Politechnika
Wrocławska

Programowanie Obiektowe

projekt wtorek 17:05

Symulacja Domu Towarowego

Autor:
Gabriel Malanowski 281081
Kamil Kondrat 281177

Prowadzący:
mgr inż. Tobiasz Puślecki

1 Wstęp

1.1 Opis projektu

Projekt Symulacja Domu Towarowego to system wspomagający zarządzanie magazynem, który umożliwia symulowanie i optymalizowanie procesów magazynowych. Dzięki niemu użytkownicy mogą modelować różne scenariusze, analizować wyniki oraz podejmować decyzje w czasie rzeczywistym. Projekt integruje algorytmy symulacyjne z interaktywnym interfejsem użytkownika, co pozwala na dynamiczne monitorowanie stanu magazynu oraz szybką reakcję na zmiany w otoczeniu biznesowym. Główne cechy projektu obejmują definiowanie magazynów, produktów, atrybutów, symulację działania magazynu w pętli czasowej, generowanie zdarzeń, interwencję użytkownika oraz generowanie raportów z wynikami symulacji. Celem projektu jest dostarczenie narzędzia wspomagającego efektywne zarządzanie magazynem, które pozwoli firmom na zwiększenie efektywności operacyjnej i maksymalizację zysków.

1.2 Cele projektu

Zyskanie i utrwalenie wiedzy w następujących zagadnieniach:

- Podstawy zunifikowanego języka modelowania (UML)
- Podstawy inżynierii i metodologii programowania obiektowego
- Znajomość podstawowych narzędzi obiektowo zorientowanego języka programowania na przykładzie języka C++
- Umiejętność stosowania technik obiektowych w programach
- Konstrukcja kodu modelującego zadany problem z wykorzystaniem hierarchii klas
- Umiejętność wykonania dokumentacji kodu źródłowego

2 Projekt symulacji

2.1 Analiza problemu

Problemem, który projekt ma rozwiązać, jest zarządzanie operacjami magazynowymi w sposób efektywny i zautomatyzowany. W szczególności chodzi o optymalizację procesów sprzedaży, dodawania i transferu produktów między magazynami oraz zapewnienie dokładnych raportów na temat stanu magazynów.

2.2 Ogólny opis symulacji

Użytkownik podaje informacje o magazynach oraz produktach w magazynie. Każdy obiekt ma swoje atrybuty, takie jak pojemność magazynu, cena produktu. Symulacja działa w pętli czasowej, gdzie każdy cykl reprezentuje jednostkę czasu (np. godzinę, dzień). W każdym cyklu, symulacja sprawdza stan magazynu i podejmuje decyzje na podstawie zdefiniowanych reguł. Zdarzenia takie jak sprzedaż

produktu są generowane losowo. Symulacja reaguje na te zdarzenia, aktualizując stan magazynu i inne powiązane obiekty. Na podstawie stanu magazynu i nadchodzących zdarzeń, symulacja podejmuje decyzje, takie jak zamówienie nowego towaru, przesunięcie zasobów, czy wysłanie powiadomień do administratorów. Wszystkie działania i zmiany są rejestrowane w systemie, co pozwala na analizę wyników symulacji i optymalizację procesów magazynowych. Na koniec symulacji, użytkownik otrzymuje raport z wynikami, takimi jak koszty operacyjne czy zysk netto.

2.3 Specyfikacja wymagań

2.3.1 Wymagania funkcjonalne

1. Dodawanie produktów do magazynu.
2. Sprzedaż produktów z magazynu.
3. Transfer produktów między magazynami.
4. Generowanie raportów o stanie magazynów.

2.3.2 Wymagania niefunkcjonalne

1. Wydajność - system powinien obsługiwać duże ilości danych.
2. Skalowalność - możliwość dodawania nowych magazynów i produktów.
3. Niezawodność - system powinien być odporny na błędy i awarie.

2.4 Zastosowane technologie

W projekcie zastosowano język C++ w połączeniu z frameworkiem Qt w wersji 6.10 wraz z systemem budowania CMake. Do przeprowadzania testów jednostkowych wykorzystano bibliotekę Google Test. W celu synchronizacji działań w zespole użyto rozproszonego systemu kontroli wersji Git wraz z możliwościami jakie daje platforma GitHub. W teorii dokumentacja Qt6 zapewnia działanie oprogramowania na platformach takich jak: dystrybucje systemu GNU/Linux oparte na serwerze graficznym X11, macOS w wersji 11 bądź nowszy oraz Microsoft Windows 10 (1809 bądź nowszy) czy Microsoft Windows 11. Twórcy projektu potwierdzili działanie aplikacji dla wybranych platform Microsoft Windows 10 (1809 bądź nowszy) czy Microsoft Windows 11 bez uprzednio zainstalowanych bibliotek Qt w systemach.

