Dokumentacja Symulacji Supermarketu (Grupa 1.8)

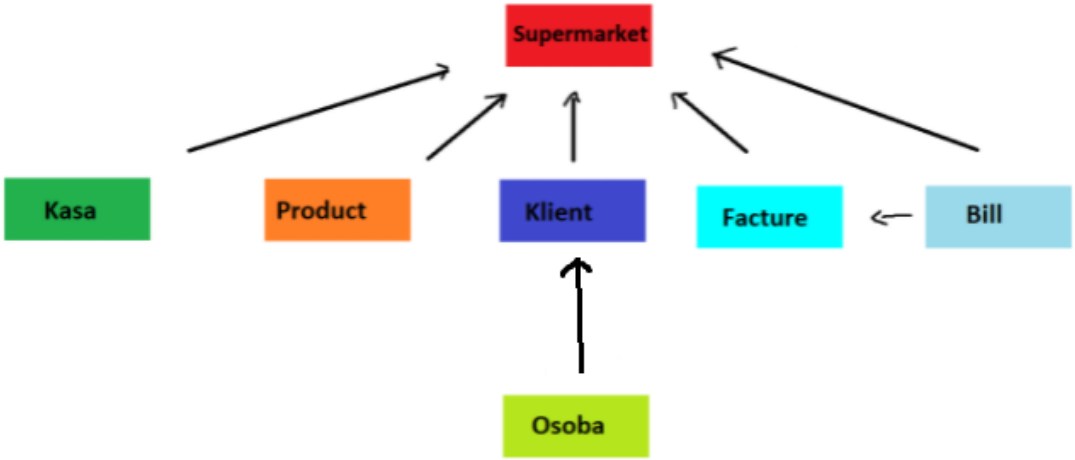
Wykonali:
Grzegorz Dzieniszewski
Hubert Sobieski
Adam Misiak

Spis treści tematycznych:

Punkt 1

Informacje o dołączonych plikach (klasy, pliki tekstowe) oraz ich funkcjonowanie

Klasa główna programu (supermarket) składa się z wielu podklas:
Tak przedstawia się graf klas:



W skład klasy supermarket wchodzą klasy:
-Kasa
-Product
-Klient (dziedziczy po klasie osoba)
-Facture (dziedziczy po klasie Bill)
-Bill
Oraz pośrednio klasa:
-Osoba (służy jedynie do konstrukcji klienta)

Atrybuty oraz metody poszczególnych klas:

1 a) Kasa:

Skrócony opis klasy:
Klasa kasy reprezentuje każdą z kas znajdujących się w supermarkecie. Przy każdej z kas obsługiwani są klienci sklepu (klasa klient) ustawiający się w jednej głównej kolejce do kas, mogą być one otwierane i zamykane w zależności od czasu ich otwarcia. Charakteryzują się numerem, dzięki któremu są one w naszym programie rozróżniane. Konstruowane są na początku symulacji.

Atrybuty:

```
int num, money, tury_pracy, tury_przerwy;
bool isopen;
```

num: liczba całkowita reprezentująca numer kasy w sklepie
money: liczba całkowita reprezentująca pieniądze w kasie
tury_pracy: liczba całkowita reprezentująca czas otwarcia kasy wyrażony w turach
tury_przerwy: liczba całkowita reprezentująca czas zamknięcia kasy wyrażony w turach
isopen: zmienna typu bool informująca czy kasa jest otwarta (1 - otwarta, 2 - zamknięta)

Metody:

Konstruktory:

Kasa() - konstruktor domyślny ustawiający zadane atrybuty na wartość 0
Kasa(int n, int cash, bool o) - konstruktor klasy, ustawiający atrybut num na wartość n, money na cash, isopen na o, tury_pracy oraz tury_przerwy ustawiane są na wartość 0

Gettery:

getnum() - Zwraca wartość atrybutu num klasy, czyli jej numer

getmoney() - zwraca ilość pieniedzy znajdujących się w kasie
get_tury_pracy() - zwraca ilość tur podczas, których kasa była otwarta podczas obecnego czasu pracy
get_tury_przerwy() - zwraca ilość tur podczas, których kasa była zamknięta podczas obecnej przerwy
isKasaopened() - zwraca wartość atrybutu isopen, czyli czy kasa jest otwarta

