Politechnika Krakowska

Wydział inżynierii Elektrycznej i Komputerowej

Studia Niestacjonarne, Informatyka

Sprawozdanie z projektu:

# Podstawy Programowania

**Prowadzący: dr inż. Sławomir Bąk**

Temat:

**Aplikacja do zarządzania browarem.**

**Wykonali:**

**Bartłomiej Bal**

**Matylda Kłujszo**

Spis treści:

1. Krótki opis realizowanego projektu
2. Opis implementacji z fragmentami kodu źródłowego
3. Przykłady działania ze zrzutami ekranu
4. Wnioski

# 1. Krótki opis realizowanego projektu

Celem projektu postawionym przed zespołem było napisanie aplikacji w języku C++, implementujący paradygmat programowania obiektowego. Aplikacja miała za zadanie mieć możliwość sprawdzania stanów magazynowych oraz możliwość sprawdzania zamówień, dodawania ich. W przedstawieniu danych miało pomóc wyświetlanie ich przy pomocy środowiska graficznego (nasz zespół użył środowiska FTXUI, link do repozytorium: https://github.com/ArthurSonzogni/FTXUI). Same dane przechowywane są w plikach tekstowych w formacie json (postawiliśmy na ten format plików tekstowych ponieważ jego struktura w prosty sposób pozwala na interakcje kodu z danymi oraz jest on przystępny dla oka przez co łatwo wykryć ewentualnie błędy), które odczytujemy przy pomocy biblioteki RapidJson, która pozwala na komunikację między aplikacją, a plikami tekstowymi.

# 2. Opis implementacji z fragmentami kodu źródłowego

Tak jak wspomnieliśmy w pkt. 1 aplikacja miała być napisana w technice obiektowej. Przez zespołem zostały postawione następujące wymagania co do napisanego kodu:

a) dziedziczenie
b) polimorfizm
c) klasy abstrakcyjne
d) klasy wewnętrzne
e) Prezentacja danych w konsoli lub z wykorzystaniem biblioteki graficznej,
f) Aplikacja ma odczytywać dane z plików tekstowych (pliki reprezentują bazę danych dla aplikacji),
g) Aplikacja ma zapisywać dane do pliku,
h) Zapis wyników działania programu do pliku tekstowego w formie „raportu”.

# a) dziedziczenie

**Dziedziczenie**, jak sama nazwa wskazuje, pozwala jednej klasie odziedziczyć coś (zmienne, metody) po drugiej. Klasę dziedziczącą nazywamy klasą pochodną, a klasę, po której klasa pochodna dziedziczy, nazywamy klasą bazową. (Klasa pochodna pochodzi od bazowej).

W przypadku naszej aplikacji zaimplementowaliśmy ją np. w klasach Json, z której publicznie dziedziczy klasa BeerStockJson.

```
#ifndef BREWERY_MANAGEMENT_JSON_H
#define BREWERY_MANAGEMENT_JSON_H

#include <rapidjson/document.h>
#include <functional>
#include <string>
#include <rapidjson/schema.h>

namespace brewery {

class Json {
public:
    virtual ~Json() = default;

    void setSchema(const std::string&);
    void validate(const rapidjson::Document& doc) const;
    static std::string doc2str(rapidjson::Document& doc);
protected:
    void setMember(rapidjson::Value&, const std::string&, const
std::function<void(std::string)>&) const;
    std::unique_ptr<rapidjson::SchemaDocument> schema_;
};
}
#endif //BREWERY_MANAGEMENT_JSON_H
```

klasa Json

```
#ifndef BREWERY_MANAGEMENT_BEERSTOCKJSON_H
#define BREWERY_MANAGEMENT_BEERSTOCKJSON_H

#include <models/Stock.h>
#include <models/Beer.h>
#include "Json.h"

namespace brewery {
class BeerStockJson : public Json {
public:
    BeerStockJson() = default;
    ~BeerStockJson() override = default;

    void load(std::vector<Stock<Beer>>&, const rapidjson::Document&) const;
    void toJson(std::vector<Stock<Beer>>&, rapidjson::Document&) const;
};
}
#endif //BREWERY_MANAGEMENT_BEERSTOCKJSON_H
```

klasa BeerStockJson

# b) polimorfizm

**Polimorfizm** jest słowem zaczerpniętym do informatyki stosunkowo niedawno, podczas rozwoju języków programowania. W języku **C++** możemy korzystać z tego mechanizmu za pomocą **metod wirtualnych**. Dzięki niemu mamy pełną kontrolę nad wykonywanym programem, nie tylko w momencie kompilacji (**wiązanie statyczne**) ale także podczas działania programu (**wiązanie dynamiczne**) – niezależnie od różnych wyborów użytkownika.

Polimorfizm w naszej aplikacji został użyty w klasach, które pracują na plikach przestawionych na poniższych zrzutach.

```
#ifndef BREWERY_MANAGEMENT_OUTPUT_H
#define BREWERY_MANAGEMENT_OUTPUT_H

#include <string>

namespace brewery {
class Output {
public:
    virtual ~Output() = default;
    virtual void write(const std::string&) const = 0;
};
}

#endif //BREWERY_MANAGEMENT_OUTPUT_H
```

klasa Output

```
#ifndef BREWERY_MANAGEMENT_FILEOUTPUT_H
#define BREWERY_MANAGEMENT_FILEOUTPUT_H

#include <string>
#include <streambuf>
#include "Output.h"

namespace brewery {
class FileOutput : public Output {
public:
    FileOutput() = delete;
    explicit FileOutput(const std::string&, bool trunc = false);
    ~FileOutput() override = default;

    void write(const std::string&) const override;

private:
    std::string path_;
};
}

#endif //BREWERY_MANAGEMENT_FILEOUTPUT_H
```

klasa FileOutput

## c) klasy abstrakcyjne

W C++ klasą abstrakcyjną jest **klasa**, która posiada zadeklarowaną co najmniej jedną metodę czysto wirtualną. Każda **klasa**, która dziedziczy po **klasie abstrakcyjnej** i sama nie chce być abstrakcyjną, musi implementować wszystkie odziedziczone metody czysto wirtualne.

Klasą abstrakcyjna w naszym projekcie jest m.in. klasa Output przedstawiona we fragmencie kodu w poprzednim podpunkcie.

## d) klasy wewnętrzne

Klasa wewnętrzna lub zagnieżdżona to klasa, która jest zdefiniowana wewnątrz innej klasy. Klasy wewnętrzne posiadają dostęp do metod statycznych klasy głównej, lecz nie muszą być instancjonowane wraz z nią. Zależnie od języka, mogą, ale nie muszą być dostępne spoza klasy głównej.

Klasy wewnętrznej w aplikacji użyliśmy do stworzenia buildera, klasy OrderBuilder, która pomaga w zmianie stanów zamówienia oraz odczytywania danych z określonego zamówienia.