2.5 Diagram klas

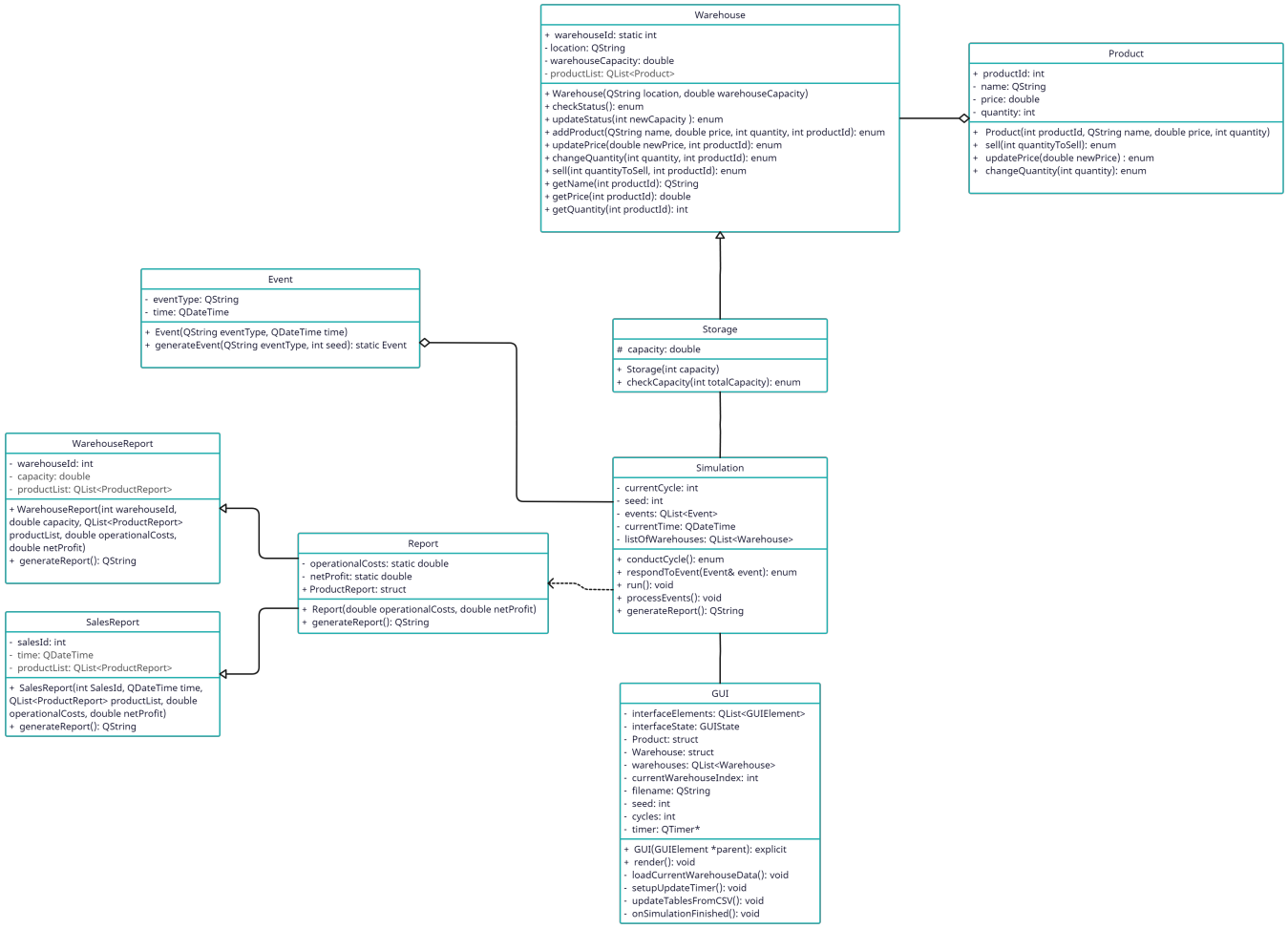

Figura 1: Diagram klas.

Opis diagramu klas dla symulacji domu towarowego w języku C++, który spełnia wymienione warunki:

1. Klasa **Simulation** :
   - **Atrybuty:** currentCycle, seed, currentTime, events, listOfWarehouses.
   - **Metody:** conductCycle(), respondToEvent(Event& event), run(), processEvents(), generateReport().
2. Klasa **Storage** :
   - **Atrybuty:** capacity.
   - **Metody:** checkCapacity(int totalCapacity).
3. Klasa **Warehouse** :
   - **Atrybuty:** warehouseId, location, werehouseCapacity, productList.

- **Metody:** checkStatus(), updateStatus(int newCapacity), addProduct(QString name, double price, int quantity, int productId), updatePrice(double newPrice, int productId), changeQuantity(int quantity, int productId), sell(int quantityToSell, int productId), getName(int productId), getPrice(int productId), getQuantity(int productId).

4. Klasa **Product** :

   - **Atrybuty:** productId, name, price, quantity.
   - **Metody:** sell(int quantityToSell), updatePrice(double newPrice) changeQuantity(int quantity).

5. Klasa **Event** :

   - **Atrybuty:** eventType, time.
   - **Metody:** generateEvent().

6. Klasa **Report** :

   - **Atrybuty:** operationalCosts, netProfit, ProductReport.
   - **Metody:** generateReport().

7. Klasa **WarehouseReport** :

   - **Atrybuty:** warehouseId, capacity, productList.
   - **Metody:** generateReport().

8. Klasa **SalesReport** :

   - **Atrybuty:** salesId, time, productList.
   - **Metody:** generateReport().

9. Klasa **GUI** :

   - **Atrybuty:** interfaceElements, interfaceState, Product, Warehouse, warehouses, currentWarehouseIndex, filename, seed, cycles, timer.
   - **Metody:** render(), loadCurrentWarehouseData(), setupUpdateTimer(), updateTablesFromCSV(), onSimulationFinished().

Hermetyzacja jest zastosowana poprzez ustawienie atrybutów jako prywatnych (*private*) i dostęp do nich poprzez publiczne metody (*public*).

