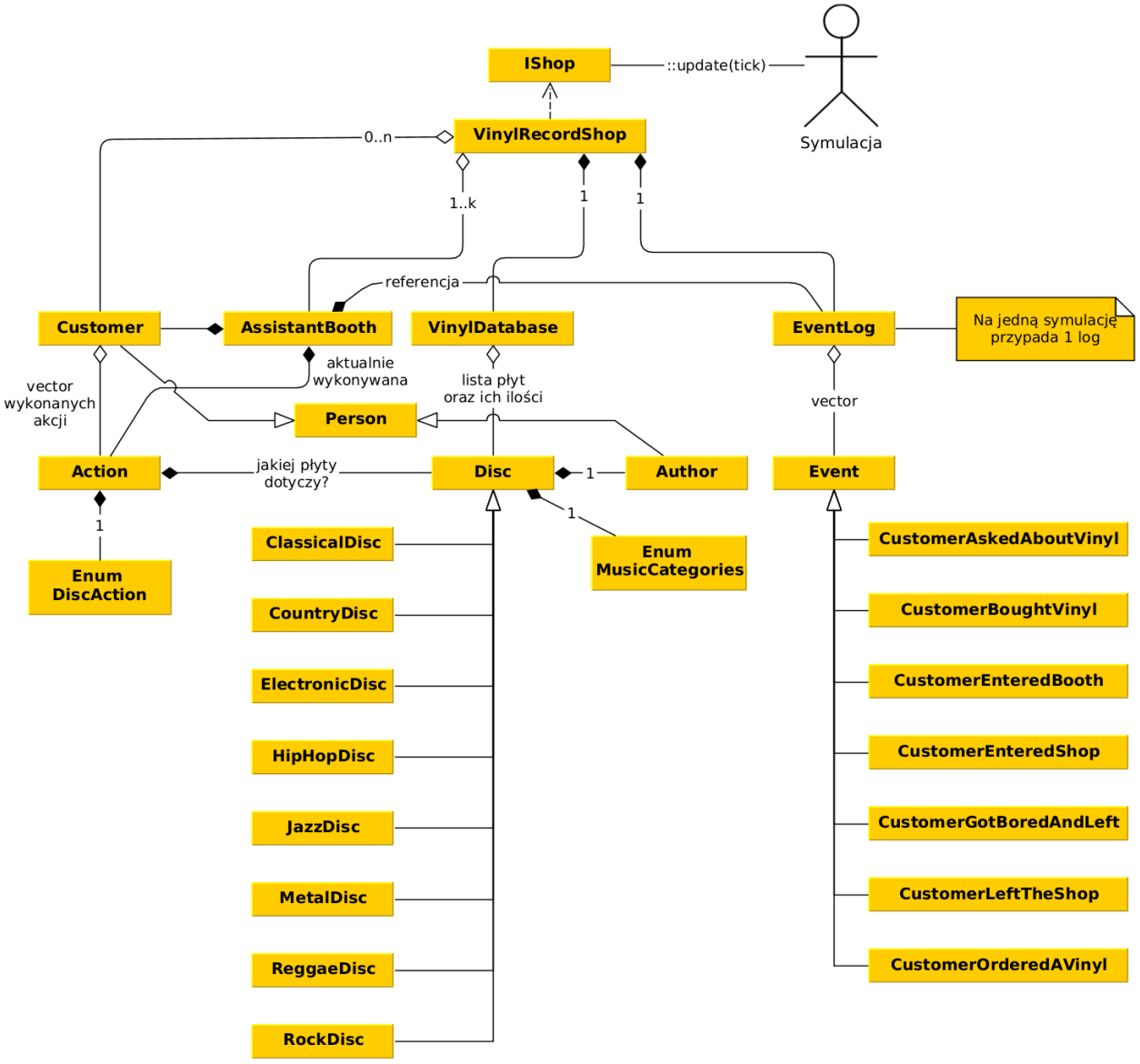
4 Klasy programu

Dokumentację doxygen (w języku angielskim) można wygenerować wykonując polecenie *$ make docs_gen* (jeśli na komputerze zainstalowany jest doxygen).