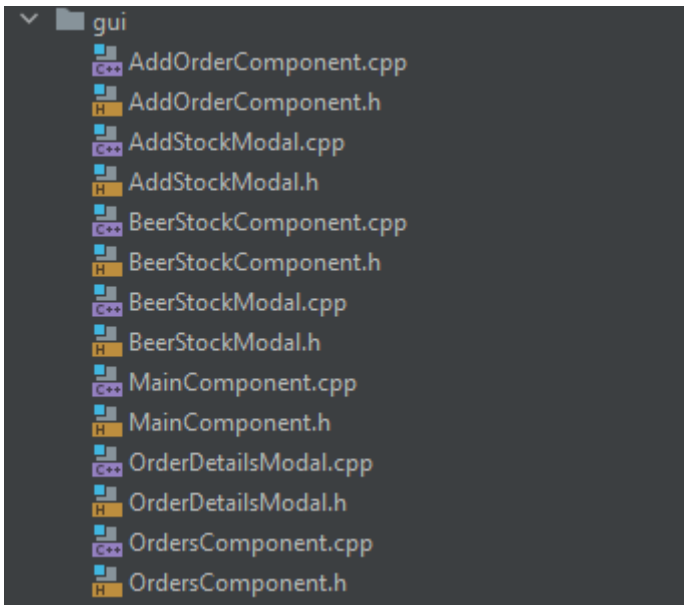
```
#ifndef BREWERY_MANAGEMENT_ORDER_H
#define BREWERY_MANAGEMENT_ORDER_H

#include <vector>
#include <ctime>
#include <optional>
#include <memory>

#include "Beer.h"
#include "Stock.h"
#include "Customer.h"

namespace brewery {

class Order {
public:
    using OrderList = std::vector<ProductQuantityPair<Beer>>;

    Order() = default;
    ~Order() = default;

    std::string getOrderDate() const;
    bool isCompleted() const;
    std::string getCompletionDate() const;
    Customer getCustomer() const;
    OrderList getOrderedProducts() const;

    class OrderBuilder
    {
    public:
        OrderBuilder();
        explicit OrderBuilder(Order&);
        ~OrderBuilder() = default;
```

```
        operator Order() const;

        OrderBuilder& placed(const std::string&);
        OrderBuilder& completed(const std::string&);
        OrderBuilder& products(OrderList);
        OrderBuilder& by(const Customer&);

    private:
        std::unique_ptr<Order> order_;
    };

private:
    OrderList beerList_;
    Customer customer_;
    std::string orderDate_;
    std::optional<std::string> completionDate_;

public:
    static OrderBuilder create();
    static OrderBuilder update(Order&);
};

} //namespace

#endif //BREWERY_MANAGEMENT_ORDER_H
```

klasa wewnętrzna OrderBuider zagnieżdżona w klasie Order

# e) Prezentacja danych w konsoli lub z wykorzystaniem biblioteki graficznej

Do prezentacji danych w konsoli użyliśmy biblioteki FTXUI, która renderuje prymitywną aplikację okienkową wewnątrz terminalu systemowego. W naszej aplikacji cały kod związany ze środowiskiem graficznym zawarty jest w plikach w folderze gui.



Przykładowo do wyświetlenia w aplikacji stanów magazynowych używany jest kod z pliku BeerStockComponent.cpp

```
#include <ftxui/component/event.hpp>
#include <sstream>
#include <iomanip>
#include <logging/LogHelpers.h>
#include "BeerStockComponent.h"
#include "BeerStockModal.h"

using namespace brewery;

BeerStockComponent::BeerStockComponent(std::shared_ptr<BreweryController>&
controller) :
    breweryController_(controller),
    detailsModal_(ftxui::Make<BeerStockModal>(breweryController_,
openDetailsModal_)),
    addStockModal_(ftxui::Make<AddStockModal>(breweryController_,
openNewStockModal_))
{
    addStockButton_ = ftxui::Button(L"Add stock", [&] {
        openNewStockModal_ = true;
    });
    Add({ftxui::Container::Vertical({
        ftxui::Container::Vertical({}),
        addStockButton_}),
    });
}

bool
BeerStockComponent::OnEvent(ftxui::Event event)
{
    if (openDetailsModal_)
    {
        return detailsModal_->OnEvent(event);
    }

    if (openNewStockModal_)
    {
        return addStockModal_->OnEvent(event);
    }

    int oldSelected = selected_;
    if (event == ftxui::Event::ArrowUp || event ==
ftxui::Event::Character('k'))
    {
        selected_--;
    }
    if (event == ftxui::Event::ArrowDown || event ==
ftxui::Event::Character('j'))
    {
        selected_++;
    }

    if (selected_ == size_)
    {
        return ftxui::ComponentBase::OnEvent(event);
    }
    else
    {
        if (event == ftxui::Event::Return || event ==
```

```
ftxui::Event::Character(' '))
        {
            openDetailsModal_ = true;
        }
    }

    selected_ = std::max(0, std::min(size_, selected_));

    return selected_ != oldSelected;
}

ftxui::Element
BeerStockComponent::Render()
{
    auto beerStocks = breweryController_->getAllBeerStocks();
    size_ = beerStocks.size();

    auto header = ftxui::hbox({
        ftxui::text(L"Name") | ftxui::flex,
        ftxui::separator(),
        ftxui::text(L"Quantity") | ftxui::size(ftxui::WIDTH, ftxui::EQUAL,
20) | ftxui::notflex,
        ftxui::separator(),
        ftxui::text(L"Singular price") | ftxui::size(ftxui::WIDTH,
ftxui::EQUAL, 20) | ftxui::notflex
    });

    int idx = 0;
    ftxui::Elements records;
    for (auto& stock : beerStocks)
    {
        ftxui::Decorator decorator = ftxui::dim;
        if (selected_ == idx++)
        {
            decorator = decorator | ftxui::focus | ftxui::inverted;
        }
        std::stringstream ss;
        ss << std::fixed << std::setprecision(2) << stock-
>getSingularPrice();

        ftxui::Element record = ftxui::hbox({
            ftxui::text(stock->getProduct().getName()) | ftxui::flex,
            ftxui::separator(),
            ftxui::text(std::to_string(stock->getQuantity())) |
ftxui::size(ftxui::WIDTH, ftxui::EQUAL, 20) | ftxui::notflex,
            ftxui::separator(),
            ftxui::text(ss.str()) | ftxui::size(ftxui::WIDTH, ftxui::EQUAL,
20) | ftxui::notflex
        }) | ftxui::flex | decorator;
        records.push_back(record);
    }

    auto window = ftxui::window(ftxui::text(L"Beer Stock"),ftxui::vbox({
        header,
        ftxui::separator(),
        ftxui::vbox(records) | ftxui::yframe,
        ftxui::filler(),
        addStockButton_->Render() | ftxui::align_right |
ftxui::size(ftxui::HEIGHT, ftxui::LESS_THAN, 3)
    }));
```

```
    if (openDetailsModal_)
    {
        LOG_INFO("Opening stock #" << selected_ << " details");
        window = ftxui::dbox(window,
                             detailsModal_->Render(beerStocks[selected_]) |
ftxui::clear_under | ftxui::center);
    }

    if (openNewStockModal_)
    {
        window = ftxui::dbox(window,
                             addStockModal_->Render() | ftxui::clear_under
| ftxui::center);
    }

    return window;
}
```

W konstruktorze klasy ładowany jest kontroler w którym znajdują się funkcje łądujące dane oraz funkcje pozwalające operować na uprzednio załadowanych danych. Następnie do dwóch smartpointerów detailsModal_ oraz addStockModal_ przypisywne są 2 modale (małe okna pozwalające na modyfikacje danych z określonej pozycji na liście stanów magazynowych lub dodawanie nowych pozycji w stanach. Kolejno deklarowany oraz dodawany jest guzik do dodawania stanów, który otwiera modal z addStockModal_.

Funkcja OnEvent() wyłapuje interakcje użytkownika z aplikacją poprzez używanie klawiatury.