Dziedziczenie jest reprezentowane przez klasę *Warehouse*, która dziedziczy po klasie *Storage*.

Kompozycja występuje, gdy *Simulation* zawiera obiekty *Warehouse*, które z kolei zawierają obiekty *Product*.

Agregacja jest zastosowana w klasie *Simulation* dla klasy *Event*.

Polimorfizm może być reprezentowany przez różne typy zdarzeń, które są obsługiwane przez metodę *generateReport()* w klasie *Report*. Każde zdarzenie może mieć inną implementację tej metody, w zależności od jego typu.

2.6 Diagram obiektów

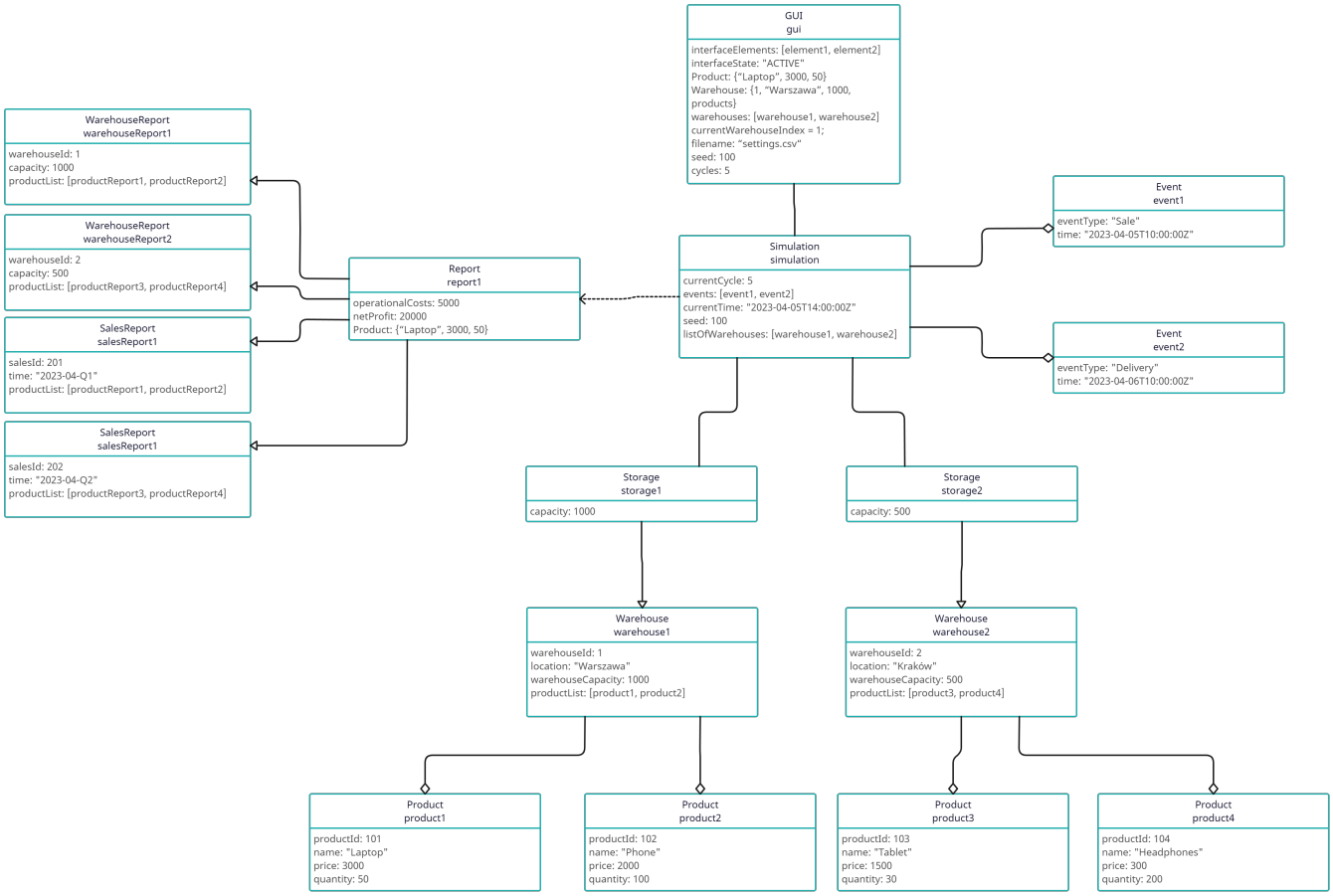

Figura 2: Diagram obiektów.

Opis diagramu obiektów dla przykładowej symulacji domu towarowego w języku C++.

