UNIWERSYTET GDAŃSKI
WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI

Kiedrowicz Kamil Stanisław
Kierunek: Informatyka
Specjalność: Tester Programista
Numer albumu: 238237
Kirsanau Aliaksei
Kierunek: Informatyka
Specjalność: Tester Programista
Numer albumu: 235353
Rybak Mark
Kierunek: Informatyka
Specjalność: Tester Programista
Numer albumu: 235381
Wardziński Marcin
Kierunek: Informatyka
Specjalność: Informatyka Ogólna
Numer albumu: 235423

Implementacja gry “Kupcy w Reganii”

Praca licencjacka napisana
pod kierunkiem dr Hanny Furmańczyk

Gdańsk 2018

Oświadczenie

My, niżej podpisani, oświadczamy, że przedłożona przez nas praca dyplomowa (będąca rezultatem pracy zespołu studentów) została przygotowana przez nas - w częściach, za które każdy z nas odpowiadał - samodzielnie, nie narusza praw autorskich, interesów prawnych i materialnych innych osób.

.....	.....
data	podpis
.....	.....
data	podpis
.....	.....
data	podpis
.....	.....
data	podpis

Streszczenie

Zakres pracy obejmuje projekt i implementację gry komputerowej "Kupcy w Reganii". Jest to prosta gra ekonomiczna dla jednego gracza, operująca na platformie Windows w trybie okienkowym, niewymagająca łączności z siecią. Polega ona na wymianie handlowej pomiędzy karawanami, którymi kieruje gracz, a rozsianymi po świecie miastami, oraz na przewozie towarów z miasta do miasta - celem gry jest zarabianie na różnicach cen poszczególnych towarów w różnych miastach. Dodatkowymi funkcjonalnościami są rozbudowa naszej karawany poprzez zakup nowych wozów i zatrudnienie kolejnych pracowników oraz możliwość utworzenia zupełnie nowej karawany. Cechami wyróżniającymi produkt na tle konkurencji są dynamicznie obliczane ceny towarów (oparte na zasadzie popytu i podaży), symulacja życia świata i pośredni wpływ gracza na jego kształt. W pracy opisano założenia programu, szczegóły jego implementacji oraz przebieg procesu testowania oprogramowania, od pierwszych testów pojedynczych elementów, przeprowadzanych w protezowym środowisku przed implementacją poszczególnych fragmentów do zasadniczego programu, przez testowanie poszczególnych funkcjonalności gotowego programu aż do testów przejrzystości interfejsu i jasności zasad gry przeprowadzanych przez przypadkowe osoby niezwiązane z projektem. Całość pracy została opatrzona zrzutami ekranu ukazującymi fragmenty kodu oraz interfejsu użytkownika, a także diagramami ilustrującymi architekturę programu oraz stojącej za nim bazy danych. Projekt został wykonany w języku C#, z wykorzystaniem bazy danych Microsoft SQL Server do przechowywania zapisanego stanu gry.

Spis treści

1.Opis projektu

Celem projektu było stworzenie ekonomicznej gry komputerowej polegającej na przewożeniu towarów pomiędzy miastami. Czynnikiem ją wyróżniającym miały być dynamicznie obliczane ceny, realistyczny model ekonomiczny, rozmaite zdarzenia losowe wpływające na świat oraz pośredni wpływ gracza na takowy - koronnym przykładem była kwestia wojny. Grający nie ma bezpośredniego wpływu na zmagania zbrojne - aczkolwiek sprzedaż broni jednej ze stron może przechylić szalę zwycięstwa.

Owocem projektu są "Kupcy w Reganii" - turowa gra dla jednego gracza, operująca na platformie Windows, okienkowa, nie wymagająca połączenia z internetem. Gracz, wcielając się w postać kupca żyjącego w paraśredniowicznych uniwersum fantasy, podróżuje pomiędzy miastami, kupując i sprzedając rozmaite towary, starając się zarobić na różnicach cen. Gra nie ma zdefiniowanego pozytywnego zakończenia - nie ma wyznaczonej ilości gotówki czy stanu świata, która powodowałaby wygraną. Gracz może kontynuować podróż tak długo, aż nie uzna że czas zakończyć przygodę, osiadając w jednym miejscu i używając zdobytego majątku lub, zakładając wersję mniej optymistyczną, aż nietrafione interesy nie doprowadzą go do bankructwa.

Projekt został wykonany z użyciem technologii C# i Microsoft SQL Server. Program w wersji elektronicznej został nagrany na płytę dołączoną do pracy. Ponadto projekt, wraz z pomocniczymi programami i dodatkowymi plikami dostępny jest w postaci repozytorium na serwisie GitHub, pod adresem https://github.com/CaravanProject01/projekt.

1.1 Porównanie z istniejącymi produktami

Pierwszymi z produktów jakie przychodzą na myśl są gry z serii Port Royale[1]. W wymienionych tytułach gracz wciela się w kupca z czasów kolonizacji Ameryki przez Europejczyków, gdzie główną różnicą pomiędzy wyżej wymienioną serią a naszą grą jest to, że w "Kupcach w Regani" w przeciwieństwie do Port Royale panuje system turowy.

[1] Port Royale: https://www.gry-online.pl/S016.asp?ID=2387

Drugą różnicą jest fakt, iż brak w naszym tytule jakichkolwiek zadań potocznie zwanymi "questami" oraz nie ma żadnego systemu symulującego walkę. Oznacza to, że nasza gra jest dosłownie w pełni zorientowana na handel pomiędzy lokacjami, więc nie ma możliwości przejmowania i łupienia miast.

Do podobieństw można zaliczyć rozwój lub regres miejscowości pod wpływem posiadanych dóbr i podobnie jak w produktach serii zapoczątkowanej przez firmę Ascaron Entertainment im bardziej rozwinięta jest dana osada, tym większa jest produkcja oraz zapotrzebowanie na określone surowce.

Drugim z kolei produktem jest gra Anno 1404[2], w której gracz zostaje założycielem piętnastowiecznej kolonii. Główną różnicą pomiędzy wyżej wymienionym tytułem a naszą grą jest fakt, że jej zasadniczym założeniem jest rozbudowa swojej osady, a handel występuje jedynie w formie narzędzia pomagającego osiągnąć ten cel. Odmienność ma miejsce także w przypadku zjawiska jakim jest rozgrywka w czasie rzeczywistym, która pojawia się także w uprzednio wspomnianej serii Port Royale.

Kolejnym przykładem podobieństw i różnic jest seria Heroes of Might and Magic[3], gdzie gracz kieruje bohaterami dowodzącymi armią, która ma za zadanie zniszczyć przeciwnika i najczęściej przejąć wszystkie jego miasta. Podobnie jak w "Kupcach w Regani" gra toczy się w trybie turowym, gdzie każda tura to jeden dzień i występuje przemijanie tygodni oraz miesiący. Ponadto przy przemijaniu wymienionych jednostek czasu jest szansa na wystąpienie losowych efektów, które mogą być zarówno pozytywne jak i negatywne dla gracza.

Natomiast, jeśli chodzi o rozbieżności, to brak w naszej grze bezpośredniego poruszania się po świecie gry, ponieważ mapa w "Kupcach w Regani" pełni funkcję pomocniczą przy zorientowaniu się jakie miejscowości leżą obok siebie najbliżej.

Kolejną różnicą w stosunku do marki Heroes of Might and Magic jest brak możliwości rozbudowy miast, co przekłada się na ingerowanie w świat gry jedynie w sposób pośredni, jakim może być inwestowanie i handel.

[2] Anno 1404: http://www.gram.pl/artykul/2009/07/01/anno-1404-recenzja.shtml
[3] Heroes of Might and Magic: https://www.gry-online.pl/gry-z-serii-i-podobne.asp?ID=68

Reasumując, “Kupcy w Reganii” jest to gra turowa oparta na kształtowaniu świata jedynie poprzez oddziaływanie na rynek.

2.Projekt i analiza

2.1 Podstawowe funkcjonalności

Jak przystało na grę ekonomiczną, podstawową i najważniejszą funkcjonalnością jest kupno i sprzedaż rozmaitych towarów. Niewiele mniej ważna jest podróż z miasta do miasta i możność odczytania rozmaitych danych o mieście i jego uwarunkowaniach ekonomicznych.

Kolejnym istotnym elementem jest możliwość rozwijania karawan, dokupowania do nich dodatkowych wozów, regulowania liczby zatrudnionych pracowników, a także tworzenia zupełnie nowych karawan.

Wśród funkcjonalności nie sposób nie wymienić także zapisu aktualnego stanu gry i odczytu stanu zapisanego, jak również możliwości rozpoczęcia nowej gry bez ingerencji w zapisany uprzednio stan. Istotnym faktem odnośnie zapisu jest to, że gra przewiduje przechowywanie tylko jednego zapisanego stanu gry.

Szczegółowy opis funkcjonalności zawarty jest w opisie interfejsu użytkownika.

2.2 Wymagania funkcjonalne i niefunkcjonalne

Najważniejszymi wymaganiami funkcjonalnymi które musiała spełniać nasza aplikacja są:

1. gra dla jednego gracza,
2. możliwość zapisu stanu gry i odczytu stanu uprzednio zapisanego,
3. GUI będące częścią programu, otwierające się jako okno na pulpicie,
4. możliwość rozpoczęcia nowej gry bez ingerowania w zapisany stan gry,
5. kończenie tury, co przesuwa zegar naprzód,
6. handel pomiędzy miastami a karawanami,
7. przemieszczanie karawan w podróż pomiędzy miastami,
8. powiększanie karawan o kolejne wozy i załogę,
9. tworzenie nowych karawan,
10. dynamiczne obliczanie cen towarów bazując na popycie i podarzy,
11. symulacja życia świata-zmiany populacji, wytwarzanie i zużywanie towarów,
12. występowanie zdarzeń losowych modyfikujących stan świata,
13. występowanie następujących po sobie wydarzeń wynikających ze stanu świata,
14. napady bandytów na karawany w trasie i możliwość obrony przed nimi,

15. możliwość wpływania na świat poprzez dostarczanie miastom potrzebnych zasobów lub ograniczanie ich obecności.

Poza tymi wymaganiami przed aplikacją postawiono kilka dodatkowych wymagań niefunkcjonalnych:

1. gra na komputery osobiste nie wymagająca dostępu do internetu,
2. przejrzystość interfejsu, umożliwiająca rozpoczęcie gry bez pomocy samouczka czy nauczyciela,
3. szybkie tempo działania gry, nie zmuszające do czekania na wykonanie obliczeń czy otwarcie kolejnych okien.

2.3 Diagram klas

2.3.1 Opis klas z namespace Caravans.model

Klasy te przechowują wszystkie dane, z których korzysta aplikacja oraz zawierają w sobie funkcje odpowiedzialne za nawiązywanie połączenia z bazą danych i transferem danych między bazą a aplikacją.

Klasa **TableArticle** jest odbiciem encji Article bazy danych. Posiada zmienne *Id* i *Name* typu string, oraz *Price*, *Production*, i *Requisition* typu int.

Klasa **TableArtInCaravan** jest odbiciem encji Art_In_Caravan bazy danych. Posiada zmienne *Id* i *IdArticle* typu string oraz *Number* typu int.

Klasa **TableArtInTown** jest odbiciem encji Art_In_Town bazy danych. Posiada zmienne *Id* i *IdArticle* typu string oraz *Number, Requisition* i *Production* typu int.