Funkcja Render() służy do wyświetlania stanów magazynowych. W niej do zmiennej beerStocks ładowane są stany magazynowe. Następnie tworzony jest wiersz z nazwami kolumn tabeli stanów. Elements Records (jest to vector) służy jako kontener na wiersze record, do których w pętli pokolej ładowane są stany magazynowe. Do elementu window ładowane są po kolei nagłówek header, separtor (poziom linia oddzielająca nagłówki od pozycji), vbox(records) to jak nazwa wskazuje wertykalne pudełko (kontener) w którym wyświetlane są pozycje z wektora records. Filler to element który wypełnia pustą cześć okna tabeli i zmniejsza/zwiększa się proporcjonalnie do zawartości vboxa na nad nim. Kolejno addStockbutton_ czyli wcześniej wyświetlany guzik do otwierania okna dialogowego służącego do dodawania stanów magazynowych. Dwa okna dialogowe renderowane są pod warunkiem że wykonaliśmy akcję, która zmienia przypisaną im wartość boolowską na true. Na koniec Render() zwraca okno które potem wyświetalne jest w oknie konsoli.

## f) Aplikacja ma odczytywać dane z plików tekstowych (pliki reprezentują bazę danych dla aplikacji)

W aplikacji dane zapisane są w plikach o formacie json. Jednym z miejsc, w których wykorzystujemy tą funkcjonalność jest plik BeerStockJson.cpp, który służy do wyświetlania stanów magazynowych.

```
#include "BeerStockJson.h"

using namespace brewery;

void
BeerStockJson::load(std::vector<Stock<Beer>>& beerStocks, const
rapidjson::Document& doc) const
```

```
{
    validate(doc);

    auto itr = doc.FindMember("stocks");
    if (itr != doc.MemberEnd() && itr->value.IsArray())
    {
        for (auto& stock : itr->value.GetArray())
        {
            // TODO setMember...
            Beer b{stock["beer"].GetObject()["name"].GetString()};
            beerStocks.emplace_back(b, stock["quantity"].GetInt(),
stock["singularPrice"].GetDouble());
        }
    }
}

void
BeerStockJson::toJson(std::vector<Stock<Beer>>& beerStocks,
rapidjson::Document& doc) const
{
    doc.SetObject();
    auto allocator = doc.GetAllocator();

    rapidjson::Value beerStockArray{rapidjson::kArrayType};
    for (auto& stock : beerStocks)
    {
        rapidjson::Value stockVal{rapidjson::kObjectType};
        rapidjson::Value v;
        v.SetInt(stock.getQuantity());
        stockVal.AddMember("quantity", v, allocator);

        v.SetDouble(stock.getSingularPrice());
        stockVal.AddMember("singularPrice", v, allocator);

        rapidjson::Value beerVal{rapidjson::kObjectType};
        v.SetString(stock.getProduct().getName().c_str(), allocator);
        beerVal.AddMember("name", v, allocator);
        stockVal.AddMember("beer", beerVal, allocator);
        beerStockArray.PushBack(stockVal, allocator);
    }

    doc.AddMember("stocks", beerStockArray, allocator);
    validate(doc);
}
```

Do funkcji load() przesyłana jest referancja do stanów magazynowych oraz referencja do dokumentu json ze stanami.

Funkcja validate() waliduje zgodność ładowanego pliku json z plikiem schema.json, który sprawdza czy struktura pliku json jest zgodna. Dzięki temu możemy uniknąć błędów związanych z nieprawidłowościami takimi jak niepoprawne typy danych wewnątrz obiektów w pliku. Wrazie jakiś błędów zwraca ona komunikat to pliku z raportem przez co łatwo możemy zdebugować kod w pliku json i odnaleźć błędne informacje.

Następnie iteratorowi przypisujemy wartość ‘stocks’ (główny obiekt wewnątrz pliku json ze stanami magazynowymi). Następnie w pętli ładujemy po kolei obiekty z pliku ze stanami magazynowymi.

Działanie funkcji toJson() jest odwrotne do load() ponieważ w pierwszej dane są ładowane z pliku json, a w drugiej z kolei dane są konwertowane do formatu json. Funkcji tej używamy m.in. w funkcji zapisującej dane do pliku. Do funkcji toJson() przesyłamy pusty Document, któremu jego konstruktor domyślnie przypisuje wartość Null dlatego zmieniamy przy pomocy operacji SetObject() jego typ na obiekt (tak jak główny obiekt ‘stocks’ wewnątrz pliku ze stanami magazynowymi). Następnie pobieramy alokator dokumentu (pozwala on jak sama nazwa wskazuje na alokowanie ale również rozszerzanie/zmniejszanie oraz usuwanie bloków pamięci, w których przetrzymywane są dane). Kolejno tworzymy beerStockArray o typie kArrayType. Jest to jeden z typów danych json dostarczanych przez bibliotekę RapidJson. Odpowiada on tablicy elementów stanów magazynowych wewnątrz plików. Jak widać musimy ściśle trzymać się kolejności dodawania danych oraz ich typów aby później skonwertowane dane przeszły walidację. W pętli kolejno tworzymy potrzebne obiekty do przechowywania pojedynczego stanu magazynowego oraz które potem są umieszczane wewnątrz ”tablicy” beerStockArray. Po zakończeniu wykonywania się pętli tablica dodawana jest do obiektu doc, a następnie obiekt ten walidowany jest przez funkcję validate().

## g) Aplikacja ma zapisywać dane do pliku

Oprócz operacji na plikach, aplikacja ma też za zadanie zapisywać dane do pliku. Do tego służy np. funkcja save() wewnątrz pliku OrderService.cpp

```
void OrderService::save(std::shared_ptr<Output> output)
{
    rapidjson::Document doc;
    orderJson_.toJson(db_.orders_, doc);
    output->write(Json::doc2str(doc));
}
```

Funkcja ta otrzymuje output tzn. po wykonaniu jakiś operacji na wczytanych danych wynik z ich działania jest przesyłany do tej funkcji. Następnie przy pomocy funkcji toJson dane są konwertowane do formatu json, a potem zapisywane do pliku przy pomocy funkcji doc2str() z klasy Json.

```
std::string
Json::doc2str(rapidjson::Document &doc)
{
    rapidjson::StringBuffer strbuf;
    rapidjson::Writer<rapidjson::StringBuffer> writer(strbuf);
    doc.Accept(writer);

    return strbuf.GetString();
}
```

Funkcja ta otrzymuje obiekt doc. Wewnątrz niej tworzony jest bufor (strumień wyjściowy) dla przesyłanych danych.

Writer zapisuje dane do strbuf. Na koniec funkcja zwraca bufor.