- **AssistantBooth** - Reprezentuje miejsce w którym klient może wykonywać akcje

- **Author** - Klasa reprezentująca autora Disc'a. Przechowuje tylko jego imię i nazwisko.
- **Customer** - Reprezentuje klienta w sklepie, przechowuje informacje o tym kiedy klient zamierza wejść, kiedy się znudzi w kolejce, ile akcji chce wykonać oraz ile wydał łącznie pieniędzy.
- **CustomerPtrComparator** - Klasa której jedynym zadaniem jest implementacja komparatora dla 2 obiektów std::unique_ptr< Customer >. Używana jest w strukturach STL potrzebujących komparatorów.
- **Disc** Klasa reprezentująca płytę winylową, którą może kupić Customer. Zawiera m.in autora, tytuł i cenę. Cena jest zwiększona o pewną marżę zależną od typu płyty, które są klasami pochodnymi od klasy Disc. Dostępne typy płyt:
  - CountryDisc
  - RockDisc
  - MetalDisc
  - JazzDisc
  - ElectronicDisc
  - ClassicalDisc
  - HipHopDisc
  - ReggaeDisc
- **Event** - Klasa bazowa, z której dziedziczą wszystkie eventy. Eventy są używane do zapisywania wydarzeń które miały miejsce w sklepie w **EventLog'u**. Klasy pochodne:
  - *CustomerAskedAboutVinyl* - Ma miejsce, gdy klient skończył pytać się o płytę
  - *CustomerBoughtVinyl* - Ma miejsce, gdy klient skończył kupowanie płyty
  - *CustomerEnteredBooth* - Ma miejsce gdy klient przeszedł z kolejki do stanowiska sprzedawczego
  - *CustomerEnteredShop* - Ma miejsce gdy do sklepu wchodzi klient
  - *CustomerGotBoredAndLeft* - Ma miejsce gdy klient spędza za dużo czasu w sklepie, mi musi z niego wyjść
  - *CustomerLeftTheShop* - Ma miejsce gdy klient wychodzi ze sklepu w warunkach nominalnych (czekał w kolejce i został obsłużony)
  - *CustomerOrderedAVinyl* - Ma miejsce, gdy skończył zamawiać płytę
- **EventLog** - Przechowuje i wypisuje do pliku i do konsoli wydarzenia mające miejsce w sklepie
- **TerminalIO** - Klasa terminalowego interfejsu użytkownika
- **VinylDatabase** - Klasa reprezetująca bazę danych płyt winylowych. Zawiera informację o wszystkich istniejących płytach, ilości każdej z nich w sklepie, a także jest odpowiedzialna za modyfikację ilości płyt, gdy klient chce zamierza zrobić akcję z płytą.
- **VinylRecordShop** - Klasa reprezentująca cały sklep. Zawiera kolejkę klientów, listę stanowisk kasjerskich, bazę danych płyt i log wydarzeń. Wywoływanie methody update(tick) pozwala na przeprowadzenie symulacji.
- **IShop** - Interfejs definiujący funkcje które musi obsługiwać klasa sklepu

5 Hierarchia klas



6 Wyjątki

- *DiscNotFoundError* - Błąd, który pojawia się, gdy chcemy zrobić coś z obiektem typu Disc, którego nie ma w naszej bazie danych. Pojawia się podczas kupna takiego dysku, lub uzyskania o nim informacji z bazy danych.
- *DuplicateDiscError* - Błąd pojawiający się podczas tworzenia bazy danych, gdy w naszym pliku zawierającym dane o płytach są dwa rekordy o tej samej płycie.
- *EmptyTitleError* - Błąd informujący nas, iż próbowaliśmy stworzyć obiekt klasy Disc bez nadania mu tytułu.
- *NegativePriceError* - Błąd informujący nas o tym, iż próbowaliśmy stworzyć płytę, która by miała ujemną cenę.
- *NonPositiveValueError* - Błąd pojawiający się o tym, gdy pewna stała jest mniejsza bądź równa zero.

