Projekt

Ksawery Chodyniecki
Paweł Müller

# Symulacja supermarketu

Projekt nr 8

# 1. Przyjęte założenia.

**W naszym projekcie przyjęliśmy następujące założenia:**

- w sklepie pracuje n pracowników,
  - pracownik może:
    - obsługiwać klientów przy kasie,
    - obsługiwać klientów na hali supermarketu,
- w sklepie jest m kas,
- sklep odwiedza x klientów,
  - klient może:
    - wejść do sklepu,
    - włożyć do swojego koszyka produkt z półki
    - zapytać pracownika o szczegóły dotyczące danego produktu
    - stanąć w kolejce do kasy
    - rozliczyć się w kasie
    - opuścić sklep
  - każdy rozliczony klient dostaje rachunek (paragon lub fakturę, w zależności, od tego czy jest klientem biznesowym czy indywidualnym), który zapisuje się wraz z logami do plików.

# 2. Sposób przeprowadzania testów - symulacja.

**Ważne pojęcia:** sekundę symulacji definiujemy jako liczbę wydarzeń na sekundę dzieloną na 2 (jest to zrobione aby dało się przeczytać co się dzieje w symulacji). Czas symulacji jest więc podawany w sekundach symulacji, a nie prawdziwych sekundach. Liczba wydarzeń również nie gwarantuje, że wykona się dokładnie tyle wydarzeń, ponieważ program może nie wykonać jednego z wydarzeń aby uniknąć późniejszych błędów.

Symulacja najpierw tworzy wszystkie niezbędne do uruchomienia obiekty, uzależniając ich ilość od zadanych parametrów. Parametry mogą być podane przez konsolę (wymagane jest podanie długości symulacji oraz ilości wydarzeń dziejących się na sekundę symulacji), mogą być wczytane z pliku (podając nazwę pliku) albo mogą nie zostać podane wcale - wtedy program przyjmie wartości domyślne, to znaczy: długość symulacji równą 20 sekund oraz liczbę wydarzeń równą 3.

Po uruchomieniu program wykonuje w pętli różne wydarzenia w sklepie. Na rodzaj wydarzenia ma wpływ stosunek ilości klientów znajdujących się w sklepie do maksymalnej ich ilości oraz przypadkowo wygenerowana liczba. Następnie na konsolę

wypisywany jest opis tego wydarzenia; generowany jest także plik `log.txt`. Po wykonaniu odpowiedniej ilości wydarzeń w danej sekundzie, program sprawdza wszystkie otwarte kasy, skanując x produktów z koszyka pierwszego klienta w kolejce. Jeśli zeskanowany zostanie ostatni produkt, to podsumowuje należność i wystawia fakturę lub rachunek, który potem zapisuje do pliku odpowiednio jako `Invoice_{numer}.txt` lub `Receipt_{numer}.txt`.

# 3. Opis klas, ich atrybutów i metod.

## klasa Shop

Obiekt zbierający wszystkie pozostałe obiekty w jedną, zgrabną paczkę

**Atrybuty:**

- Container'y obiektów Klient, Kasa, Pracownik i Produkt
- wektor CustData - trzyma w sobie dane do tworzenia klientów
- maxTime - długość symulacji
- eventsPerTick - ilość zdarzeń wydarzająca się w sklepie w ciągu sekundy
- billNumber - zmienna pomocnicza do tworzenia rachunków i faktur

**Metody:**

- run() - wywołuje generate(), odpala symulację (która w pętli wywołuje event()) i wypisuje jej przebieg do konsoli i do pliku
- event()
- executeQueues()
- generate()

## struktura Container

Jej obiekt ma trzymać w sobie wskaźniki na wszystkie obiekty klasy danej jej przez template oraz inne potrzebne do działania programu atrybuty związane z tymi obiektami

**Atrybuty:**

- wektor wskaźników na wszystkie istniejące obiekty danej klasy
- wektor wskaźników na wszystkie aktywne obiekty tej klasy (zdolne do akcji, na przykład otwarte kasy albo produkty, które jeszcze są w magazynie)
- iterator - nieujemna wartość całkowita pomagająca w tworzeniu unikalnych identyfikatorów
- maxAmount - nadana maksymalna ilość obiektów (tzn. jeśli będzie już tyle obiektów to symulacja nie będzie tworzyć ich więcej)

## struktura CustData

Jej obiekt ma trzymać w sobie dane wczytane z pliku potrzebne do tworzenia klientów w trakcie symulacji.