1. Obiekty klasy **Simulation**
   - **simulation**: { currentCycle: 5, events: [event1, event2], currentTime: "2023-04-05T14:00:00Z", seed: 100, listOfWarehouses: [warehouse1, warehouse2] }
2. Obiekty klasy **Storage**
   - **storage1**: { capacity: 1000 }
   - **storage2**: { capacity: 500 }
3. Obiekty klasy **Warehouse**
   - **warehouse1**: { warehouseId: 1, location: "Warszawa", capacity: 1000, productList: [product1, product2] }
   - **storage2**: { warehouseId: 2, location: "Kraków", capacity: 500, productList: [product3, product4] }
4. Obiekty klasy **Product**

- **product1**: { productId: 101, name: "Laptop", price: 3000, quantity: 50 }
- **product2**: { productId: 102, name: "Phone", price: 2000, quantity: 100 }
- **product3**: { productId: 103, name: "Tablet", price: 1500, quantity: 30 }
- **product4**: { productId: 104, name: "Headphones", price: 300, quantity: 200 }

5. Obiekty klasy **Event**
   - **event1**: { eventType: "Sale", time: "2023-04-05T10:00:00Z" }
   - **event2**: { eventType: "Delivery", time: "2023-04-06T10:00:00Z" }
6. Obiekty klasy **Report**
   - **report1**: { operationalCosts: 5000, netProfit: 20000, Product: {"Laptop", 3000, 50} }
7. Obiekty klasy **WarehouseReport**
   - **warehouseReport1**: { warehouseId: 1, capacity: 1000, productList: [productReport1, productReport2] }
   - **warehouseReport2**: { warehouseId: 2, capacity: 500, productList: [productReport3, productReport4] }
8. Obiekty klasy **SalesReport**
   - **salesReport1**: { salesId: 201, time: "2023-04-Q1", productList: [productReport1, productReport2] }
   - **salesReport2**: { salesId: 202, time: "2023-04-Q2", productList: [productReport3, productReport4] }
9. Obiekty klasy **GUI**
   - **gui**: { interfaceElements: [element1, element2], interfaceState: "ACTIVE", Product: {"Laptop", 3000, 50}, Warehouse: {1, "Warszawa", 1000, products}, warehouses: [warehouse1, warehouse2], currentWarehouseIndex = 1, filename: "settings.csv", seed: 100, cycles: 5 }

2.7 Szczegółowy opis działania symulacji

Użytkownik rozpoczyna symulację poprzez interfejs użytkownika **GUI** lub terminal, tworząc plik konfiguracyjny CSV oraz wywołując metodę *run()* klasy **Simulation**. Ta metoda inicjuje główną pętlę symulacji, która będzie się wykonywać przez określoną liczbę cykli reprezentujących jednostki czasu zawartą w zmiennej *currentCycle*. W każdym cyklu symulacji, metoda *conductCycle()* jest wywoływana. Odpowiada ona za przetwarzanie zdarzeń zaplanowanych na bieżący cykl, które są przechowywane w atrybucie *events*.

Zdarzenia takie jak przyjęcie nowego towaru, sprzedaż produktu lub zmiana zapotrzebowania są generowane losowo lub według harmonogramu. Są one tworzone przez metodę *generateEvent()* klasy **Event** i dodawane do kolejki zdarzeń w **Simulation**.

Metoda *respondToEvent()* klasy **Simulation** jest wywoływana, aby zareagować na każde zdarzenie. Może to obejmować aktualizację stanu magazynu, zamówienie nowego towaru czy przesunięcie zasobów.

Obiekty klasy **Warehouse**, które są częścią *listOfWarehouses* w **Simulation**, są aktualizowane w odpowiedzi na zdarzenia. Metody takie jak *checkStatus()* i *updateStatus()* są używane do monitorowania i modyfikacji stanu magazynu.

Produkty reprezentowane przez obiekty klasy **Product** są sprzedawane i zarządzane poprzez metody takie jak *sell(), updatePrice() i changeQuantity()*, które są wywoływane w odpowiedzi na zdarzenia sprzedaży.

Na koniec epoki, metoda *generateReport()* klasy **Report** jest wywoływana, aby utworzyć raport z wynikami symulacji, takimi jak koszty operacyjne i zysk netto. Raporty mogą być szczegółowe dla magazynów **(WarehouseReport)** lub sprzedaży **(SalesReport)**.

Po zakończeniu określonej liczby cykli, symulacja kończy działanie, a użytkownik otrzymuje końcowy raport z wynikami.

3 Implementacja

3.1 Użyte wzorce projektowe

W projekcie Symulacji Domu Towarowego zastosowano szereg wzorców projektowych, które przyczyniają się do zwiększenia elastyczności, skalowalności oraz czytelności kodu źródłowego.

- Wzorzec **Singleton** pozwolił na kontrolę nad tworzeniem instancji komponentów aplikacji.
- Wzorzec **Fabryka** ułatwił zarządzanie tworzeniem obiektów oraz pozwolił na elastyczne dostosowanie systemu do tworzenia różnorodnych instancji obiektów bez konieczności modyfikacji kodu.
- Wzorzec **Obserwator** był wykorzystywany do monitorowania zmian stanu obiektów i informowania o nich innych części symulacji, co pozwoliło zachować spójność stanu aplikacji i interfejsu użytkownika.
- Wzorzec **Strategia** umożliwił dynamiczną zmianę algorytmów działania obiektów, co jest kluczowe w symulatorze domu towarowego, gdzie różne scenariusze wymagają różnych decyzji (np. zakupienia produktu do magazynu).

3.2 Zmiany w stosunku do pierwotnego projektu

- Konstruktory klas przyjmują parametry w celu ustawienia swoich atrybutów danymi już w momencie tworzenia obiektów. Poprawia to również czytelność kodu.
- W klasie **Warehouse** *warehouseId* jest zmienną statyczną w celu lepszej kontroli nad unikalnością ID magazynu.
- Dodano dodatkowe metody umożliwiające operacje na obiektach klasy **Product** w celu utrzymania hierarchii klas oraz kompozycji powyższych klas.