- *EmptyDatabaseError* - Błąd pojawiający się, gdy po wczytaniu naszej bazy danych z pliku ta jest wciąż pusta.
- *InvalidMusicGenre* - Pojawia się przy próbie wykorzystania niewłaściwego ID gatunku muzyki
- *BadInputDataError* - Ma miejsce gdy podany jest nieprawidłowy tym w interfejsie terminalowym
- *ComandLineArgumentError* - Błąd informujący o nieprawidłowych argumentach wywołania programu
- *JsonMissingFieldError* - Ma miejsce gdy we wczytywanym pliku json brakuje potrzebnych pól

7 Użyte struktury STL

- std::unique_ptr - struktura unique ptr została używa w wielu miejscach w kodzie dla przetrzymywania unikalnych obiektów. Były to na przykład Customer i AssistantBooth w klasie sklepu
- std::shared_ptr - struktura ta została wykonana wszędzie tam, gdzie potrzebne było przekazanie pointera na pewien obiekt paru innym pobiektom. Stosowany w VinylDatabase, gdzie ilość płyt winylowych, oraz fakt, iż każda płyta może być inna klasą (dziedzicząca po klasie bazowej), wymagał użycia tego pointera, a także w Customerze i AssistantBooth, gdzie nie możemy od razu po zakończeniu akcji przekazać klientowi pointera na już skończoną akcję, gdyż potrzebujemy później jeszcze paru informacji o płycie z tej akcji.
- std::vector - stosowany za każdym razem, gdzie wymagane było użycie jakiejś kolekcji (budki w VinylRecordShop, skończone akcje w Customer, logi w EventLog)
- std::push_heap / std::pop_heap - funkcje pozwalające na tworzenie kopca w wybranej strukturze. Wykonywane były na wektorze kolejki Customer'ów, aby uporządkować ich wedle kolejności wchodzenia. Użyliśmy vector'a i push_heap, pop_heap zamiast kolejki priorytetowej, ponieważ nie pozwalała ona na iterację po swoich elementach.
- std::map - stosowana, gdy potrzebowaliśmy przechowywać pary jakiś minimalnie powiązanych ze sobą obiektów (w tym sensie jedyne wykorzystanie mapy to skojarzenie danej płyty winylowej wraz z jego ilością w DiscDatabase), lub w celu własnej "konwersji typów" (w naszym przypadku, zmienialiśmy stringa, który informował o gatunku muzyki na używany w kodzie gatunek muzyki - MusicCategories (używane w klasie Disc)).

8 Testowanie

Wykonano następujące testy:

Klasa **Action**: Testy, czy gettery obiektów tej klasy są zwracane odpowiednio, sprawdzenie czy operatory przyrównania odpowiednio działają.

Klasa **Author**: Sprawdzenie, czy obiekty te są odpowiednio generowanie z pliku typu JSON (jako "pseudo JSON"), sprawdzenie getterów i setterów, sprawdzenie, czy ustawianie pól jako pustych sprawia, że zwracanie pełnego imienia i nazwiska jest poprawne.

Klasa **Disc**: Sprawdzenie, czy gettery i settery poprawnie działają, sprawdzenie operatorów, sprawdzenie, czy podanie, lub ustawianie niewłaściwych danych powoduje wywołanie odpowiedniego błędu (ujemna cena, pusty tytuł w konstruktorze).

Klasy pochodne od klasy **Disc**: Sprawdzenie, czy odpowiednie typy dysków mają odpowiedni id typu, odpowiednią cenę bazową i cenę całkowitą.

Klasa **AssistaneBooth**: sprawdzenie, czy klienci odpowiednio wchodzą do budki, czy są odpowiednio obsługiwani i czy są ustawiane odpowiednie flagi, sprawdzenie, czy obsługiwanie akcji jest poprawne, sprawdzenie czy po zakończeniu obsługiwanej akcji (czy następna akcja po poprzedniej jest taka, jak powinna być), sprawdzenie, czy po zakończeniu obsługi danego klienta następuje poprawne działanie programu i flag,

Klasa **Customer**: sprawdzenie getterów, setterów, pieniędzy na rachunku oraz systemu stanów.

Klasa **EventLog**: test sprawdzający czy dodawanie event'u wpisuje go na stream.

Klasa **Event**: test sprawdzający czy timestamp jest właściwie zapisany oraz czy działa operator wypisywania na stream.