# klasa Product

Obiekt reprezentuje pojedynczy produkt.

**Atrybuty:**

- name - nazwa produktu
- ID - unikalny identyfikator produktu
- price - cena netto produktu podawana w groszach
- VAT - wartość podatku VAT wyrażona w procentach
- quantity- ilość towaru w magazynie (zmniejszana, gdy klient zapłaci za zakupy) podawana w sztukach lub gramach
- measureUnits - wartość typu wyliczeniowego, definiuje jednostkę miary: sztuka lub gram

**Metody:**

- getName() - zwraca nazwę produktu
- getID() - zwraca identyfikator produktu
- getPrice() - zwraca cenę produktu
- getVAT() - zwraca wartość VAT produktu
- getQuantity() - zwraca ilość produktu
- getMeasureUnits() - zwraca jednostkę miary produktu
- setName() - ustawia nową nazwę produktu
- setID() - ustawia nowy identyfikator produktu
- setPrice() - ustawia nową cenę produktu
- setVAT() - ustawia nową wartość VAT produktu
- setQuantity() - ustawia nową ilość produktu
- addQuantity() - dodaje podaną ilość produktu
- decQuantity() - dekrementuje ilość produktu
- calculatePriceBrutto() - oblicza i zwraca cenę brutto produktu

# klasa Customer

Obiekt reprezentuje klienta.

**Atrybuty:**

- ID - unikalny identyfikator klienta
- isBusiness - status biznesowy klienta
- name - nazwa klienta
- taxNumber - numer identyfikacji podatkowej klienta
- street - ulica klienta
- buildingNumber - numer budynku klienta
- postcode - kod pocztowy klienta
- city - miasto klienta
- country - kraj klienta
- basket - koszyk klienta (map<Product*, unsigned short>)

**Metody:**

- getID() - zwraca identyfikator klienta
- getIsBusiness() - zwraca status biznesowy klienta
- getName() - zwraca nazwę klienta
- getTaxNumber() - zwraca numer identyfikacji podatkowej klienta
- getStreet() - zwraca ulicę klienta
- getBuildingNumber() - zwraca numer budynku klienta

- getPostcode() - zwraca kod pocztowy klienta
- getCity() - zwraca miasto klienta
- getCountry() - zwraca kraj klienta
- getBasket() - zwraca koszyk klienta
- getBasketSize() - zwraca rozmiar koszyka klienta
- setID(unsigned int) - ustawia identyfikator klienta
- setIsBusiness(bool) - ustawia status biznesowy klienta
- setName(string) - ustawia nazwę klienta
- setTaxNumber(string) - ustawia numer identyfikacji podatkowej klienta
- setStreet(string) - ustawia ulicę klienta
- setBuildingNumber(string) - ustawia numer budynku klienta
- setPostcode(string) - ustawia kod pocztowy klienta
- setCity(string) - ustawia miasto klienta
- setCountry(string) - ustawia kraj klienta
- addToBasket() - dodaje produkt do koszyka
- removeFromBasket() - usuwa produkt z koszyka
- clearBasket() - czyści koszyk klienta

# klasa Employee

Obiekt reprezentuje pracownika.

**Atrybuty:**

- ID - unikalny identyfikator pracownika
- name - nazwa pracownika
- bool isOccupied - czy pracownik jest zajęty w tej chwili, czy nie
- scanSpeed - ilość produktów skanowanych na sekundę kiedy ten pracownik jest na kasie

**Metody:**

- getID() - zwraca identyfikator pracownika
- getName() - zwraca nazwę pracownika
- getScanScpeed - zwraca scanSpeed
- isOccupied() - zwraca wartość atrybutu o tej samej nazwie
- setFree() - ustawia isOccupied na fałsz
- setBusy() - ustawia isOccupied na prawdę

# klasa Bill

Obiekt będzie reprezentował każdy rachunek powstający w sklepie.