Klasa **TableCaravan** jest odbiciem encji Caravan bazy danych. Posiada zmienne *Id* i *IdLoc* typu string oraz *Wagons*, *Guard*, *Duration* i *Minions* typu int.

Klasa **TableLoc** jest odbiciem encji Locs bazy danych. Posiada zmienne *Id* i *Name* typu string.

Klasa **TableRoad** jest odbiciem encji Roads bazy danych. Posiada pola *Id*, *IdLoc_1*, *IdLoc_2* typu string oraz *Length* i *Quality* typu int.

Klasa **TableState** jest odbiciem encji State bazy danych. Posiada pola *Id*, *Name*, *Description* oraz *ShortDes*, wszystkie typu string.

Klasa **TableTown** jest odbiciem encji Town bazy danych. Posiada pola *Id*, *IdLoc* i *Name* typu string oraz *Population*, *Military*, *Prosperity* i *Food* typu int.

Klasa **TableTownState** jest odbiciem encji Town_State bazy danych. Posiada pola *Id* i *IdState* typu string oraz pole *Duration* typu int.

Poszczególne obiekty tych klas odpowiadają danym rekordom w encjach. Poza zmiennymi zawierają odpowiadające im "getery" i "setery".

Klasa **SQLreq** odpowiada za połączenie i wymianę danych pomiędzy modelami a bazą danych. Zawiera kilka zestawów podobnych do siebie metod operujących na listach obiektów różnych klas.

Metody wywołujące czytanie danych z bazy do modelu, to znaczy:

-ReadLocsToMod(List<TableLoc> Location)

-ReadArticleToMod(List<TableArticle> Article)

-ReadStateToMod(List<TableState> State)

-ReadRoadsToMod(List<TableRoad> Road)

-ReadTownToMod(List<TableTown> Town)

-ReadCaravanToMod(List<TableCaravan> Caravan)

-ReadTownStateToMod(List<TableTownState> TownState)

-ReadArtInCaravanToMod(List<TableArtInCaravan> ArtInCaravan)

-ReadArtInTownToMod(List<TableArtInTown> ArtInTown)

-ReadResourcesToMod(out int Gold, out int Turn)

Każda z nich przyjmuje jako argument listę obiektów odpowiedniej klasy (z wyjątkiem metody *ReadResourcesToMod*), z której dane są czytane. Tworzą się nowe elementy listy na podstawie danych, pobieranych poleceniem SELECT. Metoda *ReadResourcesToMod,* zwracająca 2 zmienne za pomocą OUT, odpowiada za odczytanie poziomu gotówki i numeru tury.

Metody wywołujące wyczyszczenie tabeli:

-*TruncateTownState()*

-*TruncateCaravan()*

-*TruncateArtInCaravan()*

Każda z tych metod usuwa zawartość odpowiedniej tabeli.

Metody wywołujące aktualizację tabeli:

-*UpdateTown(TableTown Town)*

-*UpdateCaravan(TableCaravan Caravan)*

-*UpdateTownState(TableTownState TownState)*

-*UpdateArtInTown(TableArtInTown ArtInTown)*

-*UpdateArtInCaravan(TableArtInCaravan ArtInCaravan)*

-*UpdateArticle(TableArticle Article)*

-*UpdateResources(int Gold, int Turn)*

Metody te przyjmują obiekty odpowiedniej klasy (poza *UpdateResources),* a następnie przekazują dane z tego obiektu do bazy danych, aktualizując odpowiedni rekord w tabeli. Ostatnia z nich przyjmuje ilość złota posiadaną przez gracza i numer tury i aktualizuje te dwie zmienne w bazie danych.

Klasa **Modele** jednoczy wszystkie elementy modelu. Zawiera listy obiektów klas: ***TableArticle, TableArtInCaravan, TableArtInTown, TableCaravan, TableLoc, TableRoad, TableState, TableTown, TableTownState.*** Posiada zmienne *Gold* i *Time* typu int (oraz odpowiednie do nich "getery" i "setery") i *SQLTest* - obiekt klasy **SQLreq**. Klasa zawiera kilka większych metod dotyczących połączenia z bazą danych, uruchamiających garść mniejszych metod, oraz kilka niedużych funkcji uruchamianych w trakcie gry.

Metoda *Load()* odpowiada za wczytanie danych z bazy danych do modelu. Usuwa ona zawartość list obiektów klas **Table[encja]**, a następnie wywołuje metody wczytujące dane z bazy danych do modelu.

Metoda *Repop()* odpowiada za wczytanie danych z plików do modelu. Usuwa ona zawartość list obiektów klas **Table[encja]**, a następnie wywołuje metody odczytujące dane z pliku i zapisujące je w modelu.

Metoda *Save()* zawiera inicjalizacje metod wywołujących aktualizację tabeli bazy danych, tym samym zapisując wartości aktualnej sesji.

Metody wczytywania danych do modelu:

-*ReadResources()*

-*ReadListLocation()*

-*ReadListArticle()*

-*ReadListState()*

-*ReadListRoad()*

-*ReadListTown()*

-*ReadListTownState()*

-*ReadListCaravan()*

-*ReadListArtInCaravan()*

-*ReadListArtInTown()*

Te metody wywołują odpowiednie funkcje z klasy *SQLreq* za pomocą obiektu *SQLtest* (dla *ReadListLocation()* zostanie użyta *SQLTest.ReadLocsToMod(List<TableLoc> Location)*, dla *ReadListTown()* - *SQLTest.ReadTownToMod(List<TableTown> Town)*, itd.).

Metody wczytujące dane z pliku:

-*ReadLocs_NG()*

-*ReadArticle_NG()*

-*ReadState_NG()*

-*ReadRoads_NG()*

-*ReadTown_NG()*

-*ReadCaravan_NG()*

-*ReadArt_in_caravan_NG()*

-*ReadArt_in_town_NG()*

Te metody wypełniają model danymi z plików zawartych w folderze *"..\NewGame_data"*. Są to dane o stanie świata na początku gry.

Metody wywołujące aktualizację tabeli:

-*UpdateListArticle()*

-*UpdateListArtInTown()*

-*UpdateListTownState()*

-*UpdateListCaravan()*

-*UpdateListTown()*

-*UpdateListArtInCaravan()*

-*UpdateResources()*

Te metody używają odpowiednie funkcje z klasy *SQLreq* za pomocą obiektu *SQLtest* (dla *UpdateListArtInTown()* zostanie użyta *SQLTest.UpdateArtInTown(TableArtInTown ArtInTown)*, dla *UpdateListArticle()* - *SQLTest.UpdateArticle(TableArticle Article)*, itd.).

Metoda *dodajKarawane(string a, string b)* - przyjmuje dwie zmienne typu string, jedną z których jest Id karawany, drugą (*IdLoc*) Id lokacji. Na podstawie tego tworzy obiekty typu **TableCaravan** oraz odpowiadającą mu listę obiektów **TableArtInCaravan**, które dodaje do odpowiednich list.

Metoda *KoniecStanu(string ids, string idm)* przyjmuje dwie zmienne typu string. Pierwsza z nich jest id stanu, druga - id miasta. Wyszukuje ona obiekt klasy **TableTownState** o takich id z listy *tableTownState,* a następnie usuwa go.

Metoda *NowyStan(string ids, string idm, int czas)* przyjmuje dwie zmienne typu string i 1 typu int. Pierwsza z nich jest id stanu, druga - id miasta, trzecia zaś czas trwania stanu. Na początku metoda sprawdza, czy obiekt klasy **TableTownState** o takich id istnieje już na liście *tableTownState*. Jeśli tak, to czas trwania zostanie zmieniony na podany w argumencie, a metoda zwróci wartość true. W przeciwnym przypadku utworzony zostanie nowy stan o odpowiednich wartościach, metoda zaś zwróci wartość false.

2.3.2 Opis klas z namespace Caravans.matma

Klasy te pośredniczą pomiędzy modelem przechowującym dane a oknami aplikacji, prezentującymi te dane graczowi. Spełniają one trzy podstawowe zadania - przekazują (odpowiednio przerabiając po drodze) dane z modelu do okien, realizują funkcje wywołane przez użytkownika (np kupowanie albo podróż) oraz wraz z czasem rozgrywki modyfikują część danych zawartych w modelu, symulując życie świata gry.

Klasa **ceny** tworzy obiekty składające się z czterech tablic. Jedna zawiera zmienne typu string i przechowuje id poszczególnych towarów, w trzech kolejnych, typu int, składowane są ceny tychże towarów (oddzielnie cena kupna i sprzedarzy) oraz ilość towaru w mieście. Jedyne metody w klasie to *GetCenaKup(string idt)*, *GetCenaSp(string idt)* i *GetIle(string idt)*. Przyjmują one zmienną string, będącą id towaru, i zwracające odpowiednią wartość z danej tablicy. Poza tym klasa zawiera rozbudowany konstruktor, przyjmujący jako argument zmienną typu string będącą id miasta. Konstruktor pobiera z klasy **model.Modele** informacje o mieście (populacja)

i o towarach w nim obecnych, tworząc obiekty klasy **towar**, wywołuje dla każdego z nich metodę *policzCena(int pop)*, po czym wypełnia wytworzonymi w ten sposób tablice, w taki sposób by informacje o danym towarze zajmowały miejsca o tym samym indeksie we wszystkich trzech tablicach.

Klasa **czas** zawiera cztery metody-*tura()*, *wyplata()*, *dzien()* i *tydzen()*, nie zwracające żadnych danych. Pierwsza z nich wywołuje się każdorazowo po wciśnięciu przycisku zakończenia tury. Zmienia ona zmienną time w klasie **model.Modele**, zwiększając jej wartość o jeden, dla każdego obiektu klasy **model.TableCaravan** na liście *Modele.tableCaravan* wywołuje metodę *ChangeDuration()*, po czym generuje losową liczbę z przedziału 1-100. Jeśli wylosowaną liczbą jest 1, wywołuje metodę *napad(string id)*. Następnie wywołuje metodę *wyplata()*, *dzien()* i, jeśli numer tury podzielny jest przez 7, *tydzen()*.

Funkcja *napad(string id)* jako argument przyjmuje id napadniętej karawany. W pierwszej kolejności pobiera z listy Modele.TableCaravans obiekt klasy **model.TableCaravan** o odpowiadającym przyjętemu id, po czym na bazie wartości zmiennych *Minions* i *Guard* w nim zawartych wylicza siłę bojową karawany. Następnie przeprowadzając serię losowań określa siłę bojową napastników, po czym porównuje te dwie wartości. W zależności od ich stosunku tworzy zmienną typu string, modyfikuje zmienne w listach *Modele.TableCaravans* i *Modele.tableArtInCaravan* oraz wywołuje metodę *napadliNas(string a)* z klasy **GamesWindow**, podając wytworzony wcześniej string jako argument.

Metoda *wyplata()* pobiera z obiektów klasy **model.TableCaravan** obecnych na wymienionej wcześniej tablicy zmienne *Guard* i *Minions*, zliczając na ich podstawie dzienne wydatki. Następnie zmniejsza zmienną *gold* z klasy **model.Modele** o wyliczoną sumę. Jeśli w ten sposób ilość twego złota spadnie poniżej zera, wywołana zostaje funkcja *zbankrutowales()* z klasy **GamesWindow**.