Settery:

setnum - ustawia numer kasy na podany w argumencie
setmoney - ustawia ilość pieniędzy w kasie na podaną w argumencie
openKasa() - otwiera kasę, zmienia wartość isopen na “true”
closeKasa() - zamyka kasę, zmienia wartość isopen na “false”
clear_tury_pracy() - ustawia ilość tur pracy na 0
clear_tury_przerwy() - ustawia ilość tur przerwy na 0

Pozostałe metody:

increment_tury_pracy() - zwiększa ilość tur pracy o 1
increment_tury_przerwy() - zwiększa ilość tur przerwy o 1
break_ended() - sprawdza czy kasa powinna być otwarta
needs_break() - sprawdza czy kasa powinna byc zamknieta
change_Kasa_status() - w zależności od zwracanych wartości przez metody needs_break() oraz break_ended() zmienia status kasy

1 b) Klient : Osoba

Skrócony opis:

Klasa klient reprezentuje klienta supermarketu, każdy klient posiada wartości odczytywane z pliku:

```
vector<tuple<Product, int>> listazak;
vector<tuple<Product, int>> items;
int money;
bool want_facture;
string adress;
string post_code;
string town;
int items_amount;
```

-**Imię -** odziedziczone po **Osobie**
-**Nazwisko -** odziedziczone po **Osobie**
-**Liczbę pieniędzy -** money
-**Liczbę przedmiotów** znajdujących się na liście zakupów - **items_amount**
-**Zmienną reprezentującą czy klient chce fakturę czy paragon - want_facture**
-**Ulicę oraz numer domu - adress**
-**Kod pocztowy - post_code**
-**Miasto - town**

Co więcej każdy klient posiada także dwa wektory:
-jeden reprezentujący listę zakupów - **listazak**
-drugi reprezentujący przedmioty znajdujące się w koszyku - **items**

W celu prostej integracji z programem klasa klienta ma dużą ilość metod:
Konstruktor - przyjmuje parametry (name, surname,money,amount_of_items, want_facture, adress, post_code,town)

Dekonstruktor - nie przyjmuje szczególnej roli

Gettery:

GetAdress() - zwraca adres klienta
GetPostCode() - zwraca kod pocztowy klienta
GetTown() - zwraca nazwę miasta klienta
GetMoney() - zwraca ilość pieniędzy użytkownika
WantFacture() - zwraca wartość bool oznaczającą chęć posiadania faktury
GetItemsAmount() - zwraca ilość produktów, jakie klient ma zadeklarowaną na swojej liście zakupów

Inne metody:
AddToPurchaseList() - dodaje produkt na vector reprezentujący listę zakupów
ReadFromPurchaseList() - zwraca vector listy zakupów
AddToCart() - przenosi pierwszy produkt z listy zakupów do koszyka
RemoveFromPurchaseList() - usuwa element z listy zakupów
EmptyCart() - czyści koszyk klienta
SubtractMoney() - odejmuje pieniądze od konta klienta

1 c) Klasa Osoba

Skrócony opis:
Klasa ta służy do tworzenia klasy Klient, która po niej dziedziczy, posiada ona atrybuty jakimi charakteryzuje się każda osoba:

```
string name;
string surna
me;
```

-**Imię - name**
-**Nazwisko - surname**

Klasa ta posiada również metody służące do otrzymania atrybutów, które są prywatne:

-**GetName()** - zwraca nam imię osoby
-**GetSurname()** - zwraca nazwisko osoby

1 d) Bill

Skrócony opis klasy:
Klasa Bill odpowiedzialna jest za prawidłowe wyświetlanie paragonów. Dla każdego z klientów, który ma odpowiednią ilość pieniędzy i zadeklarował brak chęci wzięcia faktury, zostaje wygenerowany paragon na którym znajdują się wszystkie produkty które znalazły się w koszyku danego klienta.