**Atrybuty:**

- date - data wystawienia rachunku
- ID - unikalny identyfikator rachunku
- buyer - obiekt klasy Customer, reprezentujący kupującego
- seller - obiekt klasy Customer, reprezentujący sprzedającego (sklep)

**Metody:**

- getDate() - zwraca datę wystawienia rachunku
- getID() - zwraca identyfikator rachunku
- getBuyer() - zwraca obiekt kupującego
- getSeller() - zwraca obiekt sprzedającego
- setID(newID) - ustawia identyfikator rachunku

- setDate(newDate) - ustawia datę wystawienia rachunku
- setBuyer(newBuyer) - ustawia kupującego
- setSeller(newSeller) - ustawia sprzedającego
- convertPricePLN(unsigend int price) - zwraca string przedstawiający cenę w złotówkach
- convertToKg(unsigned int quantity) - zwraca string przedstawiający ilość produktu (biorąc pod uwagę jednostkę miary)
- save(string filename) - zapisuje graficzną reprezentację rachunku do pliku

## klasa Invoice

Dziedzicząca po klasie Rachunek

**Metody:**

- generate() - zwraca string reprezentujący fakturę w graficznej formie

## klasa Receipt

Dziedzicząca po klasie Rachunek

**Metody:**

- generate() - zwraca string reprezentujący paragon w graficznej formie

# klasa CashDesk

Obiekt będzie reprezentował pojedyncze stanowisko kasowe razem z kolejką do niej.

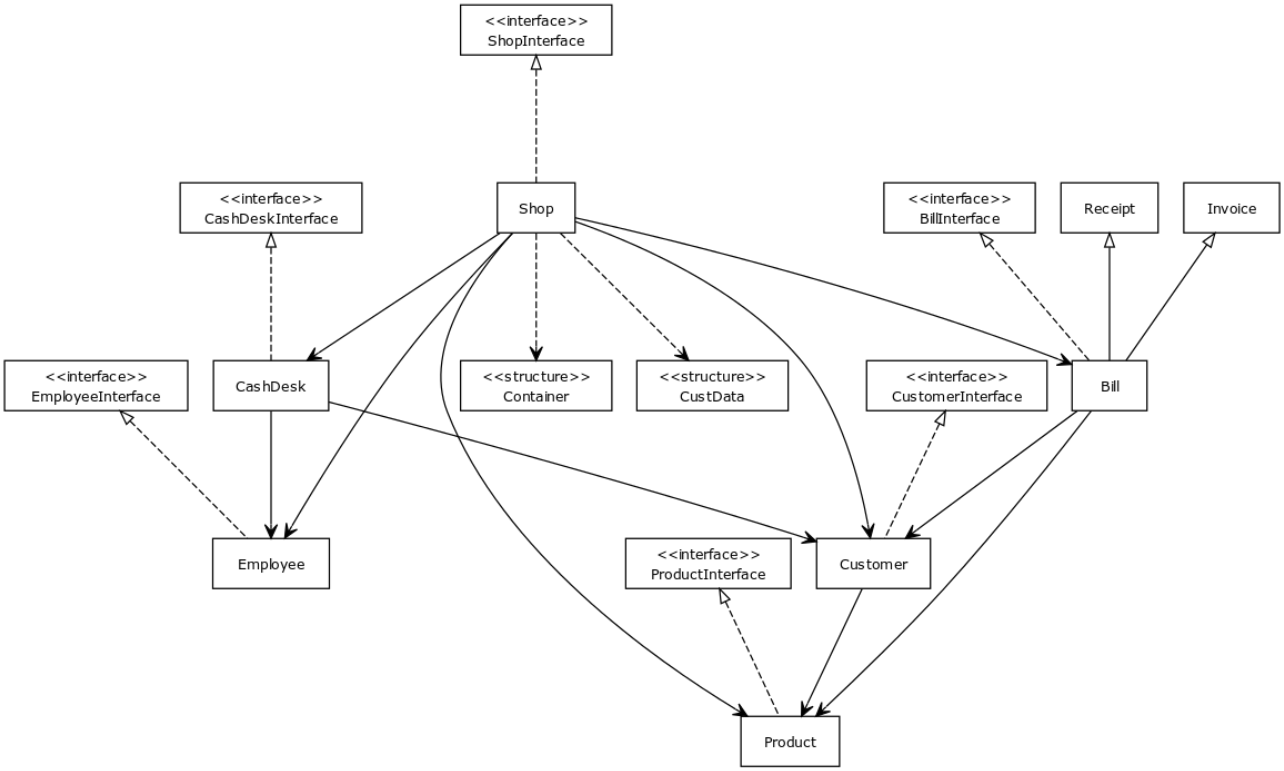
**Atrybuty:**

- ID - unikalny identyfikator kasy
- std::queue customerQueue - kolejki do kasy, zawierająca wskaźniki na obiekty Klient
- wskaźnik na obiekt Pracownik, który na tej kasie pracuje w tej chwili
- bool isOpen - czy jest otwarta ta kasa
- cashAmount - ilość pieniędzy w kasie w groszach
- itemsScaned - zmienna mówiąca ile w tym momencie zostało już ze skanowanych produktów pierwszego klienta w kolejce