Funkcja *dzien()* odpowiada za obsługę stanów. W pierwszej kolejności tworzy szereg zmiennych pomocniczych, w tym listę obiektów klasy **model.TableTownState**, przechowującą stany zakończone oraz tablicę zmiennych string, przechowującą id zrujnowanych miast. Następnie przechodzi przez wszystkie obiekty **model.TableTownState** na liście *Modele.tableTownState*. Każdemu zmniejsza wartość zmiennej Duration o jeden, po czym sprawdza czy ta liczba spadła do zera. Jeśli tak, modyfikuje rozmaite dane z obiektów klasy **model.TableArtInTown** obecnych

na liście *Modele.tableArtInTown*, w zależności od id stanu. Jeśli zakończony stan miał id ST10 lub ST06, co odpowiada oblężeniu i pożarowi, dodaje id tego miasta do tablicy miast zrujnowanych. Ponadto do listy stanów zakończonych dodaje atrapowy stan o identycznych id.

Jeśli czas nie spadł jeszcze do zera, w zależności od id stanu może dojść do zmian w obiektach wymienionej wyżej klasy lub też klasy **model.TableTown**, zawartych w tabeli *Modele.tableTown*. W przypadku innych stanów nic się nie dzieje.

Kolejnym krokiem jest finalne usunięcie stanów zakończonych. Dla każdego atrapowego stanu na liście stanów kasowanych wyszukuje się odpowiedni obiekt na liście *Modele.tableTownState*, a następnie go usuwa.

Ostatnim fragmentem jest dodanie nowych stanów. Dzieje się to w kilku etapach. Najpierw pobiera się liczne zmienne z obiektów klasy **model.TableTown**, zawartych na odpowiedniej liście. Na bazie wartości zmiennych *Prosperity* i *Military* określa się czy zaistniały stany złotej ery, wojny i wojennych przygotowań. Następnie dla każdego miasta, którego id znalazło się na liście miast zniszczonych, wywołuje się stan odbudowy. Ostatnim krokiem jest sprawdzenie wystąpienie stanów losowych - dla każdego miasta losowana jest liczba z zakresu 1 do 100 - wyniki od 1 do 6 powodują wystąpienie odpowiednio zarazy, suszy, nieurodzaju, obfitych bądź bardzo obfitych plonów lub też pożaru. W przeciwieństwie do stanów wyliczanych, które mają z gry zdefiniowaną długość, czas trwania stanu losowego jest ponownie losowany z przedziału 5-15, przy czym czas trwania pożaru jest zmniejszony o połowę.

```
case 2:
    flaga = Modele.NowyStan("ST02", idm, czas2);  //susza
    if (flaga == false) {
        foreach (TableArtInTown tow in Modele.tableArtInTown)
        {
            idt = tow.GetIdArticle();
            idm2 = tow.GetId();
            if (idt == "TO02" && idm == idm2)   //jabłka
            {
                licznik = tow.GetProduction();
                licznik -= 50;
                tow.SetProduction(licznik);
            }
            if (idt == "TO05" && idm == idm2)   //chleb
            {
                licznik = tow.GetProduction();
                licznik -= 50;
                tow.SetProduction(licznik);
            }
        }
    }
    break;
```

Rysunek 2: Fragment metody *dzien()* - rozpoczęcie stanu suszy powoduje spadek produkcji jabłek i chleba. [opracowanie własne]

Każdorazowo powstanie nowego stanu oznacza wywołanie funkcji *Modele.NowyStan(string ids, string idm, int czas)*. Jeśli zwróci ona wartość false, oznaczającą że tego stanu nie było wcześniej na liście *Modele.tableTownState*, zostają zmodyfikowane odpowiednie wartości obiektów z tabeli *Modele.tableArtInTown* - podobnie jak w przypadku usuwania stanu, tylko że w drugą stronę. Przykładowo - rozpoczęcie stanu suszy zmniejsza modyfikator produkcji jabłek i chleba o 50 punktów. Zakończenie tego stanu zwiększa te wartości o 50, przywracając do podstawowej wartości.

Metoda *tydzien()* przechodzi przez wszystkie obiekty klasy **model.TableTown** w liście *Modele.tableTown*. Dla każdego tworzy obiekt klasy **miasto**, po czym wywołuje dla niego kolejno metody *wypelnij(string id)*, *policzTowary()* i *zmianaPopulacji()*. Następnie ustawia zmienen Population, Military, Prosperity i Food z klasy **model.TableTown** (za pomocą setterów) odpowiadającymi wartościami pobranymi getterami z obiektu klasy **miasto**, i wywołuje metodę *dajDane()*, która przekazuje resztę informacji do modeli.

Klasa **miasto** tworzy obiekty składające się ze zmniej id typu string, czterech zmiennych typu int: *populacja*, *gotowosc*, *zywnosc* i *dobrobyt* oraz listy obiektów klasy **towar**, nazwanej *towary*. Zawiera konstruktor, który przyjmuje wszystkie pięć tych wartości oraz "gettery" zwracające wartości int. Zawiera także szereg funkcji.

Metoda *wypelnij(string id)* przyjmuje id miasta i na jego podstawie wybiera odpowiednie wartości z obiektów klas **model.TableArticle** i **model.TableArtInTown** zawartych w listach *Modele.tableArticle* i *Modele.tableArtInTown* i na ich podstawie tworzy obiekty klasy **towar**, które dodawane są do listy towary.

Funkcja *zmianaPopulacji()* oblicza jak zmienia się populacja miasta, bazując na wartości zmiennej *zywnosc*. Początkowo wywołuje metodę *policzZywnosc()*.

Sama metoda *policzZywnosc(int popu)* oblicza wartości zmiennej *zywnosc*, bazując na wybranych danych z listy towary-konkretnie za ilości i zapotrzebowaniu na mięso, jabłka i chleb. Korzysta ona z funkcji *policzZapotrzebowanie(int pop)* i *dajIlosc()* z klasy **towar**.

Funkcja *policzTowary()* dla każdego obiektu klasy **towar** na liście towary wywołuje metodę *zmianaIlosci(int pop)* i w zależności od jej wyniku modyfikuje

wartość zmiennej *dobrobyt*. Jak można się domyśleć z nazw, funkcja symuluje zmianę ilości towarów w czasie - produkcję i zużycie.

Ostatnia jest metoda *dajDane()*, która wywołuja funkcję *wyprowadzDane(string idm)* dla każdego obiektu klasy **towar** na liście towary. Powoduje to zwrot policzonych danych do modeli.

Klasa **przekaznik** spełnia głównie pośrednika między modelami i oknami, większość jej metod jest wyjątkowo prosta - pobiera dane z modelu i zwraca jako wynik. Wszystkie metody tej klasy są statyczne. Funkcje *DajCzasS()* i *DajCzas()* oddają zmienną Modele.time jako stringa lub inta, *DajKaseS()* i *DajKase()* czynią to samo ze zmienną Modele.gold. Podobnie prosta jest metoda *dajKarawany()*, która zwraca listę zmiennych typu string, będącymi id kolejnych karawan (obiektów klasy **model.TableCaravan** z listy *Modele.tableCaravan*).

Niewiele bardziej skomplikowane są metody *dajNazwe(string id)* i *DajPopulacje(string id)*. Przyjmują one id miasta, wyszukują na liście *Modele.tableTown* obiekty klasy **model.TableTown** o identycznym id, pobierają wartość zmiennej *Name* lub *Population* (w zależności od metody) i zwracają jako wynik-w pierwszym przypadku jako string, w drugim jako int.

Podobnie działają funckje *dajWozy(string id)*, *dajOchrone(string id)*, *dajPomagierow(string id)* i *CzasPodrozy(string id)*. Przyjmują one id karawany i wyszukują z listy Modele.tableCaravan obiekt klasy **model.TableCaravan** o odpowiadającym id. Następnie pobierają z niego zmienne *Wagons*, *Guard*, *Minions* lub *Duration* (w zależności od metody) i zwracają jako wynik.

Metoda *IleTowaru(string idk, string idt)* jest już bardziej skomplikowana. Przyjmując id karawany i towaru, wyszukuje odpowiedni obiekt klasy **model.TableArtInCaravan** w liście *Modele.tableArtInCaravan* i zwraca wartość zawartej w nim zmiennej *Number* - ilość danego towaru przewożoną przez daną karawanę.

Metody *PoliczObciozenie(string id)* i *PoliczPojemnosc(string id)* przyjmują jako argument id karawany. Pierwsza wyszukuje z listy *Modele.tableArtInCaravan* wszystkie obiekty klasy **model.TableArtInCaravan** o identycznym id i sumują wartości zawartych w nich zmiennych *Number* - suma zostaje zwrócona jako wynik funkcji. Druga wybiera odpowiedni obiekty klasy **model.TableCaravan** i pobiera

wartość jego zmiennej *Wagons*. Pomnożona przez pojemność wozu (200) liczba ta staje się wynikiem metody.

Funkcja *lokalizuj(string idk)* przyjmuje id karawany i na jej podstawie odszukuje odpowiedni obiekt klasy **model.TableCaravan** (z listy *Modele.tableCaravan* oczywiście), a następnie pobiera wartości zmiennych *IdLoc* i *Duration*. Następnie wyszukuje w liście *Modele.tableTown* obiektu który posiada identyczne *IdLoc*, pobierając wartość jego zmiennej *Name*. Następnie przy pomocy owej zmiennej i pobranej wcześniej zmiennej *Duration* generuje zmienna typu string opisującą aktualną lokację karawany, a następnie ją zwraca.

Funkcja *lok(string idk)* działa podobnie do poprzedniczki, z tą różnicą, że przeszukując obiekty klasy **model.TableTown** zamiast nazwy pobiera zmienną *Id*, a następnie wprost ją zwraca.

Bardziej skomplikowane są metody *dajWojo(string id)*, *dajZarcie(string id)* i *dajBogactwo(string id)*. Wszystkie przyjmują id miasta, po czym wyszukują odpowiedni obiekt na liście *Modele.tableTown*, po czym wyciągają z niego zmienne *Military*, *Food* lub *Prosperity*. Następnie na ich podstawie tworzą stringa zawierającego fabularny opis pobranej wartości liczbowej, który jest następnie zwracany.

```
. odwołanie | wardasz, 30 dni temu | 1 autor, 1 zmiana
ublic static string dajBogactwo(string id)
{
    int hajs=0;
    foreach (TableTown miasto in Modele.tableTown)
    {
        if (miasto.GetId() == id) hajs = miasto.GetProsperity();
    }
    if (hajs < -100) return "bida z nędzą";
    if (hajs < -50) return "bieda";
    if (hajs < -0) return "ubióstwo";
    if (hajs > 200) return "burżujstwo";
    if (hajs > 100) return "bogactwo";
    if (hajs > 50) return "nadmiar zbytków";
    return "przeciętna";
}
```

Rysunek 3: Kod metody dajBogactwo(string id). [opracowanie własne]

Podobnie działa funkcja *dajStany(string id)*. Przegląda ona jestak listę *Modele.tableTownState*. Z każdego obiektu klasy **model.TableTownState**, który dzieli to samo Id co podane jako argument pobierane jest jego *IdState*, po czym z listy *Modele.tableState* pobierana jest nazwa stanu (zmienna *Name*) o takim samym Id. Następnie nazwa ta dodawana jest do stringa, będącego opisową listą stanów miasta, który następnie jest zwracany.