Atrybuty:

```
    static int previous_number;
    int number, counter_number;
    string seller_street, seller_zip, seller_town, seller_name;
    vector<tuple<Product, int>> items;
```

previous_number - reprezentuje numer poprzedniego paragonu, na jego podstawie generowany jest numer obecnego paragonu;
number - numer paragonu;
counter_number - numer kasy dla której generowany jest paragon;
seller_street, seller_zip, seller_town, seller_name - atrybuty reprezentujące adres oraz nazwę sprzedawcy;
items - jest to vector który przechowuje informacje o produktach oraz ilości danego produktu zakupionego przez klienta.

Metody:
Bill() - konstruktor domyślny, ustawiający numer kasy na 0;
Bill(int counter_number, vector<tuple<Product, int>> items, string seller_street = "Ziolowa 69", string seller_zip = "00-321", string seller_town = "Warszawa", string seller_name = "Ceplusik") - konstruktor klasy, ustawiający atrybuty klasy na wartości podanych parametrów. Konstruktor posiada wartości domyślne dla atrybutów reprezentujących adres oraz nazwę sprzedawcy.

get_number() - metoda zwraca liczbę całkowitą będącą numerem paragonu;
get_counter_number() - metoda zwraca liczbę całkowitą będącą numerem kasy;
brutto_price() - metoda zwraca liczbę rzeczywistą, bedącą całkowitą ceną produktów na które opiewa dana faktura;
get_date() - metoda zwracająca aktualną datę w formacie dzień/miesiąc/rok;
get_seller_street() - metoda zwracająca ulicę sprzedawcy;
get_seller_zip() - metoda zwracająca kod pocztowy sprzedawcy;
get_seller_town() - metoda zwracająca miasto sprzedawcy;
get_seller_name() - metoda zwracająca nazwę sprzedawcy;
item_price_gr(int item_index) - metoda ta jako parametr wejściowy przyjmuje indeks elementu znajdującego się w vector items. Metoda zwraca cenę całkowitą danego produktu w groszach;
display_items_list() - metoda ta wyświetla na ekran listę produktów z następującymi ich parametrami: nazwa produktu, zakupiona ilość, cena jednostkowa (z naliczonym podatku) oraz całkowita cena za dany produkt;
display_bill() - metoda ta wyświetla na ekran zawartość paragonu. Wyświetlane są: jego numer, data, dane sklepu, numer kasy, lista produktów (przy pomocy funkcji display_item_list(), oraz cena całkowita za wszystkie produkty;
set_counter_number(int new_number) - metoda która przypisuje parametrowi counter_number nową wartość - new_number;

1 e) Facture : Bill
Skrócony opis klasy:
Klasa Facture reprezentuje fakturę wystawioną dla klienta który zadeklarował chęć wzięcia faktury. Klasa ta dziedziczy po klasie Bill.
Atrybuty:
Oprócz atrybutów posiadanych przez obiekt klasy Bill, obiekt klasy Facture posiada dane atrybuty:

```
    static int previous_facture_number;
```

```
    int facture_number;
    std::string id, buyer_street, buyer_zip, buyer_town, buyer_name,
place_of_issue;
```

previous_facture_number - numer poprzednio wystawionej faktury;
facture_number - numer faktury;
id - jest to numer faktury oraz data jej wystawienia;
buyer_street, buyer_zip, buyer_town, buyer_name - atrybuty reprezentujące adres oraz nazwę kupującego;
place_of_issue - miejsce wystawienia faktury;