- Atrybut *capacity* klasy **Storage** został elementem typu Protected, aby umożliwić modyfikację ów atrybutu w dziedziczonych klasach.
- W klasie **Simulation** usunięto element *eventAgenda*, ponieważ okazał się zbędnym atrybutem. Podobną informację można wyciągnąć bezpośrednio za pomocą metod klasy **Event**.
- W klasie **Simulation** zgodnie z propozycją prowadzącego zajęcia dodano atrybut *seed*, który pozwala użytkownikowi określić ziarno generatora liczb losowych.
- W klasie **Report** zamiast działać bezpośrednio odwoływać się do klasy **Product**, zastosowano specjalnie przygotowaną strukturę *ProductReport*. Poprawia to zgodność z wzorcem projektowym Obserwatora oraz nie narusza hierarchii klas.
- W klasie **GUI** dodano dodatkowe metody i atrybuty, aby implementacja nie polegała wyłącznie na slotach dostępnych w frameworku Qt oraz w celu przestrzegania zasad czystego kodu.

3.3 Testowanie

Do testowania użyto metod testowania jednostkowego za pomocą biblioteki Google Test oraz metodyki Test Driven Development. Scenariusze testowe obejmowały przetestowanie działania metod klas aplikacji np. poprawnej modyfikacji zadanego atrybutu czy poprawności zwracanej wartości. Obecny kod spełnia wszystkie przewidziane testy.

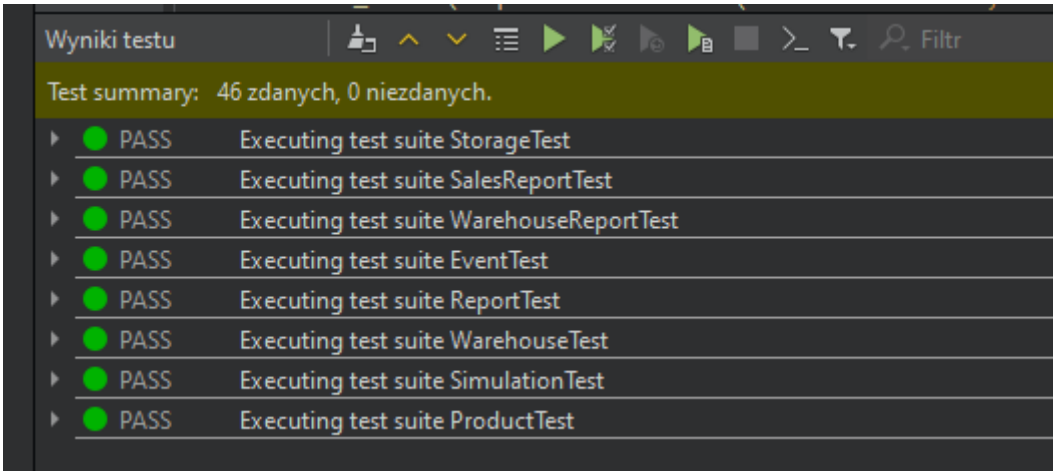

Figura 3: Wyniki testów jednostkowych symulacji.

4 Przykładowy scenariusz

4.1 Uruchomienie z poziomu wiersza poleceń

Aplikację można uruchomić z poziomu wiersza poleceń. W tym celu należy w katalogu z plikiem wykonywalnym uruchomić polecenie *Warehouse-simulator.exe –nogui*. W trybie wiersza poleceń jest możliwe również wykorzystanie parametru *–noconfig*, w celu ponownego uruchomienia poprzedniej symulacji bez konieczności przechodzenia etapu konfiguracji oraz *–file* wraz z nazwą pliku z uprzednio przygotowaną konfiguracją poza katalogiem wykonywalnym.

```
**************************
* 1. Add warehouse        *
* 2. Add product          *
* 3. Set number of cycles *
* 4. Set event seed       *
* 5. Undo last change     *
* 9. Exit configuration   *
**************************


Enter option: 1



Enter warehouse location: Wroclaw
Enter capacity of warehouse: 100
```

Listing 1: Menu interfejsu wiersza poleceń.

```
Type,Location,Capacity,Name,Price,Quantity,Cycles,Seed
Warehouse,Wroclaw,100
Product,Laptop,1299.99,20
Product,Phone,599.99,80
Warehouse,Legnica,50
Product,Tablet,50,50
Cycles,,,,10
Seed,,,,,100
```

Listing 2: Plik konfiguracyjny settings.csv

```
Warehouse ID,Capacity
0,100
Product Name,Price,Quantity
Laptop,1299.99,14
Phone,599.99,74
Sales ID,Time
18,2024-06-09 19:25:23
Product Name,Price,Quantity Sold
Laptop,1299.99,14
Phone,599.99,74
Operational Costs,Net Profit
20779.8,681861
Warehouse ID,Capacity
1,50
Product Name,Price,Quantity
Tablet,50,38
Sales ID,Time
19,2024-06-09 19:25:23
Product Name,Price,Quantity Sold
Tablet,50,38
Operational Costs,Net Profit
20867.8,683673
```

Listing 3: Fragment wyjścia programu. Plik SimulationReport.csv

4.2 Uruchomienie symulacji z poziomu interfejsu graficznego

W tym celu należy uruchomić aplikację bez żadnych parametrów. W sekcji Settings można wybrać plik z konfiguracją lub wprowadzić ręcznie parametry symulacji oraz produkty w sekcji Warehouse. Następnie można uruchomic aplikację wybierając w menu Start simulation oraz zatwierdzając wybraną opcję przyciskiem Start simulation.