Jeszcze obszerniejsza jest metoda *dajInfo(string idt, string idm, int populacja)*. Przyjmując id miasta i towaru oraz populację miasta. Tworzy ona obiekty klasy **towar**, pobierając odpowiednie dane z list *Modele.tableArticle* i *Modele.tableArtInTown*, po czym wywołuje na nich funkcje *policzZapotrzebowanie(int pop)* i *policzProdukcje(int pop)*. Bazując na ich wynikach oraz zmiennej *ilosc* tworzy zmienną string opisującą sytuację rynkową odnośnie danego towaru w danym mieście, po czym zwraca owego stringa jako wynik.

Ostatnia jest metoda *czyMoznaPodrozowac(string id)*. Przyjmuje ona id karawany, wyszukuje odpowiedni obiekt klasy **model.TableCaravan** i sprawdza, czy suma ilości pomocników (zmienna *Minions*) i ochroniarzy (zmienna *Guard*) powiększona o jeden równoważy lub przewyższa podwojoną ilość wozów (zmienna *Wagons*). Jeśli tak, zwraca wartość true, jeśli nie - false.

Klasa **towar** tworzy obiekty zawierające pola *id*, *ilosc*, *cenaDef*, *produkcjaDef*, *produkcjaMod*, *zapotrzebowanieDef*, *zapotrzebowanieMod* oraz *cenaKup* i *cenaSprzed*, Przy czym pierwsza jest typu string, reszta int. Konstruktor przyjmuje dane odpowiadające wszystkim polom poza dwoma ostatnimi, które początkowo zostają ustawione na zero. Ponadto klasa zawiera pięć funkcji.

Metoda *policzZapotrzebowanie(int pop)* przyjmuje populację miasta wyrażoną w setkach. Zwraca ilość towaru, którą miasto tej wielkości będzie potrzebowało co tydzień.

Metoda *policzProdukcje(int pop)* działa podobnie, z tym że zamiast ilości potrzebnego towaru, zwraca ilość towaru, która zostanie wyprodukowana w tym okresie.

Funkcja *policzCena(int pop)* jest bardziej skomplikowana. Przede wszystkim wywołuje ona poprzednie dwie metody, po czym oblicza stosunek ich wyników. Na jego podstawie liczone są ceny kupna i sprzedaży towaru.

Metoda *zmianaIlosci(int pop)* ponownie jest dużo prostsza. Ponownie wywołuje dwie pierwsze funkcje, po czym zmniejsza ilość towaru o zapotrzebowanie. Jeśli liczba ta spadnie poniżej zera, zerowa początkowo zmienna *wynik* zostaje obniżona o jeden. Następnie dodawana do ilości towaru jest wartość towaru wyprodukowanego - jeśli wcześniej ilość spadła poniżej zera, liczba ta jest zmniejszana, aby pokryć zaległe zapotrzebowanie. Jeśli zaległość jest tak duża, że nic nie zostało dodane, *wynik* spada jeszcze bardziej. Po drodze ilość towaru może zostać zmodyfikowana - ostatecznie nie

może być przecież ujemnej ilości towaru w mieście. Dodatkowo, na koniec metody sprawdzany jest stosunek ilości towaru w mieście do zapotrzebowania. Jeśli obecnie zgromadzone zapasy są w stanie pokryć zapotrzebowanie na kolejnych pięć tygodnie (zakładając, że zapotrzebowanie nie ulegnie zmianie) *wynik* zostanie podniesiony. Oczywiście to właśnie zmienna *wynik* jest zwracanym rezultatem funkcji.

Ostatnia jest metoda *wyprowadzDane(string idm)*. Przyjmując id miasta, wyszukuje ona obiekt klasy *model.TableArtInTown* o identycznym id, po czym wprowadza do niego nowe wartości zmiennych *Number*, *Requisition* i *Production*.

Klasa **warsztat** skupia statyczne metody obsługujące okno warsztatu. Najprostsze są niezwracające żadnego wyniku metody *najmijPomoc(string id)* i *najmijOchrone(string id)*. Obie przyjmują id karawany, wyszukują odpowiadający mu obiekt klasy **model.Caravan** i zwiększają o jeden zawartą w nim zmienną *Minions* lub *Guard*.

Podobnie, choć w drugą stronę, działają metody *zwolnijPomoc(string id)* i *zwolnijOchrone(string id)*. Poza kierunkiem zmiany wartości istotną różnicą jest także kontrola i zwracanie wartości typu boolean. Przed zmianą wartości metoda sprawdza, czy nie jest ona przypadkiem równa zeru. Jeśli jest, zwraca wartość false, jeśli nie - obniża o jeden i zwraca true.

Kolejna jest funkcja *kupWoz(string id)*, także zwracająca wartość typu boolean. Przede wszystkim sprawdza ona wartość zmiennej *Modele.gold*-jeśli jest ona niższa od 200, zwraca wartość false. Jeśli nie, wyszukuje obiekt klasy **model.Caravan** odpowiadający podanemu jako argument id i zwiększają o jeden zawartą w nim zmienną *Wagons* oraz zmniejsza zmienną *Modele.gold* o 200.

Ostatnia jest metoda *nowaKarawana(string id)*. Ponownie początkowo sprawdza wartość zmiennej *Modele.gold*, tym razem jednak wartością graniczną jest 500, nie 200 - jeśli zmienna ma niższą wartość, zwracane jest false. W przeciwnym przypadku zmniejszą tą zmienną o 500. Następnie liczy ilość obiektów na liście *Modele.tableCaravans* i na podstawie tej liczby tworzy id nowej karawany. Kolejnym krokiem jest odnalezienie obiektu klasy **model.Caravan** odpowiadający podanemu jako argument id i wydobycie z niego zmiennej *IdLoc*. Ostatnim krokiem jest wywołanie metody *Modele.dodajKarawane(string a, string b)*, podając jako atrybuty wytworzone id nowej karawany i wydobyte id lokacji (nowa karawana powstaje w mieście, w którym jest karawana, którą obecnie obsługujemy) oraz zwrócenie wartości true.

Klasa **gra** odpowiedzialna jest za wywoływanie wywoływanie przepływu danych między modelami a bazą danych. Zawiera ona obiekt klasy **Modele** oraz trzy proste funkcje:

Metoda *nowa()* wywoływana jest gry rozpoczniemy nową grę. Wywołuje ona metodę *Repop()* z klasy **Modele**.

Funkcja *wczytaj()* wywoływana jest gry chcemy kontynuować zapisaną wcześniej grę. Wywołuje ona metodę *Load()* z klasy **Modele**.

Metoda *zapisz()* wywoływana jest gdy chcemy zapisać aktualny stan gry. Wywołuje ona metodę *Save()* z klasy **Modele**.

Klasa **handel** zawiera dwie metody: *kup(string IDkarawana, string IDmiasto, string IDtowar, int ile, int cena)* i *sprzedaj(string IDkarawana, string IDmiasto, string IDtowar, int ile, int cena)*.

Pierwsza funkcja bazując na pobranych argumentach wylicza łączną kwotę transakcji i porównuje ją z majątkiem gracza. Jeśli gracz posiada odpowiednią ilość złota, wyszukuje obiekty klas **model.TableArtInCaravan** i **model.TableArtInTown.** Następnie liczy pojemność i obciążenie karawany, po czym sprawdza czy pożądana ilość towaru znajduje się w mieście, jeśli tak to odpowiednio zmienia ilości złota oraz ilości towaru zarówno w mieście jak i w karawanie. Funkcja ta zwraca zmienną typu string, o treści zależnej od od przebiegu operacji.

-”done” - jeśli operacja przebiegła bez problemów

-"Ten towar nie jest na sprzedaż" - W przypadku gdy dany towar nie jest na sprzedaż (jest go za mało w stosunku do zapotrzebowania)

-"Za mało złota!" - W przypadku gdy gracz ma zbyt małą ilość złota

-"Zbyt mała pojemność karawany!" - W przypadku gdy karawana ma zbyt małą pojemność

-"Brak/za mało towaru w miescie!" - W przypadku gdy chcemy kupić więcej towaru niż jest obecne w mieście

Druga metoda wyszukuje w liście *Modele.tableArtInCaravan* odpowiedni obiekt klasy **model.TableArtInCaravan** i sprawdza czy karawana posiada odpowiednią ilość towaru. Następnie po wyszukaniu obiektu klasy **model.TableArtInTown** modyfikuje ilość złota gracza oraz ilość towaru w mieście i karawanie. Funkcja ta, podobnie jak

poprzednia, zwraca napis “done” jeśli wszystko pójdzie po dobrej myśli, natomiast jeśli nie ma dość dużo danego towaru w karawanie, zwraca napis: "Niewystarczajaca ilość towaru w karawanie!".

Klasa **podroz** posiada tylko jedną funkcję o nazwie *podrozdo(string miastoid, string karawanaid)*, która przyjmujeid miasta docelowego oraz id wybranej karawany i liczy ona najkrótszą odległość pomiędzy lokacjami wykorzystując przy tym algorytm Dijkstry. Dana metoda składa się z czterech faz:

- Pierwszą jest inicjalizacja zmiennych oraz tablicy potrzebnej do wyliczania odległości.
- W przypadku drugiej wyliczane są odległości do miast sąsiadujących z aktualną lokalizacją karawany, po czym wybiera lokację, do której ma najbliżej i ustawia ją jako obliczoną.
- W następnej fazie analogicznie do poprzedniej wylicza odległości do reszty wierzchołków.
- Ostatnią fazą jest wpisywanie id lokacji docelowej oraz czasu podróży do karawany.

Klasa **Tablicadoliczenia** zawiera informacje potrzebne przy tworzeniu tablicy wykorzystywanej do liczenia najkrótszej drogi pomiędzy miastami. Posiada ona następujące prywatne pola: string *IdLoc*, int *czas*, int *done*, string *poprzIdloc* i oznaczają one kolejno: *IdLoc* - id lokacji, *czas* - czas potrzebny do dotarcia do niej z lokacji zawartej w karawanie, *done* - informację o tym czy odległość do danej lokacji już jest według algorytmu obliczona oraz *poprzIdloc* - id lokacji, która jest poprzednim wierzchołkiem grafu w liczonej trasie.

2.3.3 Opis klas z namespace Caravans

Klasy te są klasami częściowymi dziedziczącymi po klasie systemowej **Window**. Każda z nich odpowiada jednemu oknu programu, obsługując przyciski i przygotowując dane do wyświetlenia. Wiele metod (wszystkie reagujące na wciśnięcie przycisku) przyjmuje argumenty “*(object sender, RoutedEventArgs e)”* - dla skrócenia opisu będzie to zastąpione frazą *“(kliknięcie)”*.

Klasa **authors** obsługuje okno prezentujące autorów projektu. Racji wyjątkowej prostoty i niezmienności tego okna zawiera jedynie konstruktor i funkcję *bexit_Click(kliknięcie),* obsługującą przycisk zamknięcia okna. Nie robi ona nic innego poza tym właśnie - zamyka okno.