# h) Zapis wyników działania programu do pliku tekstowego w formie „raportu”

Zapisywanie działania programu w formie raportu wykonywane jest przy pomocy dyrektywy LOG(). Otrzymuje ona poziom logu, jego typ oraz tekst który ma w nim być zapisany.

```
#ifndef BREWERY_MANAGEMENT_LOGHELPERS_H
#define BREWERY_MANAGEMENT_LOGHELPERS_H

#include <logging/Logger.h>
#include <utils/Time.h>
#include <sstream>

namespace brewery {

#define LOG(level, logLevelStr, stream) { \
    std::stringstream ss;       \
    ss << "[" << (logLevelStr) << "][" << getCurrentTime() << "][" <<
(const char*)__FILE__ << ":" << (const int)__LINE__     \
    << "] " << stream << '\n'; \
    Logger::log(level, ss.str());  \
}

#define LEVEL_DEBUG "DEBUG"
#define LEVEL_INFO "INFO"
#define LEVEL_WARNING "WARNING"
#define LEVEL_ERROR "ERROR"

#define LOG_DEBUG(stream) LOG(Logger::LogLevel::DEBUG, LEVEL_DEBUG, stream)
#define LOG_INFO(stream) LOG(Logger::LogLevel::INFO, LEVEL_INFO, stream)
#define LOG_WARNING(stream) LOG(Logger::LogLevel::WARNING, LEVEL_WARNING,
stream)
#define LOG_ERROR(stream) LOG(Logger::LogLevel::ERROR, LEVEL_ERROR, stream)
}
#endif //BREWERY_MANAGEMENT_LOGHELPERS_H
```

```
{
  "beerPath": "../json/stocks.json",
  "beerSchemaPath": "../json/schemas/stocks_schema.json",
  "ordersPath": "../json/orders.json",
  "ordersSchemaPath": "../json/schemas/orders_schema.json",
  "logPath": "../log.txt",
  "logLevel": "DEBUG"
}
```

W pliku cfg.json zapisana jest ścieżka do pliku w którym zapisywane są wyniki działania programu.

```
#include "json/ConfigJson.h"

using namespace std::placeholders;
using namespace brewery;
```

```
void
ConfigJson::load(Config& cfg, rapidjson::Document& doc)
{
    validate(doc);

    setMember(doc, "beerPath", std::bind(&Config::setBeerStocksPath, &cfg,
_1));
    setMember(doc, "beerSchemaPath",
std::bind(&Config::setBeerStocksSchemaPath, &cfg, _1));
    setMember(doc, "ordersPath", std::bind(&Config::setOrdersPath, &cfg,
_1));
    setMember(doc, "ordersSchemaPath",
std::bind(&Config::setOrdersSchemaPath, &cfg, _1));
    setMember(doc, "logPath", std::bind(&Config::setLogPath, &cfg, _1));
    setMember(doc, "logLevel", std::bind(&Config::setLogLevel, &cfg, _1));
}
```

Następnie w pliku ConfigJson.cpp odczytujemy przy pomocy funkcji load(), ścieżkę do pliku dzięki czemu aplikacja wie gdzie zapisywać raporty.

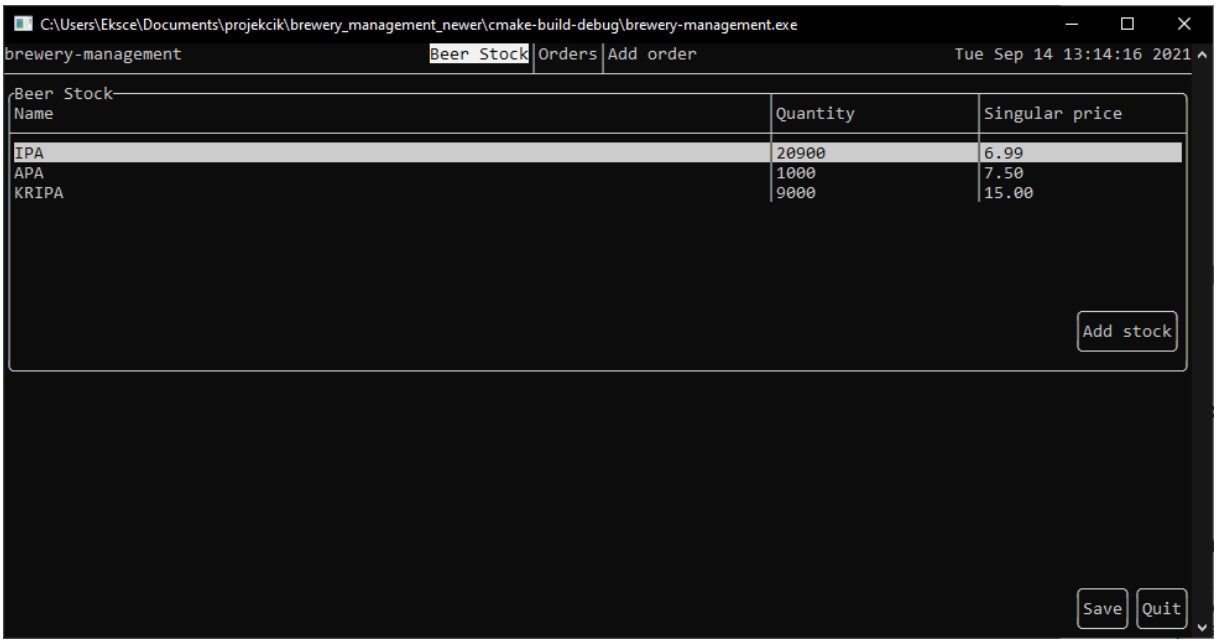
```
void
Json::setSchema(const std::string& schema)
{
    rapidjson::Document doc;
    if (doc.Parse(schema.c_str()).HasParseError())
    {
        LOG_ERROR("Schema is not a valid json");
    }
    schema_ = std::make_unique<rapidjson::SchemaDocument>(doc);
}
```

Przykładowe użycie raportowania działania programu w funkcji setSchema, która ustawia schemat plików json. Jeśli plik json nie zostanie sparsowany dokładnie do pliku z raportami zostanie wpisana przyczyna błędu.

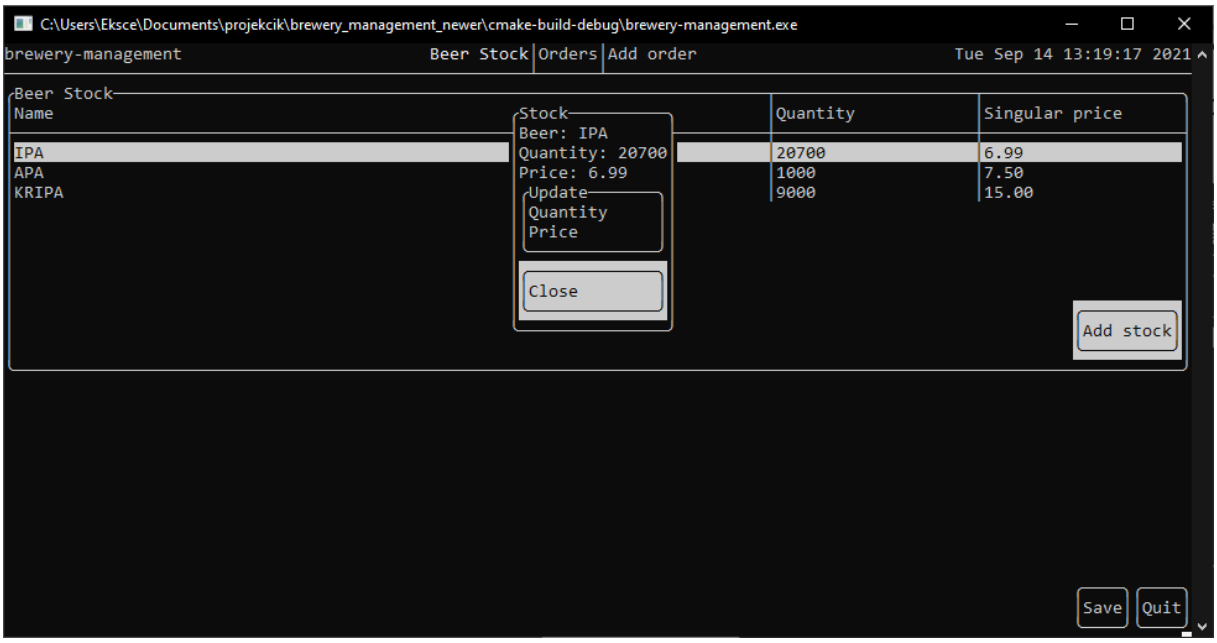
# 3. Przykłady działania ze zrzutami ekranu

Po uruchomieniu zbudowanego programu brewery-management.exe ukazuje się okno ze stanami magaznowymi.