Figura 4: Ekran startowy aplikacji.

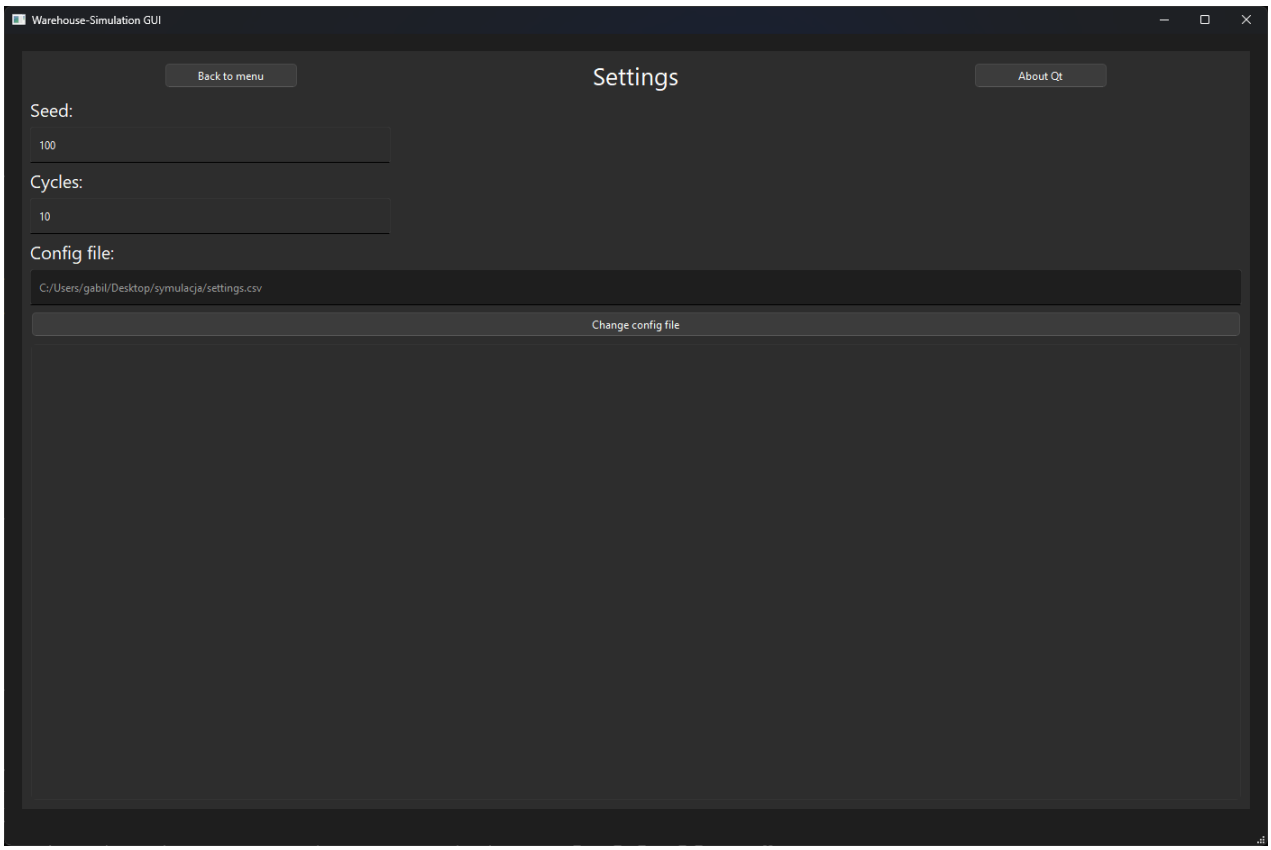

Figura 5: Ekran ustawień parametrów symulacji.