Klasa **Errors** obsługuje okno komunikatu błędu. Klasa zawiera dwie zmienne typu string - tytuł okna i zasadniczą wiadomość. Zawiera także dwa konstruktory - jeden przyjmuje jedną zmienną klasy string, która staje się wiadomością, tytuł zaś otrzymuje domyślną wartość "Pomyłka!". Drugi zaś przyjmuje dwie zmienne, wpisując je w obie zmienne klasy. Do tego klasa zawiera dwie właściwości typu string, które przekazują treść zmiennym do okna. Klasa zawiera jedną metodę *Exiterror_Click(kliknięcie)*. Wywołuje się ona w momencie naciśnięcia przycisku zamknięcia okna i powoduje jego zamknięcie.

Klasa **GamesWindow** obsługuje główne okno gry. Zawiera ona trzy obiekty, po jednym z klas **WaggonShop**, **MainWindow** i **MiastoTour**, oraz dwie zmienne typu string wraz z odpowiadającym im właściwościami. Dodatkowo znajduje się w niej także pomocnicza zmienna typu Boolean *kar*. Posiada tylko jeden konstruktor, który inicjalizuje wszystkie elementy okna. Jako że okno posiada cztery przyciski, klasa zawiera cztery metody im odpowiadające: *Bmenu_Click(kliknięcie)*, *BWaggon_Click(kliknięcie)*, *Bend_Click(kliknięcie)* i *Save_Click(kliknięcie)*. Pierwsza wywoływana jest przyciskiem wyjścia do menu głównego-zamyka to okno otwierając w zamian menu główne. Druga odpowiada przyciskowi "karawany", otwierając okno przeglądu karawan, zmieniając wartość zmiennej *kar* - otwarcie okna powoduje, że można je odświeżyć. Trzecia wywoływana jest wciśnięciem przycisku końca tury. Wywołuje ona metodę *tura()* z klasy **matma.czas** oraz funkcję *odswiez()*, o której będzi niżej. Ostatnia z metod przycisków wywołuje się w momencie wciśnięcia przycisku zapisu i wywołuje metodę *zapisz()* z klasy **matma.gra**.

Ponadto klasa zawiera pięć zwykłych metod, wywoływanych z poziomu kodu. Pierwsza z nich to *z1()*, powodująca zamknięcie okna przeglądu karawan, druga to *odswiezKarawane()*, która dla odmiany odświeża informacje w tym oknie poprzez wykonanie dla niego metody *odswiez()* (jest to metoda klasy **WaggonShop**). Trzecia jest metoda o tej samej nazwie. Metodami *DajKaseS()* i *DajCzasS()* z klasy **przekaznik**

pobiera ilość gotówki i numer tury i wpisuje je w odpowiednie pola w oknie, po czym, jeśli to możliwe, wywołuje metodę *odswiezKarawane()*.

Kolejna jest metoda *napadliNas(string a)*. Przyjmuje ona komunikat informujący o skutkach napadu i tworzy okno komunikatu błędu, korzystając z konstruktora dwuargumentowego - jako pierwszy argument (tytuł) podaje wartość “Napad!!!”, jako drugi przekazuje przyjęty komunikat.

Ostatnia jest metoda *zbankrutowales()*. Ona także wywołuje okno komunikatu błędu, podając jako pierwszy argument wartość "Przegrana" a jako drugi "Złoto się skończyło, zbankrutowałeś". Poza tym zamyka okno gry i otwiera okno menu głównego.

Klasa **MainWindow** obsługuje okno menu głównego. Zawiera prosty konstruktor i żadnych właściwości lub zmiennych, a jedynie cztery metody obsługujące przyciski i piątą wywoływaną z kodu. Ta ostatnia nazywa się *odzwierzGlowne()* i powoduje wywołanie funkcji *odswiez()* z klasy **GamesWindow.** Metody wywołujące się po wciśnięciu przycisku to:

Bauthors_Click(kliknięcie), która tworzy i wywołuje okno autorów.

nowa_click(kliknięcie), która tworzy i wywołuje główne okno gry, oraz wywołuje metodę *nowa()* z klasy **matma.gra.**

nowa_click(kliknięcie) działa podobnie, ale zamiast funkcji **nowa()** wywołuje funkcję *wczytaj()* z tej samej klasy.

Ostatnia jest metoda *Bexit_Click(kliknięcie)*, która powoduje wyłączenie całego programu.

Klasa **MiastoTour** obsługuje okno informacji o mieście. Klasa zawiera cały zestaw zmiennych typu string i odpowiadających im właściwości: nazwę miasta, liczbę populacji, fabularne opisy poziomu dobrobytu, żywności, gotowości bojowej, stanów miasta i sytuacji odnośnie każdego z towarów, a także pojedynczą, pozbawioną właściwości zmienną *id*, także typu string. Konstruktor klasy przyjmuje zmienną string będącą id miasta, wywołuje metodę *zassaj()* i inicjalizuje wszystkie elementy okna. Sama metoda *zassaj()* wywołuje metody *dajNazwe(string id), DajPopulacje(string id)* i *dajInfo(string idt, string idm, int populacja)* z klasy **matma.przekaznik**, wypełniając zmienne informacjami o mieście. Ponadto w klasie znajduje się jedna metoda

wywoływana poprzez wciśnięcie przycisku-*ExitM_Click(kliknięcie)*. Powoduje ona zamknięcie okna.

Klasa **Podroz** obsługuje okno podróży. Klasa zawiera jedną zmienną, *id*, typu string. Prosty konstruktor przyjmuje wartość id karawany i przypisuje ją do tej zmiennej. Zasadniczą częścią jest cały zestaw funkcji wywoływanych przez użytkownika, obsługujące przyciski podróży do konkretnych miast. Każdy z nich wywołuje funkcję *podrozdo(string miastoid, string karawanaid)*, z klasy **matma.podroz** podając jako pierwszy argument zapisane w kodzie id miasta (inne w każdej funkcji), jako drugi zaś id karawany zapisane w zmiennej id. Następnie wywołuje funkcję *z1()* z klasy **GamesWindow** i zamyka okno. Dodatkową funkcją jest metoda *exitP_Click(kliknięcie)*, która po prostu zamyka okno podróży.

Klasa **WaggonShop** obsługuje okno przeglądu karawan. Zawiera cały zestaw zmiennych typu int - ilości każdego towaru w karawanie, jej id, pojemność i obciążenie. Dla każdej z nich poza id istnieje odpowiadająca właściwość, dodatkowa właściwość obsługuje informacje o lokalizacji karawany, wywołując metodę *lokalizuj(string id)* z klasy **matma.przekaznik**. Ponadto klasa przechowuje także listę obiektów typu string, zawierającą id wszystkich karawan. Rozbudowany konstruktor ustawia wartość zmiennej *idk*, przechowującą id karawany, na “KA01”, wywołuje metodę *zassaj()*, inicjalizuje wszystkie elementy, po czym usuwa zawartość listy karawan (tej w oknie) i wpisuje do niej zawartość listy karwan z klasy, przerabiając każdą wartość, by zamiast id prezentować tylko jej numer. Na zakończenie wywołuje metodę *CzasPodrozy(string id)* z klasy **matma.przekaznik** i sprawdza jej wynik - jeśli wynosi zero, czyni przyciski shop, warsztat, karczma i podroz klikalnymi, jeśli nie - nieklikalnymi.

```
int jazda = przekaznik.CzasPodrozy(idk);
if (jazda == 0)
{
    shop.IsEnabled = true;
    podroz.IsEnabled = true;
    warsztat.IsEnabled = true;
    karczma.IsEnabled = true;
}
else
{
    shop.IsEnabled = false;
    podroz.IsEnabled = false;
    warsztat.IsEnabled = false;
    karczma.IsEnabled = false;
}
```

Rysunek 4: Fragment kodu odpowiedzialny za ustawienie klikalności przycisków w zależności od tego czy karawana jest w podróży czy też nie. [opracowanie własne]

Klasa zawiera także 5 metod aktywowanych przez gracza. Pierwsza z nich nazywa się *ExitW_Click(kliknięcie)* i powoduje zamknięcie okna. Druga, *Shop_Click(kliknięcie)*, tworzy i wywołuje okno handlu, przekazując zmienną id jako argument. *Info_Click(kliknięcie)* działa podobnie, tylko zamiast okna handlu wywołuje okno informaji o mieście. Jako że okno to jako argument przyjmuje id miasta, zaś klasa **WaggonShop** przechowuje tylko id karawany, przed utworzeniem okna wywoływana jest metoda *lok(string id)* z klasy **matma.przekaznik**. Funkcja Podroz_Click(*kliknięcie*) w pierwszej kolejności wywołuje metodę czyMoznaPodrozowac(string id) z wymienionej wyżej klasy. Jeśli odda ona wartość true, tworzone i wywoływane jest okno podróży, jeśli false - okno komunikatu błędu, prezentujące wiadomość "W twojej karawanie jest za dużo wozów a za mało ludzi by wyruszyć w podróż.". Ostatnia jest metoda *Workshop_Click(kliknięcie)*, która tworzy i wywołuje okno warsztatu.

Poza reagowaniem na wciśnięcie przycisku, klasa reaguje też na wybór karawany z listy. Funkcjonalność tę realizuje metoda *listunia_SelectionChanged(object sender, SelectionChangedEventArgs e)*, która pobiera wybrany element, przerabia go aby był Id karawany i wpisuje w zmienną id klasy, a następnie wywołuje metodę *odswiez()*.

```
1 odwołanie | wardasz, 7 dni temu | 1 autor, liczba zmian: 2
private void listunia_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    string txt = listunia.SelectedItem.ToString();
    txt = txt.Remove(0, 13);
    txt = "KA" + txt;
    idk = txt;
    odswiez();
}
```

Rysunek 5: Metoda listunia_SelectionChanged(object sender, SelectionChangedEventArgs e), wywoływana wyborem karawany z listy. [opracowanie własne]

Ponadto klasa zawiera dwie zwykłe metody, *zassaj()* i *odswiez()*. Pierwsza wywołując metody *dajKarawany(),* I*leTowaru(string idk, string idt)*, *PoliczObciozenie(string id)* i *PoliczPojemnosc(string id)* z klasy **matma.przekaznik** pobiera informacje dotyczące ilości towarów w karawanie i jej pojemności i przekazuje je do okna. Druga nie wywołuje pierwszej z tych metod, za to na zakończenie wywołuje metodę *CzasPodrozy(idk)* i na podstawie jej wyniku czyni przyciski klikalnymi lub nie, podobnie jak czyni to konstruktor.

Klasa **Warsztat** obsługuje okno warsztatu. Zawiera ona sześć zmiennych typu string: *id*, *wozy*, *pojemnosc*, *ludzie*, *ochrona* i *pomoc*. Wszystkie poza pierwszą mają także odpowiadające im właściwości, które przekazują ich wartości do okna. Konstruktor przyjmuje wartość id karawany, inicjalizuje okno i wszystkie jego elementy, po czym wywołuje funkcję *zassaj()*. Ona sama wywołując metody *dajWozy(string id)*, *PoliczPojemnosc(string id)*, *dajOchrone(string id)* i *dajPomagierow(string id)* (wszystkie z klasy **matma.przekaznik**) pobiera informacje o karawanie, inne zaś wylicza, i przekazuje do okna. Jako że okno zawiera siedem przycisków, tyle też metod wywoływanych przez użytkownika znajduje się w klasie.