Metody:
Oprócz metod dziedziczonych po klasie Bill, klasa Facture posiada dane metody:
Facture(int counter_number, vector<tuple<Product, int>> items, string buyer_street, string buyer_zip, string buyer_town, string buyer_name, string seller_street = "Ziolowa 69", string seller_zip = "00-321", string seller_town = "Warszawa", string seller_name = "Ceplusik") - konstruktor klasy Facture, ustawiający atrybuty klasy na wartości podanych parametrów. Konstruktor posiada wartości domyślne dla atrybutów reprezentujących adres oraz nazwę sprzedawcy. Ponatdo w konstruktorze wartość atrybutu place_of_issue ustawiana jest na wartość podanej wartości seller_town;
get_facture_number() - zwraca liczbę całkowitą będącą numerem faktury;
get_id() - zwraca string reprezentujący id faktury (tzn. jej numer wraz z datą);
get_buyer_street() - metoda zwracająca ulicę kupującego;
get_buyer_zip() - metoda zwracająca kod pocztowy kupującego;
get_buyer_town() - metoda zwracająca miasto kupującego;
get_buyer_name() - metoda zwracająca nazwę kupującego;
get_place_of_issue() - metoda zwracająca miejsce wydania faktury;

display_items_list() - metoda ta wyświetla na ekran listę produktów z następującymi ich parametrami: nazwa produktu, zakupiona ilość, cena jednostkowa (bez naliczonego podatku), cena całkowita (bez naliczonego podatku), klasa podatkowa oraz całkowita cena za dany produkt (wraz z naliczonym podatkiem);
display_bill() - metoda ta wyświetla na ekran zawartość faktury. Wyświetlane są: jego numer, data, miejsce wydania, dane sklepu, dane klienta, lista produktów (przy pomocy funkcji **display_item_list()**, oraz cena całkowita za wszystkie produkty;

1 f) Product

Skrócony opis klasy:
Klasa Product reprezentuję produkt znajdujący się w supermarkecie. Każdy obiekt klasy Product posiada wartości atrybutów wczytywane z plików:

- **nazwa;**
- **klasa podatkowa (w procentach);**
- **cena (w groszach);**
- **ilość produktu znajdującego się w sklepie.**

Atrybuty:

```
    int amount, tax_class, price;
    string name;
```

amount - ilość danego produktu w sklepie;
tax_class - klasa podatkowa podana w procentach;
price - cena jednostkowa;
name - nazwa produktu

Metody:
Product() - konstruktor domyślny przypisujący wszystkim atrybutom całkowitoliczbowym wartość 0, a atrybutowi name przypisuje pusty string;

Product(int p, int t, string n, int a = 0) - konstruktor, przypisuję wartość p do atrybutu price, wartość t do atrybutu tax_class, wartość n do atrybutu name, a wartość a do atrybutu amount; **modife_amount(int new_amount)** - metoda służąca do zmiany wartości przypisanej do atrybutu amount.

Załączone pliki tekstowe:

W celu przeprowadzenia symulacji informacje o klientach są odczytywane z pliku. Tak przedstawia się linia zawierająca dane jednego klienta:

Gustaw Szewczyk 75 2 1 Skośna 13 00-375 Warszawa

Są to po kolei oddzielone spacjami:
-Imię
-Nazwisko
-Ilość posiadanych pieniędzy
-Liczba produktów które klient ma na liście (czyli np 2 produkty to szampon i mydło, każde z nich może wystąpić wiele razy np. Szampon x3 mydło x 6)
-Wartość 0 lub 1 reprezentująca czy klient chce fakturę (0 nie chce, 1 chce)
-Ulica na której mieszka klient
-Nr domu
-Kod pocztowy
-Miasto

Następnie wartości odczytane z pliku są przekazywane do konstruktora Klasy Klient, nowy Klient jest zapisywany do vectora wszystkich klientów.

Drugim plikiem tekstowym załączonym do programu jest lista dostępnych w sklepie produktów, linia zawierająca informacje potrzebne do wygenerowania jednego produktu wygląda następująco:

Jablko 210 5 599

Są to po kolei oddzielone spacjami:
-Nazwa produktu
-Ilość produktu w sklepie
-Wartość podatku vat w %
-Cena produktu w groszach

Następnie odczytane informację są przekazywane do konstruktora klasy Produkt, oraz każdy obiekt jest zapisywane do vectora reprezentującego listę dostępnych produktów.

Punkt 2
Przyjęte założenia

Nasza symulacja posiada wiele założeń, które należy omówić:

2 a) Przebieg symulacji jest turowy, w każdej turze wykonywane są operacje na stworzonych klasach, a tury podzielone są na trzy fazy.