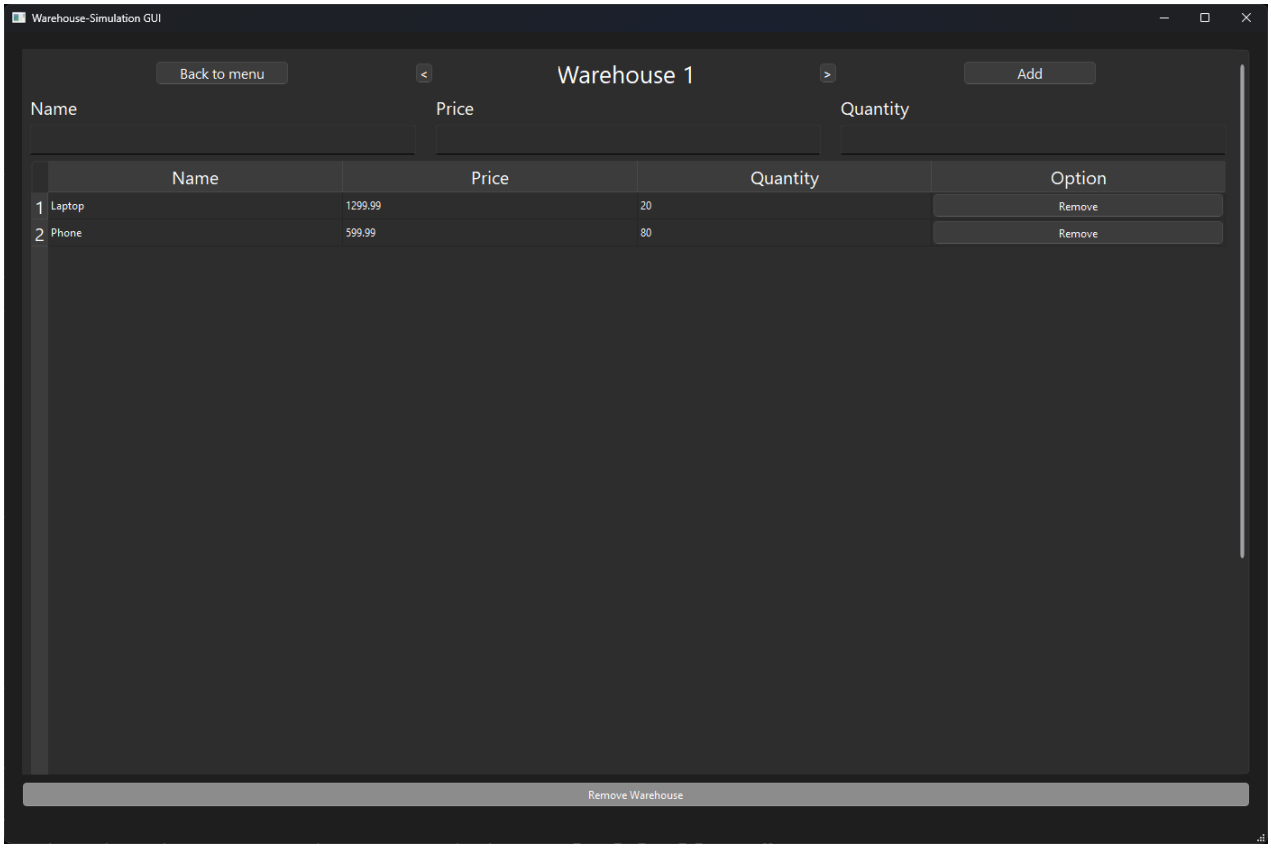

Figura 6: Ekran ustawień pierwszego domu towarowego.

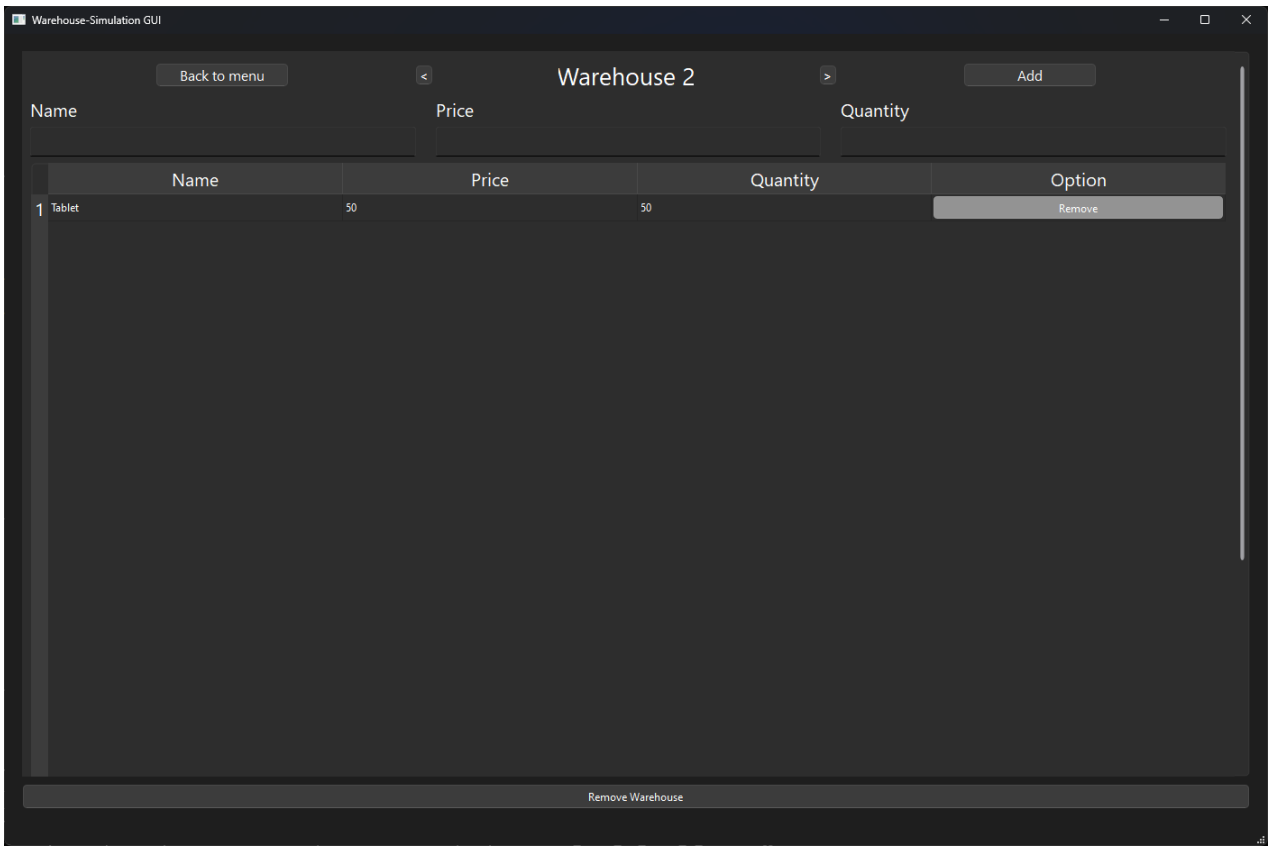

Figura 7: Ekran ustawień drugiego domu towarowego.