Najprostsza metoda *ExitWarsztat_Click(kliknięcie)* powoduje zamknięcie okna. Kolejne są *button3_Click(kliknięcie)* i *button7_Click(kliknięcie)*. Pierwsza wywołuje metodę *najmijPomoc(string id)*, druga *najmijOchrone(string id)*, (obie z klasy **matma.warsztat**). Obie na zakończenie wywołują metodę *zassaj()*.

Dalej są metody *button6_Click(kliknięcie)* i *button8_Click(kliknięcie)*. Pierwsza wywołuje metodę *zwolnijPomoc(string id)*, druga *zwolnijOchrone(string id)* (ponownie obie z klasy **matma.warsztat**). Następnie sprawdzają zwróconą wartość - jeśli wynosiła ona false, tworzą i wywołują okno komunikatu błędu, wyświetlając wiadomość "Nie masz pomocników do zwolnienia" lub "Nie masz najemników do zwolnienia". Na zakończenie wywołują metodę *zassaj()*.

Metoda *button_Click(kliknięcie)* wywołuje funkcję *kupWoz(id)* z klasy **matma.warsztat**. Jeśli zwróci ona wartość false, tworzą i wywołują okno komunikatu błędu, wyświetlając wiadomość "Nie stać cię na zakup wozu". Następnie wywołują metodę *zassaj()* oraz *odzwierzGlowne()* z klasy **MainWindow**.

Ostatnia metoda *nowa(kliknięcie)* wywołuje funkcję *nowaKarawana(id)* z klasy **matma.warsztat**. Jeśli zwróci ona wartość true, zamyka okno i wywołuje metody *odzwierzGlowne()* z klasy **MainWindow** oraz *z1()* z **GamesWindow**. Jeśli zaś zwróconą wartością będzie false, utworzone zostanie i wywołane okno komunikatu błędu z wiadomością “Nie stać cię na nową karawanę”.

Klasa **Zakupy** obsługuje okno handlu. Zawiera ona trzy zmienne typu string - *idk*, *idm* i *kasa*, obiekt klasy **matma.cena** i serię zmiennych typu string opisujące wszystkie dane każdego z towarów - ilość w mieście, ilość w karawanie, cenę kupna i cenę sprzedaży. Dla każdej z nich istnieje odpowiadająca zmiennej właściwość,

podobna skonstruowana jest także dla zmiennej kasa - łącznie daje to 45 właściwości, przekazujących zmienne do okna. Konstruktor jest zadziwiająco prosty, bowiem przyjmuje on jedną zmienną typu string, wpisując ją do zmiennej idk, inicjalizuje okno i wywołuje metodę *zassaj()*. Wypełnia ona zmienną *idm*, wywołując metodę *lok(string id)* z klasy matma.przekaznik. Wynik funkcji *DajKase()* z tej samej klasy wpisany zostaje jako wartość zmiennej *kasa*. Następnie utworzony zostaje obiekt klasy **matma.ceny**, do którego konstruktora przekazano wartość zmiennej *idm*. Po tym następuje wypełnianie danymi zmiennych opisujących towary, z użyciem "geterów" klasy **matma.ceny** oraz funkcji *IleTowaru(string idk, string idt)*. Jeśli cena kupna wynosi -1, wartość zmiennej zostaje zmieniona na "niemożliwe". Na koniec funkcja przekazuje wszystkie wartości do okna.

```
winoIK = przekaznik.IleTowaru(idk, "TO09");
winoIM = dane.getIle("TO09");
winoCK = dane.getCenaKup("TO09");
if (winoCK == "-1") { winoCK = "niemożliwe"; }
winoCS = dane.getCenaSp("TO09");
```

Rysunek 6: Fragment metody zassaj(), wypełniający zmienne przechowujące informacje o winie. [opracowanie własne]

Metoda *sprzedaz(string IDkarawana, string IDmiasto, string IDtowar, int ile, int cena)* wywołuje metodę *sprzedaj(string IDkarawana, string IDmiasto, string IDtowar, int ile, int cena)* z klasy **matma.handel**, wprost przekazując jej wszystkie przyjęte wartości. Jeśli zwrócona zmienna będzie miała wartość "done" wywołana zostanie metoda *zassaj()* oraz *odzwierzGlowne()* z klasy **MainWindow**. W każdym innym przypadku wyświetlone zostanie okno komunikatu błędu, w którym zostanie wyświetlony zwrócony napis.

Podobnie działa funkcja *kupowanie(string IDkarawana, string IDmiasto, string IDtowar, int ile, int cena)*, z tą tylko różnicą, że wywołuje ona metodę *kup(string IDkarawana, string IDmiasto, string IDtowar, int ile, int cena)*, ponownie prost przekazując jej wszystkie przyjęte wartości. Poza tym dalsza część działania jest identyczna.

Okno handlu ma czterdzieści pięć przycisków, a więc i czterdzieści pięć metod je obsługujących. Jedna z nich, exitZ_Click(*kliknięcie*), powoduje zamknięcie okna i wywołanie funkcji odswiezKarawane() z klasy GamesWindow. Pozostałe pogrupowane są w czwórki, po jednej dla każdego z jedenastu towarów. Pierwszą z czterech metod jest opisana w komentach znakiem "-". Pobiera ona wartość z pola

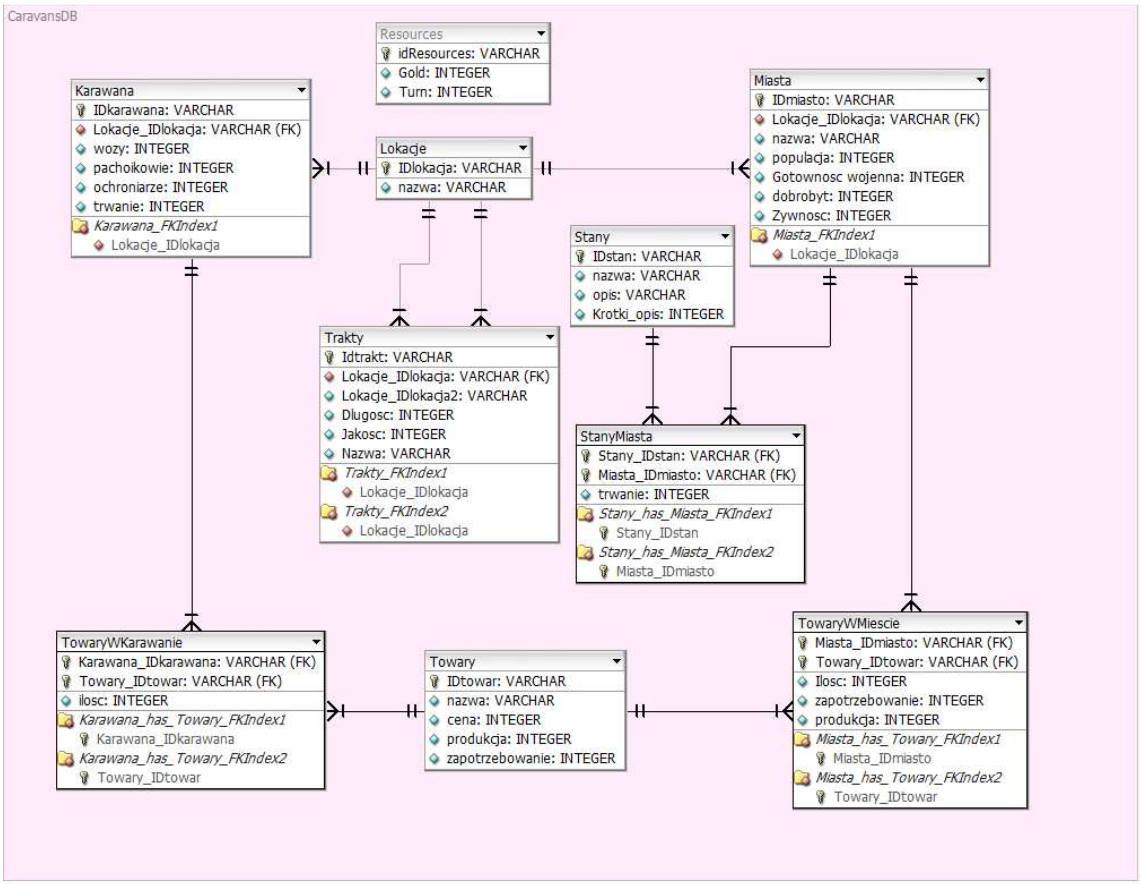
tekstowego odpowiadającego danemu towarowi, sprawdza czy jest ona większa od zera, jeśli jest - zmniejsza o jeden, po czym zwraca do pola tekstowego. Podobnie działają metody oznaczone znakiem "+", z tą różnicą że zamiast zmniejszać, zwiększają wartość. Jako że gracz może naraz kupić sprzedać nieograniczenie dużą ilość towaru, nie ma tutaj wstępnej kontroli wartości. Następny jest przycisk oznaczony słowem "sprzedaj". Pobiera ona wartość z odpowiedniego pola tekstowego, wyświetloną w oknie cenę, po czym wywołuje metodę *sprzedaz(string IDkarawana, string IDmiasto, string IDtowar, int ile, int cena)*. Id towaru wpisane jest w wywołanie ręcznie, reszta danych to zmienne pobrane przed chwilą lub zasadnicze zmienne klasy. Ostatnie są funkcje oznaczona słowem "kup". Zachowują się podobnie, z dwiema różnicami. Po pierwsze, jeśli pobrana z okna "cena" wynosi "niemożliwe", wartość ta jest zmieniana na "-1". Po drugie, wywoływaną funckją jest metoda *kupowanie(string IDkarawana, string IDmiasto, string IDtowar, int ile, int cena)*.

```
1 odwołanie | wardasz, 48 dni temu | 1 autor, liczba zmian: 6
private void buttonW1_Click(object sender, RoutedEventArgs e)//wino sprzedaj
{
    int ile = Convert.ToInt32(wino.Text);
    int cena = Convert.ToInt32(winoCS);
    sprzedaz(idk, idm, "TO09", ile, cena);
}

1 odwołanie | wardasz, 131 dni temu | 1 autor, 1 zmiana
private void buttonW2_Click(object sender, RoutedEventArgs e)//wino -
{
    int x;
    x = Convert.ToInt32(wino.Text);
    if (x > 0) { x--; }
    wino.Text = x.ToString();
}

1 odwołanie | wardasz, 48 dni temu | 1 autor, liczba zmian: 8
private void buttonW3_Click(object sender, RoutedEventArgs e)//wino kup
{
    int ile = Convert.ToInt32(wino.Text);
    int cena;
    if (winoCK == "niemożliwe")
    {
        cena = -1;
    }
    else
    {
        cena = Convert.ToInt32(winoCK);
    }
    kupowanie(idk, idm, "TO09", ile, cena);
}

1 odwołanie | wardasz, 131 dni temu | 1 autor, 1 zmiana
private void buttonW4_Click(object sender, RoutedEventArgs e)//wino +
{
    int x;
    x = Convert.ToInt32(wino.Text);
    x++;
    wino.Text = x.ToString();
}
```

Rysunek 7: Cztery metody obsługujące przyciski handlu winem. [opracowanie własne]

2.4 Diagram ERD

Diagram przedstawia w formie graficznej strukturę bazy danych, zależności między encjami i zawarte w nich atrybuty. Został stworzony za pomocą programu DBDesigner.