Klasa **VinylRecordShop**: ze względu na wbudowaną losowość swojego operowania, testy są ograniczone do setterów i getterów oraz dodawania klienta do kolejki

Klasa **VinylDatabase**: Sprawdzenie, czy baza danych odpowiednio wczytuje dane, sprawdzenie, czy wyszukiwanie płyt, ich ilości i sprawdzenie dostępności płyty winylowej jest poprawne, sprawdzenie getterów do dysków, sprawdzenie, czy próba kupienia płyty odpowiednio modyfikuje bazę danych, sprawdzenie, czy złe dane argumenty w funkcji powoduje wywołanie odpowiednich błędów (gdy w bazie danych są dwa rekordy o tej samej płycie, gdy w bazie danych nie ma żadnej płyty, gdy próbujemy kupić lub zdobyć informację o płycie winylowej, której nie ma w naszej bazie danych).

9 Podział pracy

Bartosz Han - stworzenie klasy o bazie danych o płytach jak też klasy o samej płycie (i klas pochodnych do płyty), stworzenie logiki kupna płyty, stworzenie klasy reprezentującą akcję, stworzenie logiki odpowiedzialnej za obsługę klienta (odpowiednie ustawianie flag, przekazywanie i generowanie akcji, ustawianie czasu trwania akcji, obsługa skrajnych przypadków) w klasie AssistaneBooth, stworzenie i implementacja klasy Person, po której dziedziczą Customer i Author, dodanie randomizacji do obiektów w symulacji, stworzenie paru klas dziedziczących po klasie Event (głównie eventy dotyczące zakończenia jakiejś akcji przez klienta), stworzenie odpowiednich wyjątków do bazy danych i do dysku, oraz testowanie tych klas (w przypadku dysku wraz z klasami pochodnymi) wraz z testowaniem klas AssistantBooth i Action.

Igor Matynia - stworzenie deklaracji klasy AssistantBooth, stworzenie klasy Customer, wraz z logiką odpowiedzią za odpowiednie wchodzenie, "nudzenie się" i wychodzenie klienta ze sklepu, stworzenie interfejsu sklepu, jak i logiki odpowiedzialnej za poprawnie jego działanie (stworzenie klasy VinylRecoredShop), obsługa plików JSON,

stworzenie graficznego interfejsu dla użytkownika, stworzenie poprawnego wczytywania argumentów wywołań, stworzenie loga eventów, wraz ze stworzeniem klasy Bazowej Event oraz dodanie paru klas dziedziczących po Evencie (głównie dotyczących klienta), stworzenie wyjątków, głównie dotyczących klienta i złego wywołania programu z terminala.

Wspólnie - wymyślenie założeń projektu, pisanie dokumentacji.

10 Podsumowanie

Praca nad projektem zaczęła się w połowie kwietnia 2022 r. W ramach projektu zostało wykonanych łącznie ok. 170 commitów. Stworzyliśmy ok. 15 klas (nie licząc klas dziedziczących całkowicie po innych klasach), oraz 9 plików z testami (nie licząc plików z testami klas, które dziedziczą po innych klasach).

Celem projektu było praktycznie wykorzystanie funkcji języka C++. Dzięki temu projektowi nauczyliśmy się wielu różnych aspektów tego języka m.in. odpowiednie tworzenie i operowanie kontenerami, tworzenie i obsługiwanie inteligentnych wskaźników, czy tworzenie przeciążeń operatorów do poszczególnych klas, tworzenie wyjątków, tworzenie programu z argumentami wywołań i tworzenie interfejsu między programem a użytkownikiem. Poznaliśmy obsługę poszczególnych aspektów języka i nauczyliśmy się je odpowiednio wykorzystywać.

Dodatkowo, był to nasz pierwszych tak duży projekt informatyczny, który robiliśmy we dwie osoby. Dzięki niemu, nauczyliśmy się pracować w duecie, odpowiednio komunikować się i dzielić się sugestiami na temat projektu, a także odpowiedniego obsługiwania GitLaba, zwłaszcza w sytuacjach gdy musieliśmy mergować wspólnie zmodyfikowany kod.

Uważamy, iż wszystkie założenia, które stworzyliśmy przed przystąpieniem do projektu zostały zrealizowane, czy to w całości, czy też były one modyfikowane, czasami nawet dość mocno.