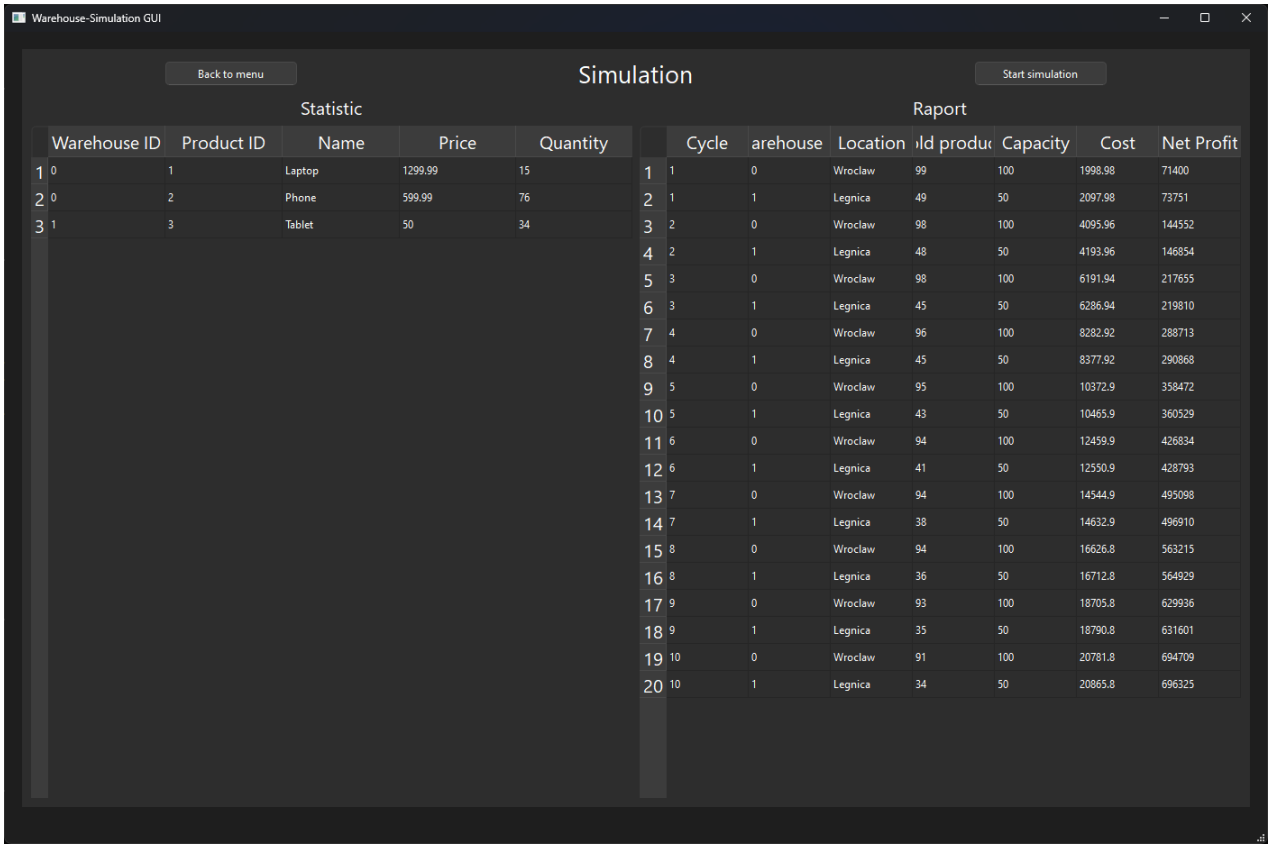

Figura 8: Ekran z wynikami symulacji.

5 Podsumowanie

W ramach projektu zrealizowano cele związane z poszerzaniem wiedzy i umiejętności w kilku kluczowych obszarach. Projekt umożliwił zapoznanie się z podstawami znormalizowanego języka modelowania UML, co jest fundamentalnym narzędziem w inżynierii oprogramowania. Ponadto, zdobyto wiedzę z zakresu inżynierii i metodologii programowania obiektowego, co jest niezbędne w projektowaniu nowoczesnych systemów.

Istotnym elementem było także poznanie podstawowych narzędzi programowania obiektowego na przykładzie języka C++, co pomogło w praktycznym zastosowaniu zdobytej wiedzy teoretycznej. W trakcie projektu rozwijano umiejętności stosowania technik obiektowych w programach, co pozwala na tworzenie bardziej modułowego i elastycznego kodu.

Konstrukcja kodu z wykorzystaniem hierarchii klas umożliwiła modelowanie złożonych problemów, a także praktyczne zastosowanie wzorców projektowych. Na koniec, umiejętność dokumentowania kodu źródłowego została ugruntowana poprzez szczegółowe opisanie implementacji, co jest kluczowe dla utrzymania i dalszego rozwoju oprogramowania.

W trakcie realizacji projektu napotkano na kilka wyzwań, takich jak synchronizacja pracy zespołu oraz integracja różnych modułów systemu. Problemy te rozwiązano poprzez regularne spotkania zespołu i zastosowanie narzędzi do zarządzania projektem, co zapewniło płynną realizację zadań.