Projekt praktyczny

Spis treści

1. Cel projektu:

Celem projektu jest praktyczne poznanie metodologii tworzenia aplikacji. Szczególny nacisk będzie kładziony na narzędzia wspomagające oraz wykorzystanie elementów syntaktyki języka C++. Tematem bazowym w tym procesie jest stworzenie aplikacji symulującej pracę kawomatu.

2. Założenia projektowe

W fazie projektowanie ustalono iż:
- program będzie aplikacją konsolową ponieważ taka aplikacja zapewniającą szybki i prosty kontakt z użytkownikiem
- program powinien być odporny na działanie użytkownika, tzn. wybory jakich będzie dokonywał użytkownik nie powinny spowodować niepoprawnego zakończenia aplikacji. W tym celu zostanie wykorzystany mechanizmy obsługi sytuacji wyjątkowych.
- wykonany „kawomat" powinien umieć przygotować cztery różne rodzaje kaw. Dwie bez mleka oraz dwie z mlekiem:
  - ✓ Espresso
  - ✓ Black coffee
  - ✓ Late
  - ✓ BIG late
- użytkownik powinien móc wybrać własny typ kawy poprzez podanie przyporządkowanych numerów.
- po uruchomieniu kawomat powinien powinien zainicjalizować swój wewnętrzny licznik w którym będzie rejestrował ilości wykonanych przez siebie usług wykonania kaw. Rejestr ten powinien uwzględniać nie licznik zbiorowy ale liczniki indywidualne dla poszczególnych rodzajów wykonanych kaw.
- licznik powinien być niewidoczny dla użytkownika w trakcie eksploatacji.
- w dowolnym momencie będzie istniała możliwość wyłączenia kawomatu.
- jako element kończący pracę kawomatu powinien zostać wyświetlony rejestr z licznikami wykonanych usług (typów serwowanych kaw).

3. Implementacja

Aby spełnić wymagania projektowe wykonano klasę

```
class CoffeeMachine
```

która w konstruktorze inicjalizuje licznik swoich usług

```
coffeeCounter= {0, 0, 0, 0};
```

wraz z metodami klasy

```
public:
     void startCoffeeMachine();
private:
     void makeCoffee(int choice);
     void stopCoffeeMachine();
```

oraz zmienną realizującą inicjowany licznik:

```
     std::array<int,4> coffeeCounter;
```

Metoda `void startCoffeeMachine()` informuje użytkownika o możliwym wyborze usług do wykonania przez kawomat. Aby uodpornić aplikację na niepożądane działanie użytkownika wykorzystano mechanizm obsługi wyjątków. Wyjątki pozwalają reagować na różne sytuacje wyjątkowe. Używa się ich tam gdzie istnieje ryzyko wystąpienia wyjątku.

Bloki `try … catch`. Tam gdzie spodziewaliśmy się wyjątku umieszczono blok try, w którym zawarte są "podejrzane" instrukcje. W naszym wypadku będzie to wybór użytkownika. Za tym blokiem znajduje się blok catch. W instrukcji catch umieściliśmy , komunikat który musi się zawsze wyświetlać i poinformaować użytkownika jeśli nastąpił błąd.

Wybór użytkownika. Ponieważ użytkownik będzie podejmował decyzji wyłącznie na podstawie wartości jednej zmiennej oraz wybór będzie dokonywany na podstawie wartości liczby całkowitej (typ kawy do wykonania lub zakończenie pracy kawomatu), wybrano do implementacji instrukcję switch:

```
switch (choice) {
      case 1:
            std::cout << "Espresso\n";
            coffeeCounter.at(0)++;
            break;
      case 2:
            std::cout << "Black coffee\n";
            coffeeCounter.at(1)++;
            break;
      case 3:
            std::cout << "Late\n";
            coffeeCounter.at(2)++;
            break;
      case 4:
            std::cout << "BIG Late\n";
            coffeeCounter.at(3)++;
      break;
      default:
            std::cout << "Nieprawidlowa opcja\n";
            choice=-1;
            break;
      }
```

Dodatkowym atutem tej instrukcji jest możliwość inkrementacji liczników poszczególnych wyborów oraz kolejny element uodpornienia programu (na działania zewnętrzne) poprzez użycie default'a. Trafiają tam wszystkie pozostałe wartości z jakimi mogła zostać wywołana metoda: `int CoffeeMachine::makeCoffee(int choice)`

Metoda `int CoffeeMachine::stopCoffeeMachine()` realizuje punkt założeń projektowych o zakończeniu pracy kawomatu wraz z wyświetlaniem rejestru wykonanych usług.