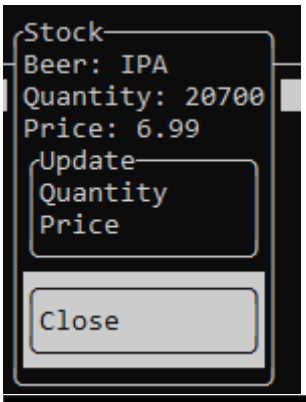
### Zakładka Beer Stock

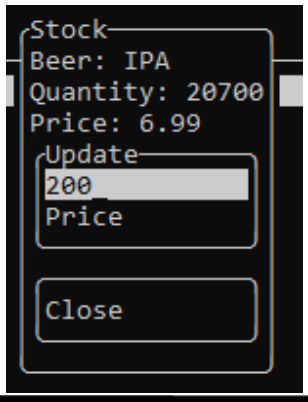



Wstępnie wybrana jest zakładka ze stanami magazynowymi. Możemy zmienić zakładkę przy pomocy myszki lub klawiszy strzałek w prawo oraz lewo. Używając strzałek góra, dół możemy przejść do tabeli ze stanami magazynowymi. Po naciśnięciu klawisza enter na wybranej pozycji program otworzy okno dialogowe.

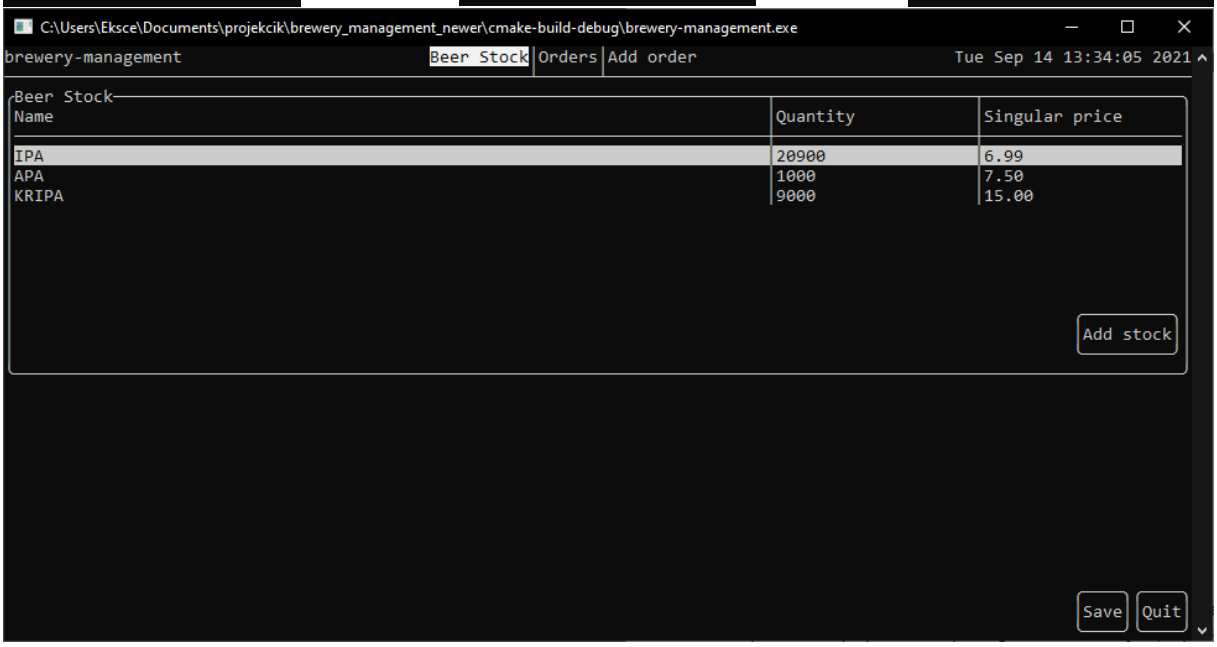



W nim wyświetlane są informacje o typie piwa, ilości piw, cena pojedynczego piwa. Pod nimi dostępne są dwa inputy, które możemy aktywować poprzez przejście na nie myszką lub klawiszami góra, dół. Następnie do aktywnego inputa możemy wprowadzić interesującą nas wartość, następnie wcisnąć klawisz enter.