Rysunek 8: Diagram związków encji (diagram ERD). [opracowanie własne]

2.5 Projekt GUI

Zasadniczym celem przy tworzeniu interfejsu graficznego było uczynienie go przejrzystym i zrozumiałym dla nowych graczy. Bardzo wymowne napisy na przyciskach i ich wygodna lokalizacja pomagają w szybko przyswoić zasady gry osobie, która pierwszy raz uruchomi “Kupcy z Reganii”.

Pierwszym oknem, które otwiera się po uruchomieniu “Kupców z Reganii” jest okno “MainWindow”, pospolicie nazywane menu głównym. Widzimy w nim cztery przyciski: nowa gra, wczytaj grę, autorzy i wyjście.

Rysunek 9: Okno "MainWindow.xaml". [opracowanie własne]

"Nowa gra" - przycisk powoduje otwarcie okna "GameWindow.xaml" wypełniając w tle modele domyślnymi wartościami startowymi.

"Wczytaj grę"- przycisk powoduje otwarcie okna "GameWindow.xaml" wypełniając w tle modele danymi pobranymi z bazy danych.

"Wyjście" - przycisk powoduje wyłączenie gry.

"Autorzy" - przycisk powoduje otwarcie okna "Authors.xaml", zawierającego informacje o twórcach gry

Rysunek 10: Okno "Autorzy.xaml" na tle okna "MainWindow.xaml". [opracowanie własne]

Zasadniczym oknem, otwieranym po rozpoczęciu gry, jest Okno “GameWindow.xaml”, nazywane także głównym oknem gry. Jego centralną część zajmuje mapa świata gry, wokoło niej znajdują się liczne przyciski i pola tekstowe.

Rysunek 11: Okno “GamesWindow.xaml”. [opracowanie własne]

Button “Save” - przycisk w kształcie dyskietki, powoduje zastąpienie starych danych w bazie danych aktualnymi wartościami z modeli.

Button “Bmenu” - przycisk **“Zamknij”**, powoduje zamknięcie okna i otwarcie menu głównego

Button “Bend” - przycisk **“Oddaj turę”,** powoduje zakończenie tury gry i rozpoczęcie nowej

Pole tekstowe “Klepsydra” - w tym polu jest wypisany jest numer aktualnej tury

Pole tekstowe “textBlock” - w polu jest wypisany aktualny stan majątkowy gracza

Button “Bwaggon” - przycisk **“Karawana”** powoduje otwarcie okna “WaggonShop.xaml”.

Okno **“WaggonShop.xaml”**, zwane oknem przeglądu karawan, jest centralnym fragmentem programu, łączącym wszystkie kluczowe elementy. Pozwala przejrzeć zawartość wybranej karawany, a jeśli ta jest w mieście, przejść do okien pozwalających na podjęcie konkretnych akcji.

Rysunek 12: Okno "WaggonShop.xaml" (na liście znajduje się tylko jedna karawana, jako że gracz w tym momencie posiadał tylko jedną). [opracowanie własne]

ListBox "listunia" - Zawiera listę posiadanych karawan. Kliknięcie w jedną z pozycji w tej liście powoduje wyświetlenie danych o danej karawanie.

TextBlock " textBlock" i "textBlock1" - Ukazują aktualne i maksymalne obciążenie karawany.

ListBox "listBox" - Ukazuje ile jakich towarów przewozi karawana

Przycisk "Targowisko" - powoduje otwarcie okna "Zakupy.xaml"

Przycisk "Warsztat" - powoduje otwarcie okna "Warsztat.xaml"

Przycisk "Podróż" - powoduje otwarcie okna "Podroz.xaml"

Przycisk "Karczma" - powoduje otwarcie okna "MiastoTour.xaml"

Przycisk "Zamknij" - powoduje zamknięcie okna

Okno **"Zakupy.xaml",** zwane oknem handlu umożliwia dokonywanie transakcji w wybranym mieście. Zdecydowana jego większość to zestawy przycisków i pól tekstowych prezentujących informacje o każdym z towarów

Rysunek 13: Okno “Zakupy.xaml” na tle okna “GamesWindow.xaml”. [opracowanie własne]

Po lewej stronie podane są informacje o produktach, które znajdują się w karawanie, po prawej - w mieście. Przyciski oznaczone jako “+” i ”-” modyfikują wartość w polu pomiędzy, zaś przyciski “Kupuj” i “Sprzedaj” powodują kupno lub sprzedaż podanej w polu tekstowym ilości danego towaru. Dodatkowy przycisk “Zamknij” powoduje zamknięcie okna.

Okno **“Podroz.xaml”,** nazywane po prostu oknem podróży, umożliwia wyruszenie w podróż do innego miasta.

rysunek 14: Okno "podroz.xaml" na tle okna, "GamesWindow.xaml". Po bokach pierwszego widać skraje okna WaggonShop.xaml". [opracowanie własne]

Okno zawiera tuzin przycisków, podpisanych nazwami poszczególnych miast. Wciśnięcie jednego z nich rozpoczyna podróż do miasta którego nazwa wypisana jest na przycisku. Kolejny, trzynasty przycisk, oznaczony napisem "Zamknij" powoduje zamknięcie okna.

Okno **"MiastoTour.xaml"**, zwane oknem informacyjnym lub oknem przeglądu miasta, zawiera liczne informacje o danym mieście.

Rysunek 15: Okno "MiastoTour.xaml" na tle okna, "GamesWindow.xaml". [opracowanie własne]

Z lewej strony okna, nad obrazkiem, wyświetlona jest liczba populacji miasta. Po prawej stronie wyświetlone są opisowe informacje o stanie gospodarczym i społecznym miasta i ilości towarów w nim

Okno **"Warsztat.xaml"** zwane po prostu oknem warsztatu umożliwia powiększenie karawany lub stworzenie nowej.

Rysunek 16: Okno "Warsztat.xaml" - obecnie wyświetlane są dane dla karawany o 1 wozie, zatrudniająca czterech zbrojnych najemników. [opracowanie własne]

Interaktywna jest lewa połowa okna. Na szczycie wyświetlone są informacje o ilości wozów i pachołków w karawanie oraz jej pojemności. Niżej jest przycisk pozwalający kupić nowy wóz, dalej zaś zestaw przycisków pozwalających regulować ilość zatrudnianych pomocników i robotników. Na dole umieszczono przycisk umożliwiający stworzenie nowej karawany.

3. Implementacja

3.1 Architektura programu

Program stworzony jest z użyciem modelu MVC. Całość podzielony jest na trzy części, zawarte z trzech osobnych namespace: Caravans, Caravans.model i Caravans.matma.

Namespace Caravans zawiera klasy dziedziczące po klasie systemowej Window, generujące kolejne okna programu. Prezentują one wybrane dane, przekazywane przez obiekty i metody zawarte w namespace Caravans.matma, realizując zadanie widoków.

Namespace Caravans.matma zawiera klasy odpowiedzialne za przekazywanie i przetwarzanie danych. Klasy te stanowią kontrolery projektu.

Namespace Caravans.model zawiera klasy generujące obiekty będące odzwierciedleniem bazy danych w projekcie oraz odpowiedzialne za transfer danych pomiędzy tymi obiektami a samą bazą danych. Klasy te stanowią modele naszej aplikacji.

3.2 Użyte technologie obiektowe

Zdecydowana większość aplikacji wykonane zostało w języku obiektowym C#, z wykorzystaniem platformy .NET Framework oraz framework'a ADO.NET. Wybór języka był kwestią jego powierzchownej, ale niezbyt głębokiej znajomości - znajomość podstaw umożliwiła natychmiastowy start pracy, zaś nieznajomość szczegółów i zaawansowanych metod sprawiła, że pisanie programu było jednocześnie nauką nowych rozwiązań. Dodatkowym czynnikiem był także wygodny edytor GUI wmontowany w środowisko Visual Studio, z którego wszyscy korzystaliśmy, który znacznie ułatwił pracę nad tym fragmentem gry.

.NET Framework jest platformą programistyczną stanowiącą podstawę tworzenia oprogramowania w języku C# z wykorzystaniem środowiska Visual Studio. Z tego powodu nie było faktycznego wybierania tej technologii-wybór języka C# z automatu oznaczał wybór tej technologii.

Framework ADO.NET jest zbiorem bibliotek umożliwiających wygodne i stosunkowo łatwe stworzenie połączenia pomiędzy bazą danych a zasadniczą częścią aplikacji. Wybór tej technologii był bardzo oczywisty, do tego stopnia że ciężko go wręcz nazwać wyborem - nie dość, że jest to element wbudowany w platformę .NET

Framework i domyślna metoda tworzenia połączenia z bazą danych w języku C#, to jeszcze było to technologia znana nam z wcześniejszych zajęć na uczelni.

3.3 Użyte technologie bazodanowe

Jako system zarządzania bazą danych został wybrany Microsoft SQL Server, korzystający z języka Transact-SQL. Wybraliśmy tę technologię z kilku powodów: dostępność wygodnych narzędzi do pracy z tą bazą w Visual Studio, znajomość tego systemu oraz języka Transact-SQL, wynikła z wcześniejszego obcowania z nim na zajęciach uniwersyteckich, oraz z powodu wydajności oprogramowania tego systemu.

Baza Danych jest przedstawiona plikiem "CaravansDB.mdf" - plik lokalnej bazy danych Microsoft SQL Server .Plik "App.config" zawiera dane niezbędne do połączeniem z bazą danych.

3.4 Użyte narzędzia projektowania GUI

Do tworzenia GUI był wykorzystany edytor wbudowany w Visual Studio 2017 bazujący na języku XAML. Pliki App.xaml, jak i App.xaml.cs zawierają tę samą definicję klasy App. Są one kompilowane w jeden plik aplikacji App.g.cs, który można znaleźć po skompilowaniu aplikacji w katalogu projektu obj / Release. W pliku App.xaml znajduje się kod jest pisany w języku XAML, odpowiedzialny za rozmieszczenie poszczególnych elementów okna (przyciski, labelki, obrazy itp), plik App.xaml.cs zawiera metody pisane w języku C# odpowiedzialne za pobieranie wyświetlanych danych, i obsługujące wszelkie akcje podejmowane przez gracza.

4. Testowanie

4.1 Wstępne testowanie na atrapach

Jako że wiele fragmentów programu powstawało samodzielnie, w osobnych projektach, bazując na tworzonych na bieżąco atrapach innych segmentów programu, wiele testów odbywało się właśnie w tych osobnych projektach. Były to testy manualne, przeprowadzane bez ustalonego scenariusza, polegające na wielokrotnym uruchamianiu metod z wpisywanymi w kodzie danymi i sprawdzaniu czy rezultaty są zgodne z oczekiwaniami. Ze względu na charakter testów i całokształtu ówczesnych artapowych projektów, niemożliwym jest teraz odtworzenie czy szczegółowe opisanie tych testów. Skutkowały one jednak relatywnie wysoką poprawnością poszczególnych metod w momencie ich implementacji do zasadniczego projektu.

4.2 Testowanie funkcjonalności

Głównym segmentem testowania było manualne testowanie poszczególnych aspektów gry. Zawierały się w tym zarówno podstawowe, standardowe czynności i akcje, jak i te niestandardowe lub mające zakończyć się niepowodzeniem. W tym drugim przypadku niejednokrotnie poprzez kilka wcześniejszych akcji doprowadzaliśmy do konkretnego stanu gry, np. zużywaliśmy nadmiarowe złoto, by jego poziom był za niski do wykonania danej transakcji.