4. Testowanie

Do testowania aplikacji wybrano framework GoogleTest dla C++. Aby wykonać test aplikacji niezbędne było przeprojektowanie metod klasy tak aby zwracały wartości kontrolne mogące posłużyć do weryfikacji poprawności działania. Wszystkie metody zostały też umieszczone za modyfikatorem public aby umożliwić ich działanie przez zewnętrzną aplikację.

```
int startCoffeeMachine()
int makeCoffee(int choice)
int stopCoffeeMachine()
```

Wykonano testy funkcji:

`int makeCoffee(int choice)` dla wszystkich możliwych argumentów oraz dla argumentu z poza zakresu.

```
EXPECT_EQ(1, cm2.makeCoffee(1));
EXPECT_EQ(2, cm2.makeCoffee(2));
EXPECT_EQ(3, cm2.makeCoffee(3));
EXPECT_EQ(4, cm2.makeCoffee(4));
EXPECT_NE(5, cm2.makeCoffee(5));
EXPECT_NE(0, cm2.makeCoffee(0));
```

W tym przypadku pokrycie instrukcji wynosi 100%, zaś pokrycie decyzji 100%.

`int startCoffeeMachine()` dla argumentu 5; Pokrycie instrukcji wynosi 33%, pokrycie decyzji 33%. Nie można było pokryć tetami automatycznymi większej ilości instrukcji ze względu na zastosowane konstrukcje (pętle do while oraz fakt iż inne parametry wejściowe powodowały wykonanie wewnętrznych instrukcji bez wychodzenia z funkcji).
Pozostałe warianty przetestowano manualnie. Łącznie pokrycie instrukcji wynosi 100%, zaś pokrycie decyzji 100%.

`int stopCoffeeMachine()` Pokrycie instrukcji testami automatycznymi wynosi 100%, brak decyzji w tej funkcji do pokrycia.

Review kodu

Podczas wykonania przeglądu kodu zaproponowano rozwinięcie programu o dodanie klas (każda klasa w osobnych plikach *.hpp, *.cpp), w celu implementacji mechanizmów dziedziczenia i polimorfizmu, a także wykorzystania dowolnego kontenera z biblioteki STL. Zgodnie z zaleceniami wykonano dodatkowe klasę bazową class Machine po której klasy: `CoffeeMachine, WeldingMachine, Computer` dziedziczą metody `getType()` oraz `getModel()` oraz dla których zostały stworzone metody virtualne `virtual void show()` oraz `virtual void sayWhatYouCanDo()` aby wykorzystać mechanizm polimorfizmu. Dodatkowo stworzono klasę `Laptop` z elementami jak powyższe oraz dziedziczącą po klasie `Computer`.

Wykorzystano kontener STL vector typu Machine* oraz Range-based for loop. Posługując się wskaźnikiem na obiekt Machine, a odnosząc sią do obiektów pochodnych wywołane zostały ich funkcje wirtualne.

```
for (auto e : m) {
            e->sayWhatYouCanDo();
            e->show();
}
```

System wersjonowania Github

W ramach projektu poznano narzędzie do wersjonowania kodu. Założono konto na githubie gdzie przechowywano poszczególne wersje programu wraz ze zmianami.

Contionious Integration

Ponieważ zakres projektu został rozszerzony w trakcie trwania projektu zmiany kodu narzuciły za sobą zmianę sposobu testów oraz ich rozszerzenie ich zakresu na nowododane elementy. Aby ułatwić sobie pracę wdrożono nowe narzędzie **cmake**.
Wykonano konfigurację `CMakeLists.txt` dla całego projektu.
Wygenerowano `Makefiile`'a który wspomagał kontrolę nad bieżącymi zmianami w projekcie a także odpowiadał za kompilację i kontrolę zmian w testach. Przy użyciu narzędzi **make** na bieżąco kompilowane były tylko te elementy, które ulegały zmianie a także elementy powiązane z nimi i mogące mieć wpływ na działanie całej aplikacji.

Reprezentacja graficzna

Efekt końcowy zależności powstałych w wyniku dziedziczenia klas przedstawia schemat UML. Zawiera informacje o związkach między elementami (klasami). Symbolem klasy jest prostokąt, zwykle podzielony poziomymi liniami na trzy sekcje:

- nazwy
- atrybutów
- operacji