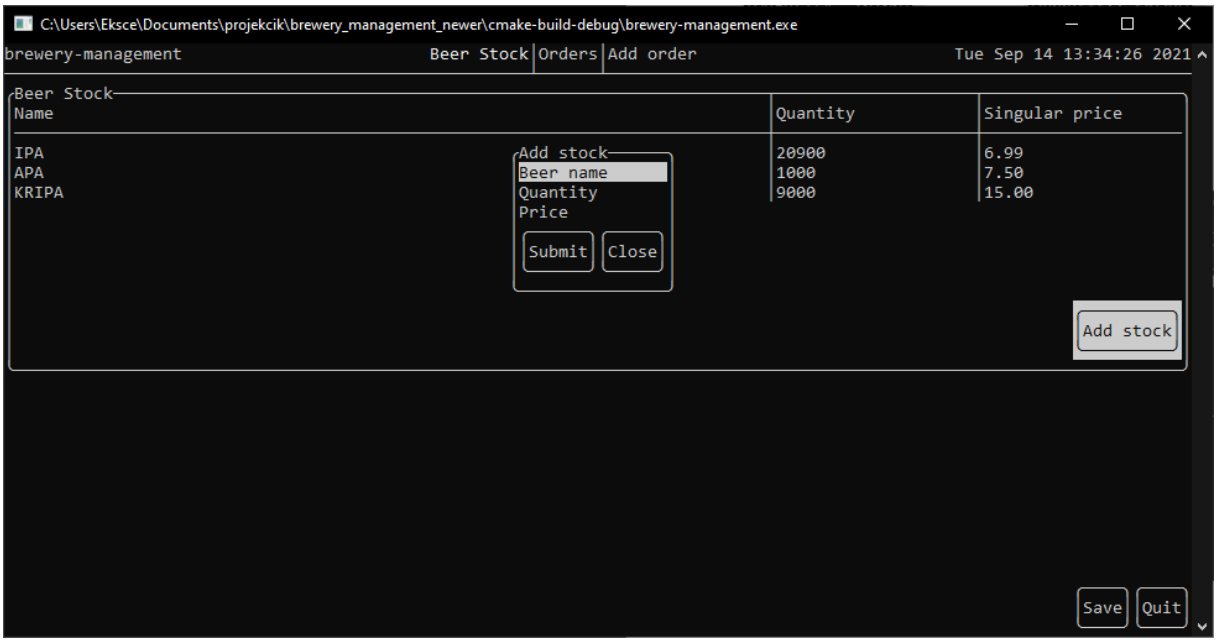





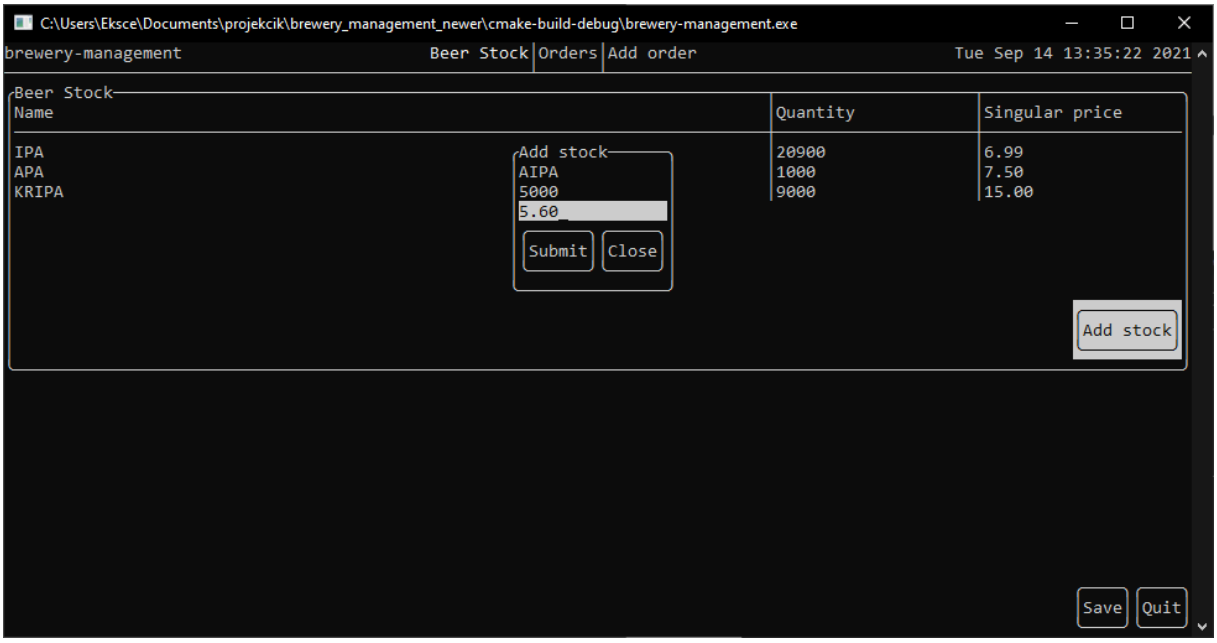



Jak widać stan magazynowy zmenił się z 20700 sztuk na 20900.

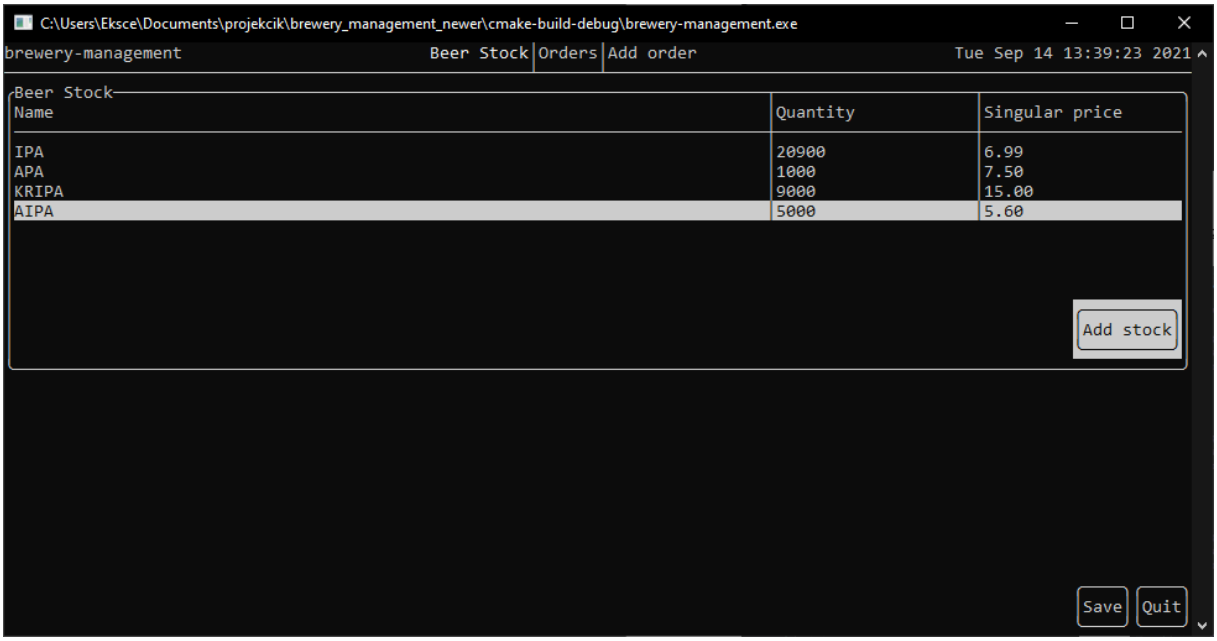
Guzik close zamyka okno dialogowe



Guzik Add Stock otwiera okno dialogowe dodawania nowej pozycji w stanach magazynowych.

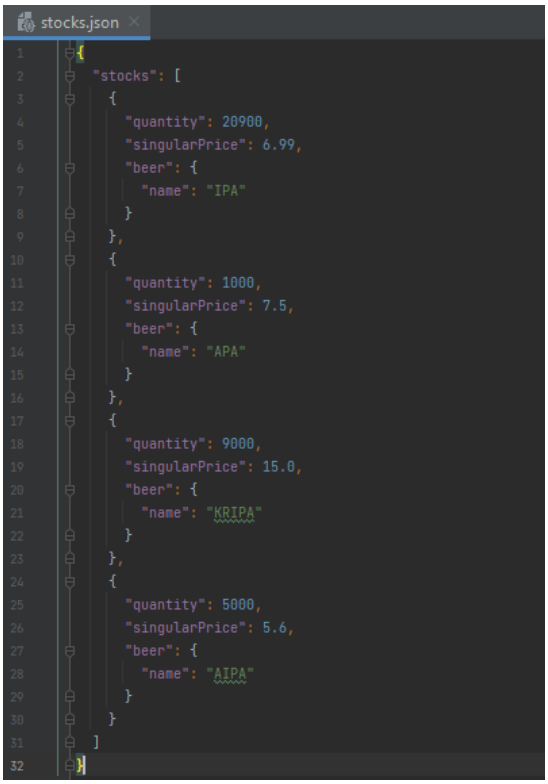



Wypełniamy inputy z danymi, a następnie guzikiem submit dodajemy nową pozycję.