2 b) Generowanie liczb losowych:
Generowanie liczb losowych odbywa się z wykorzystaniem biblioteki <random> do generowania liczb losowych oraz <chrono> do uzyskiwania unikalnego ziarna. W implementacji wykorzystujemy “Mersenne Twister 19937”, który generuje liczby losowe na podstawie ziarna ustalonego na podstawie czasu uniksowego powiększonego o liczbę iteracji programu, powiększenie te było potrzebne aby zapewnić losowość liczb generowanych w krótkich odstępach czasu. Komenda ustalająca ziarno wygląda następująco:

```
chrono::high_resolution_clock::now().time_since_epoch().count() + k
```

gdzie k to liczba iteracji.

2 c) Wczytywanie klientów oraz produktów z pliku

Proces ten przebiega w opisany w punkcie poświęconym załączonym plikom tekstowym sposób

2 d) Wypisywanie informacji do terminala i do pliku

Aby zapisać przebieg programu do pliku wystarczyłyby 3 linijki kodu, który zawierałby przeniesienie miejsca w którym nastąpiłyby wypisania do pliku tekstowego.
Służy do tego komendy:

```
ofstream out("Raport.txt", ios_base::app);
cout.rdbuf(out.rdbuf());
```

Niestety podczas wypisywania zarówno do terminala oraz do pliku wymagane jest przekierowanie na początku wyjścia tekstowego do pliku, wypisanie informacji a następnie przywrócenie wypisywania ponownie do terminalu i ponowne wpisanie tekstu.

Przykładowo:

```
ofstream out("Raport.txt", ios_base::app);
streambuf *coutbuf = cout.rdbuf(); // zapisanie adresu starego
//buffora
cout.rdbuf(out.rdbuf());
cout << "Klient: " << klienci[client_index].GetName() << " " <<
klienci[client_index].GetSurname() << " wchodzi do sklepu." <<
endl;
cout.rdbuf(coutbuf);
cout << "Klient: " << klienci[client_index].GetName() << " " <<
klienci[client_index].GetSurname() << " wchodzi do sklepu." <<
endl;
```

2 e) Poruszanie klientami za pomocą vectorów
(czekający,aktywni,w kolejce):

Na początku informacje o klientach zapisanych w pliku tekstowym są odczytywane oraz za pomocą konstruktora Klasy Klient tworzony jest obiekt klienta.

Następnie każdy z klientów po stworzeniu otrzymuję listę zakupów o długości zdeterminowanej w pliku tekstowym. Lista ta

również jest vectorem, w którym zapisywany jest tuple, zawierający losowo wybrany produkt z listy dostępnych produktów oraz losowo generowaną ilość danego produktu (np. 8 Bananów).

Następnym krokiem, po stworzeniu każdego klienta jest przeniesienie losowego klienta podczas fazy wchodzenia do vectora active_klient - zawierającego osoby aktywnie gromadzące przedmioty w sklepie.

Podczas gromadzenia przedmiotów są one przenoszone z vectora listy zakupów do vectora reprezentującego koszyk.

Gdy lista zakupów jest pusta klient “udaje się do kolejki do kasy”, czyli jest przenoszony z vectora active_klient do listy queue.

Po finalizacji zakupów w celu ewentualnego ponownego wejścia do sklepu, klientowi jest czyszczony vector koszyka oraz przypisywane są nowe produkty do listy zakupów. Jest on również przenoszony do vectora klient - listy wszystkich potencjalnych klientów którzy czekają na swoją kolej aby wejść do sklepu.