**Metody:**

- getID() - zwraca identyfikator kasy
- open(), close() i getState() - funkcję manipulujące atrybutem isOpen
- getName() i setName()
- getCash(), addCash(), setCash(), takeCash() - funkcję manipulujące ilością pieniędzy w kasie
- assign() - funkcja przypisująca danego jej pracownika do kasownika (zwróci wskaźnik na obiekt Pracownik jeśli ktoś już był na tej kasie)
- size(), push(), pop() - funkcję dodające klientów, usuwające ich oraz zwracające ich ilość w kolejce
- scan() - funkcja inkrementująca atrybut itemsScanned oraz wywołująca pop() jeśli wszystkie produkty pierwszego klienta zostały zeskanowane
- payment() - podsumowuje należność klienta, dodając ją zawartości pieniędzy w kasie oraz ją zwracając

# 4. Relacje pomiędzy klasami.




# 5. Zidentyfikowane sytuacje wyjątkowe.

Podczas pracy nad projektem wyróżniliśmy wiele miejsc, w których mogłoby dojść do sytuacji, kiedy program nie zachowuje się poprawnie lub wywołuje błędy. Aby do tego nie dopuścić, zaimplementowaliśmy różnorakie mechanizmy ochrony, głównie takie, które same naprawiają błędne dane, lecz nie przerywają działania programu.

Wybrane sytuacje wyjątkowe obsługiwane w programie:

- sprawdzanie ilości argumentów wprowadzanych podczas uruchamiania programu
  - gdy nie podano żadnego argumentu, symulacja rozpoczyna się z domyślnymi parametrami początkowymi,
  - gdy podano jeden argument przyjmujemy, że jest to nazwa pliku, z którego należy odczytać parametry początkowe z pliku - sprawdzane jest również, czy istnieje taki plik,
  - gdy podano więcej, niż dwa wymagane argumenty, wywoływany jest błąd,

- sprawdzanie poprawności podawanych przez użytkownika parametrów pod kątem zgodności z przewidywanym typem danych
  - w przypadku podania danych w niepoprawnym formacie wywoływany jest błąd
- wkładanie produktu do koszyka
  - gdy klient wylosuje, że chce kupić większą ilość produktu, niż dostępna na półce, to zabiera cały asortyment,
  - gdy klient wylosuje, że chce kupić mniej niż 100g produktu sprzedawanego na wagę, to nie decyduje się na zakup (mogło to prowadzić do patologicznych sytuacji, w której klient chce kupić 2g cukierków),
- zawsze jedna z kas w sklepie jest otwarta, by klienci mogli swobodnie finalizować swoje zakupy,
- klient nie dostaje pustego paragonu, gdy nic nie kupuje,
- ilość produktów oraz ich cena przechowywana jest jako liczba całkowita, aby uniknąć braku precyzji przy obliczeniach na liczbach zmiennoprzecinkowych (jedynie podczas wyświetlania ich na ekranie lub rachunkach dokonywana jest operacja konwersji.

# 6. Użyte elementy biblioteki standardowej.

Przy realizacji projektu niezbędne okazały się nastepujące funkcje zdefiniowane w poniższych bibliotekach:

- <cstdlib>
  - rand
  - srand
- <ostream>
  - ostream
  - endl
- <iostream>
  - cout
  - cin
- <fstream>
  - ifstream
  - ofstream
- <sstream>
  - stringstream
- <string>
  - string
  - to_string
  - stoi
- <utility>
  - pair
- <queue>
  - queue
- <map>
  - map
- <cmath>
  - ceil
  - sin
- <ctime>
  - time
- <chrono>
  - time_point<high_resolution_clock>
  - seconds
- <thread>
  - this_thread::sleep_for