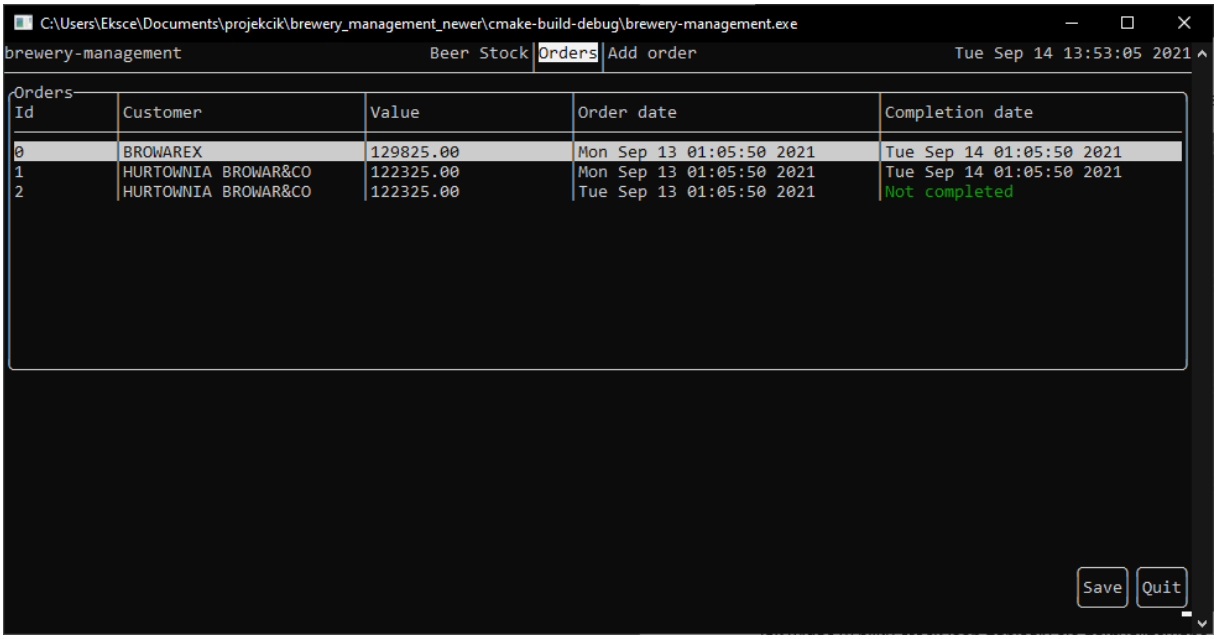

Nowo dodana pozycja na liście.

Dwa guziki (Save oraz Quit) służą kolejno do zapisywania zmian w plikach json oraz do zamknięcia aplikacji


Jak widać w pliku stock.json zapisały się zaktualizowane dane.

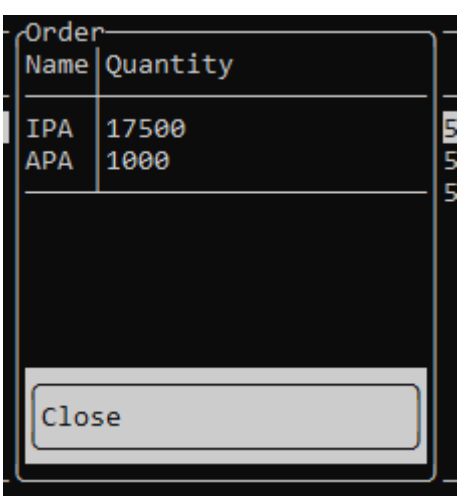
### Zakładka Orders

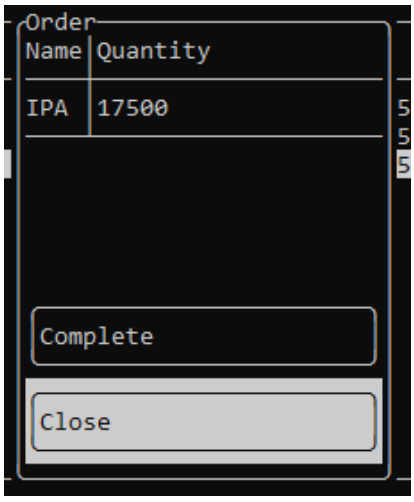



Działanie zakładki Orders jest bardzo zbliżone do jej poprzedniczki. Możemy wybierać pozycję z listy aby sprawdzić jakie piwa są wybrane w określonym zamówieniu oraz ich ilość.

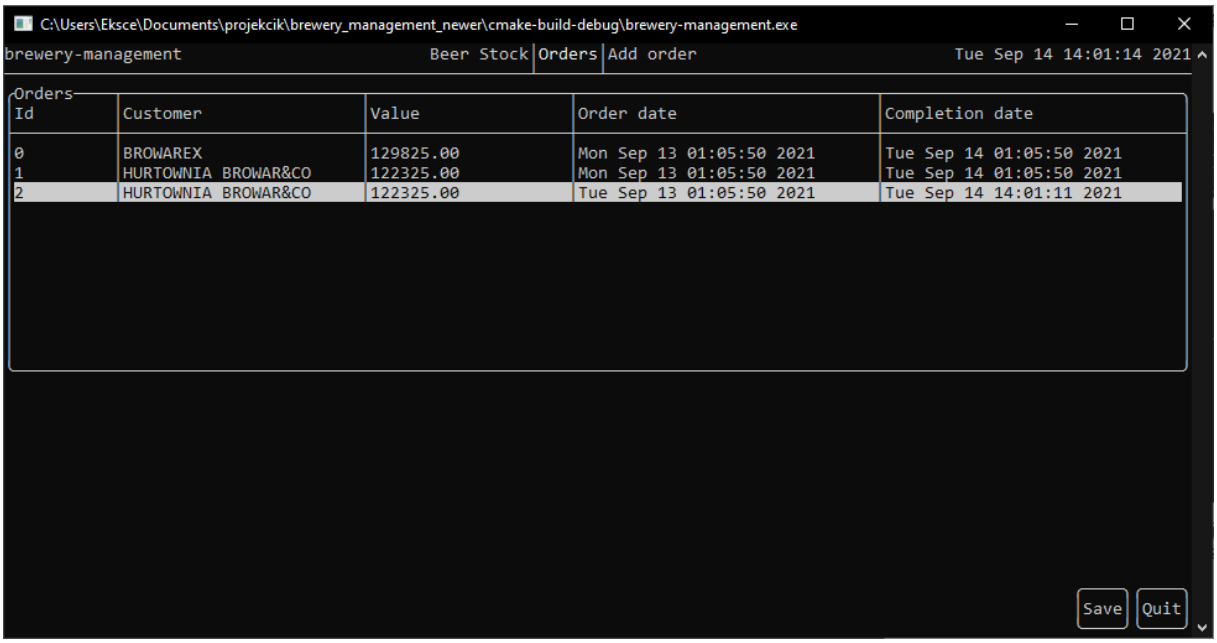
W tabeli dla danego zamówienia wyświetlane jest jego ID, klient, kwota ogólna zamówienia, data złożenia zamówienia oraz data skompletowania zamówienia.

Jeśli zamówienie nie jest skompletowane wyświetla się informacja Not completed.

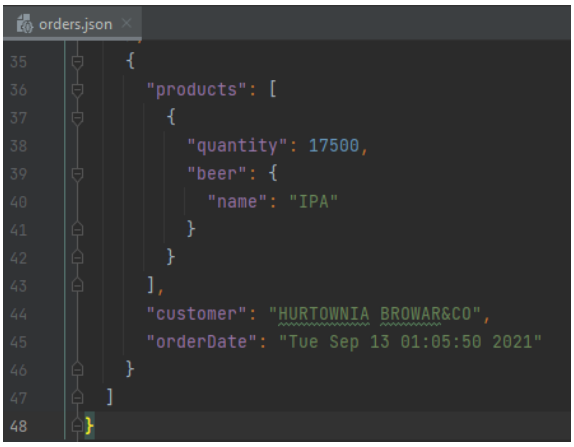


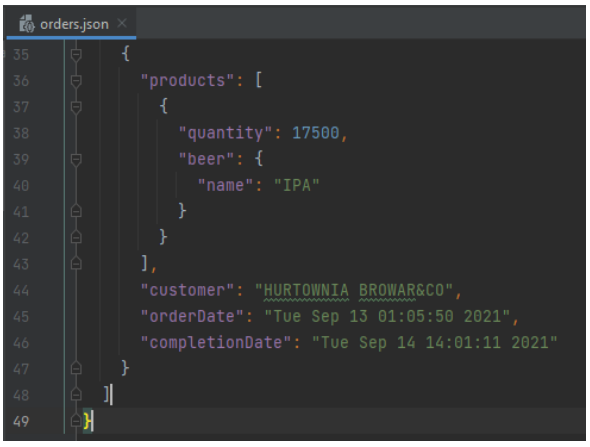




Jak widać zamówienia skompletowane (po lewej) oraz nie skompletowane (po prawej) różnią się tym że to drugie w oknie dialogowym ma opcję skompletowania.