2 f)
Używane biblioteki:

`<vector>` – biblioteka używana do tworzenia wektorów: klientów, kas, produktów, koszyka, listy zakupów
`<chrono>` – Wykorzystywana do obliczania czasu uniksowego, potrzebnego do generacji ziarna liczb pseudolosowych
`<random>` – Wykorzystywana do generowania liczb pseudolosowych, w programie wykorzystujemy generator Mersenne Twister 19937
`<list>` – biblioteka używana do tworzenia listy reprezentującej kolejkę w sklepie

<fstream> - biblioteka używana do zapisu przebiegu działania programu do pliku tekstowego
<windows.h> - Wykorzystywana do implementacji opóźnień w działaniu programu metodą sleep()

Punkt 3

Przebieg symulacji

Symulacja z założenia to turowy symulator supermarketu.
Jako parametry wejściowe przyjmowane są 2 wartości:
-**liczba kas**
-**liczba tur**

Operacje przebiegu symulacji są metodami głównej klasy Supermarket

Tuż po rozpoczęciu działania programu z plików tekstowych wczytywane są informacje o klientach oraz produktach.

Tworzone są kasy o losowych parametrach i umieszczane do wektora. Ilość otwartych kas jest również losowa, pod warunkiem, że jest większa od 0.

Czyszczony jest również plik tekstowy z zawierający poprzedni przebieg symulacji.

W każdej turze realizowane są trzy fazy:
- Faza wchodzenia klientów do sklepu
- Faza wybierania produktów
- Faza finalizacji zakupów oraz wychodzenia ze sklepu

Przebieg każdej z nich jest wyjaśniony poniżej:

Faza wchodzenia klientów do sklepu:

W tej turze z vectora reprezentującego listę potencjalnych klientów którzy nie wykonują aktualnie żadnej czynności wybierany jest losowo jeden z klientów.

Następnie jest on przenoszony z listy bezczynnych do listy czynnych klientów w sklepie.

Faza wybierania produktów:

W tej turze wszyscy klienci z vectora aktywnych klientów przenoszą pierwszy produkt z listy zakupów do koszyka. Następnie produkt ten jest usuwany z listy zakupów.

Podczas tej operacji modyfikowana jest ilość produktów dostępnych w sklepie, zależnie od ilości jaką potrzebuje klient.

W przypadku gdy nie ma produktu którym byłby zainteresowany klient, wypisywany jest komunikat o braku dostępności produktu oraz klient nie przenosi przedmiotu do koszyka.

W przypadku gdy klient opróżni swoją listę zakupów, przechodzi on ze swoim koszykiem do kasy, lub do kolejki jeżeli jest więcej chętnych do finalizacji zakupów.

Faza finalizacji zakupów oraz wychodzenia ze sklepu:

W tej turze do otwartych kas podchodzą odpowiednio pierwsi klienci oczekujący w kolejce, reprezentowanej przez listę queue.

Każdy klient na podstawie posiadanych przedmiotów w koszyku ma drukowany paragon lub fakturę, zależnie od przypisanej w pliku wartości bool.

Po wydrukowaniu faktury lub paragonu z konta klienta odbierana jest odpowiednia ilość pieniędzy. W przypadku gdy klient nie ma wystarczającej ilości pieniędzy faktura lub paragon nie jest drukowany a przedmioty znajdujące się w koszyku wracają do sklepu.

Następnie niezależnie od sytuacji koszyk klienta jest czyszczony oraz nadawana jest mu nowa lista zakupów, a sam klient jest przenoszony na listę bezczynnych klientów.

Na końcu operacji, w zależności od ilości tur przepracowanych lub podczas których była zamknięta każda kasa, zmieniany jest jej status na zamkniętą lub otwartą.

Wszystkie 3 fazy wykonują się dla każdej iteracji programu (zadanej ilością tur)

Punkt 4

Obsługiwane błędy:

- **Błąd braku pieniędzy**: klient nie posiadający wystarczającej ilości pieniędzy przy kasie pozostawia zebrane przedmioty bez kupowania ich, informacja o tym jest wypisywana do raportu i terminala
- **Błąd braku produktów w sklepie:** w tym przypadku produkt jest usuwany z listy zakupów klienta, a ten kontynuuje zakupy. Informacji o tym jest wypisywana do raportu i terminala
- **Błąd braku otwartej przynajmniej jednej kasy**: w tym przypadku otwierana jest natychmiastowo kasa numer 1, pozostałe kasy zostają zamknięte