Po użyciu tej funkcji jak widać zamówienie zostało skompletowane. Musimy jeszcze pamiętać o zapisaniu danych aby nie stracić wprowadzonych zmian.



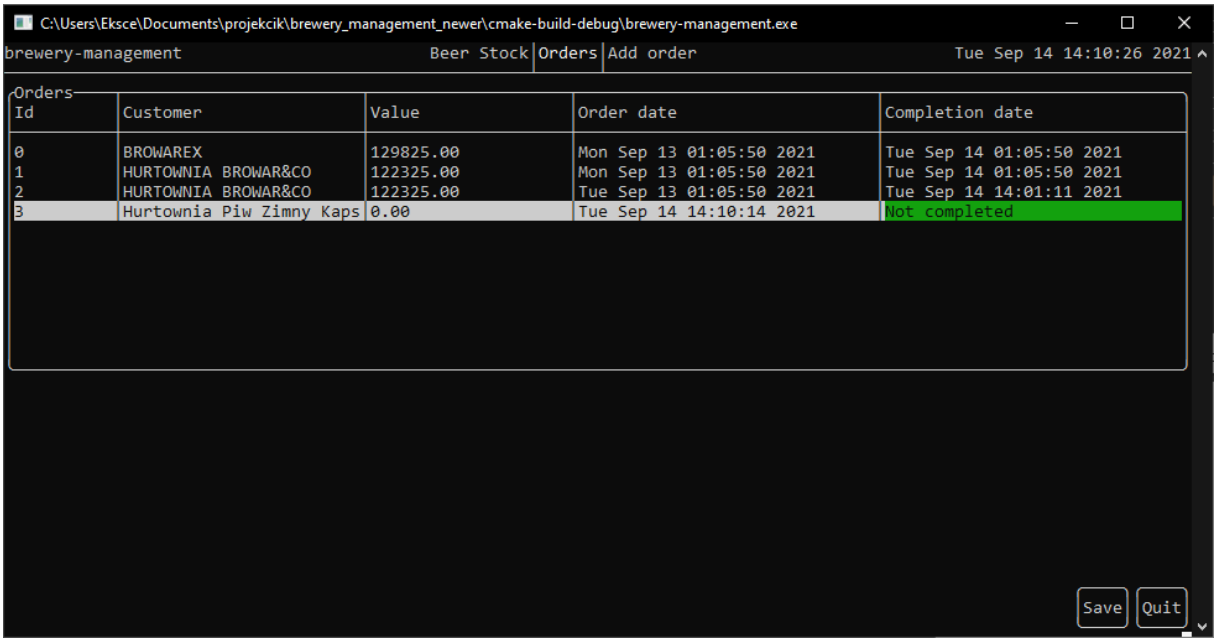




Po lewej zamówienie w pliku orders.json przed skompletowanej oraz po prawej zamówienie po wykonaniu operacji oraz zapisaniu zmian do pliku.

### Zakładka Add order


W tym oknie możemy dodać nowe zamówienia. W tym celu musimy wybrać rodzaj piwa, wpisać jego ilość, oraz nazwę firmy. Guzikiem Submit dodajemy zamówienie.

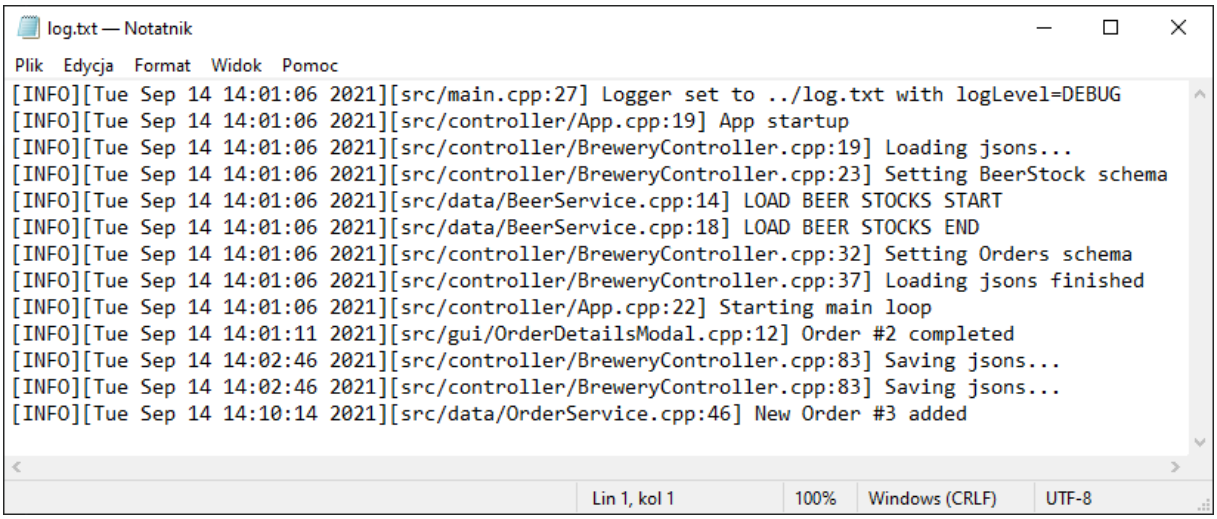

Przykładowo wypełniony formularz zamówienia.




Zamówienie dodało się do listy zamówień.

### Raportowanie działań z programu

Jak wspominaliśmy w punkcie 2. h) raportowaniem zajmuje się dyrektywa LOG. Aplikacja korzysta z niej w wielu miejscach i wszystkie działania programu zapisuję do pliku log.txt




Przykładowo raport na początku informuje gdzie są zapisywane informacje.

Wskazuje kiedy były ładowane pliki json.

Wskazuje kiedy aplikacja zaczęła renderować gui.

Wskazuje kiedy użytkownik skompletował zamówienie #2 oraz zapis danych i dodanie nowego zamówienia od ID #3.

# 4. Wnioski

Sama implementacja techniki obiektowej nie była problematyczna, ponieważ zespół sprawnie współpracował i pomagał sobie nawzajem, a ponadto w razie zderzenia ze ścianą pomagały nam zasoby z wykładów oraz sieci.

Decyzja o wybraniu FTXUI jako środowiska graficznego wyniknęła z faktu, iż łatwo jest je podpiąć pod projekt. Zdecydowaliśmy się na korzystanie z oprogramowania CLion do tworzenia/budowania aplikacji, które na etapie budowania aplikacji pozawala na linkowanie zasobów z zewnętrznych repozytoriów kodu, a takim jest właśnie środowisko graficzne FTXUI.

Kolejnym powodem dla którego zdecydowaliśmy się na IDE CLion jest fakt, że zespół pracował na komputerach z systemami z rodziny UNIX oraz Windows przez co najprostszym sposobem na unifikację środowiska zespołu było w przypadku UNIXa zainstalowanie CLion, a w przypadku Windowsa instalację CLiona oraz kompilatora GNU, który na linuxie jest preinstalowany wraz z systemem. Dzięki temu cały zespól mógł spokojnie dzielić się kodem i nie musiał martwić się o konflikty w kodzie z powodu różnych środowisk.