Lista testowanych przypadków:

-TestKupno1:

Scenariusz: próba kupna dowolnego towaru w dowolnym mieście;

Oczekiwanie: zmiany wyświetlanych ilości towaru w karawanie (zarówno w oknie handlu, jak i w przeglądzie karawan) i mieście, posiadanej gotówki (zarówno w oknie handlu, jak i w głównym oknie gry), obciążenie karawany i ewentualnych, choć nie wymaganych zmian cen;

Rezultat: wszystkie dane zmieniły się zgodnie z oczekiwaniami;

-TestKupno2:

Scenariusz: próba kupna towaru nieobecnego w danym mieście;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością, niezmienność wszelkich wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestKupno3:

Scenarusz: próba kupna towaru niedostępnego w danym mieście;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością, niezmienność wszelkich wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestKupno4:

Scenariusz: próba kupna większej ilości towaru niż jest w danym mieście;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością, niezmienność wszelkich wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestKupno5:

Scenariusz: próba kupna towaru o większej wartości niż posiadana gotówka;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością, niezmienność wszelkich wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestKupno6:

Scenariusz: próba kupna większej ilości towaru niż ilość wolnego miejsca w karawanie;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością, niezmienność wszelkich wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestSprzedaż1:

Scenariusz: próba sprzedaży dowolnego towaru w dowolnym mieście;

Oczekiwanie: zmiany wyświetlanych ilości towaru w karawanie (zarówno w oknie handlu, jak i w przeglądzie karawan) i mieście, posiadanej gotówki (zarówno w oknie handlu, jak i w głównym oknie gry), obciążenie karawany i ewentualnych, choć nie wymaganych zmian cen;

Rezultat: wszystkie dane zmieniły się zgodnie z oczekiwaniami;

-TestSprzedaż2:

Scenariusz: próba sprzedaży towaru nieposiadanego w karawanie;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością, niezmienność wszelkich wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestSprzedaż3:

Scenariusz: próba sprzedaży większej ilości towaru niż posiadamy w karawanie;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością, niezmienność wszelkich wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestPodróż1:

Scenariusz: próba wysłania karawany w podróż do dowolnego miasta;

Oczekiwanie: zamknięcie okien podróży i przeglądu karawan, zmiana lokacji karawany wyświetlanej w tym drugim oknie, zablokowanie miejskich przycisków w oknie karawan;

Rezultat: okna zamknęły się, dane zostały zmieniły się zgodnie z oczekiwaniem;

-TestPodróż2:

Scenariusz: próba rozpoczęcia podróży do miasta, w którym znajduje się karawana;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością, niezmienność wszelkich wyświetlanych danych;

Rezultat: nie doszło do zmian wyświetlanych danych, jednocześnie nie doszło do wyświetlenia komunikatu, okna zamknęły się jak w przypadku przedniego testu;

Dochodzenie i poprawki: szybko okazało się że nie została zaimplementowana instrukcja sprawdzająca czy przypadkiem wybrane miasto nie jest tym w którym znajduje się karawana. Po odkryciu tego braku instrukcja ta została dodana do programu;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestPodróż3:

Scenariusz: próba rozpoczęcia podróży bez wymaganej ilości członków karawany;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością na etapie próby wejścia do menu podróży;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestPracownicy1:

Scenariusz: próba zatrudnienia pracownika;

Oczekiwanie: wzrost ilości pracowników o jeden;

Rezultat: dana zmieniła się zgodnie z oczekiwaniami

-TestPracownicy2: :

Scenariusz: próba zwolnienia pracownika;

Oczekiwanie: spadek ilości pracowników o jeden;

Rezultat: dana zmieniła się zgodnie z oczekiwaniami;

-TestPracownicy3:

Scenariusz: próba zwolnienia pracownika przy zerowej ilości takowych;

Oczekiwanie: wyświetlenie komunikatu błędu z odpowiednią wiadomością, niezmienność wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestNajemnicy1:

Scenariusz: próba zatrudnienia najemnika;

Oczekiwanie: wzrost ilości najemników o jeden;

Rezultat: dana zmieniła się zgodnie z oczekiwaniami;

-TestNajemnicy2:

Scenariusz: próba zwolnienia najemnika;

Oczekiwanie: spadek ilości najemników o jeden;

Rezultat: dana zmieniła się zgodnie z oczekiwaniami;

-TestNajemnicy3:

Scenariusz: próba zwolnienia najemnika przy zerowej ilości takowych;

Oczekiwanie: wyświetlenia komunikatu błędu z odpowiednią wiadomością, niezmienność wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestWóz1:

Scenariusz: próba kupna wozu;

Oczekiwanie: spadek ilości złota o 200, zwiększenie ilości wozów, pojemności karawany i ilości pracowników potrzebnych do obsługi karawany;

Rezultat: wszystkie dane zmieniła się zgodnie z oczekiwaniami;

-TestWóz2:

Scenariusz: próba kupna wozu przy braku środków na taki wydatek

Oczekiwanie: wyświetlenia komunikatu błędu z odpowiednią wiadomością, niezmienność wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestKarawana1:

Scenariusz: próba sformowania karawany;

Oczekiwanie: spadek ilości złota o 500, zamknięcie okien warsztatu i przeglądu karawan i dojście nowej pozycji do listy karawan w oknie ich przeglądu;

Rezultat: okna się zamknęły, gotówka odjęła, karawana została dodana do listy. Jednocześnie obecność większej ilości karawan spowodowała problemy z wyświetlaniem informacji o nich;

Dochodzenie i poprawki: dogłębna analiza problemu wykazała usterki zarówno w metodzie odpowiedzialnej na odświeżanie ekranu przeglądu karawan (brak ponownego wczytania części danych i ustawienia klikalności przycisków) jak i odpowiedzialnej za tworzenie karawany (tworzono tylko obiekt klasy **model.TableCaravan**, nie wspierając go obiektami klasy **model.TableArtInTown**). Obie usterki zostały usunięte poprzez dodanie brakujących instrukcji.

Rezultat: okna się zamknęły, wyświetlane dane zmieniły zgodnie z oczekiwaniami

-TestKarawana2:

Scenariusz: próba sformowania karawany przy braku środków na taki wydatek;

Oczekiwanie: wyświetlenia komunikatu błędu z odpowiednią wiadomością, niezmienność wyświetlanych danych;

Rezultat: komunikat wyświetlono zgodnie z oczekiwaniem, dane nie zmieniły się;

-TestTura1:

Scenariusz: próba zakończenia tury gdy karawana stoi w mieście;

Oczekiwanie: spadek ilości złota, zwiększenie licznika tur o jeden;

Rezultat: wszystkie dane zmieniły się zgodnie z oczekiwaniami

-TestTura2:

Scenariusz: próba zakończenia tury, gdy karawana jest w trasie;

Oczekiwanie: spadek ilości złota, zwiększenia znacznika tury o jeden i zmiany czasu koniecznego na dotarcie do celu

Rezultat: wszystkie dane zmieniły się zgodnie z oczekiwaniami

-TestTura3:

Scenariusz: próba zakończenia tury, karawana nie ma najętych pracowników;

Oczekiwanie: zwiększenie licznika tury o jeden i zachowania poziomu gotówki;

Rezultat: wszystkie dane zmieniły się zgodnie z oczekiwaniami;

4.3 Testy integracyjne aplikacji

Kolejnym krokiem były końcowe testy integracyjne. Przeprowadzane były ręcznie, bez większego planu, w najprostszy możliwy sposób - poprzez długotrwałe użytkowanie aplikacji, starając się z jednej strony wykorzystać wszystkie możliwe

opcje na wszystkie możliwe sposoby, z drugiej obserwując powolne zmienianie się świata gry oszacować poprawność działania całokształtu matematycznego silnika gry.

Pierwszym, błyskawicznie wykrytym błędem jest zawieszanie się programu w momencie zakończenia tury bez wcześniejszego otwierania okna przeglądu karawan. Powodowane jest to wywołaniem funkcji *odswiez()* z klasy **WaggonShop**, które następuje w metodzie *odswiezKarawane()* w klasie **GamesWindow** - dochodzi do wywołania metody z obiektu który nie został jeszcze zainicjowany.

Problem został usunięty poprzez dodanie zmiennej logicznej będącej wskaźnikiem czy okno karawan zostało otwarte, i uzależnienie wywołania problematycznej metody od wartości tej zmiennej.

Drugim błędem było okazjonalne spadanie liczby mieszkańców miasta na wartości ujemne. Niezależnie od przyczyny zostało to poprawione poprzez dodanie instrukcji ustawiającej populację na 50 jeśli liczba spadnie poniżej tej wartości.

Dalej, na pograniczu błędu był fakt bardzo gwałtownych wahań liczby ludności. Pociągnęło to za sobą konieczność gruntownego przebadania metod odpowiedzialnych za zmiany tej wartości.

Najważniejszym wykrytym błędem był fregment metody *dzien()* z klasy **czas**, odpowiedzialny za obsługę stanu zarazy w czasie jej trwania. Pominięcie jednego warunku powodowało obniżenie populacji wszystkich miast, gdy w jednym z nich szalała zaraza. To, wraz z delikatnym skalibrowaniem metody *zmianaPopulacji()* usunęło problem.

4.4 Testy użytkownicze

Ostatnim krokiem były testy użytkownicze, czyli oddanie aplikacji w ręce osób niepowiązanych z projektem. Zasadniczym celem tych testów było sprawdzenie intuicyjności interfejsu i czytelności zasad gry, które nie są prezentowane graczowi w formie jakiegokolwiek samouczka i nie zostały przedstawione testerom.

Pierwsza testerka przy pierwszym kontakcie była wyjątkowo zagubiona, aczkolwiek kilka minut eksperymentów i jedna-dwie porady od jednego z twórców pozwoliły jej błyskawicznie wdrożyć się w grę. W momencie zrozumienia zasad gry szybko zaczęła dobrze się bawić, osiągając dość zdumiewające wyniki (były to pierwsze testy w czasie których gracz sprawdzał co i gdzie jest potrzebne - jak się okazało, nawet wiedza ograniczona do miasta w którym aktualnie przebywa karawana okazała się wielce dochodowa). Tym niemniej, testerka była zdania

że wysoce zalecane byłoby dodanie do gry samouczka, który wyeliminowałby pierwszy moment zagubienia. Ponadto zgłosiła kilka sugerowanych poprawek interfejsu, jak choćby przycisk szybkiej sprzedaży całości posiadanego towaru danego typu.

Przy okazji testowania przez drugą osobę udało się równocześnie sprawdzić działanie gry na innym komputerze, uruchamiając program z poziomu pliku .exe, a nie poprzez Visual Studio. Początkowo program działał bez zarzutu, potem jednak okazało się że próba odczytu gry powoduje przerwanie działania programu - wynika to najprawdopodobniej z braku zainstalowanego silnika bazy danych. Jednocześnie wyszły na jaw problemy z interfejsem na innym ekranie o innej rozdzielczości. Pierwszym wnioskiem drugiej testerki ponownie było "przydałby się samouczek", aczkolwiek ponownie w miarę szybko udało jej się rozgryźć zasady działania gry. Po dłuższym czasie stwierdziła, że bez samouczka da się obyć, tym niemniej miło by było aby pojawił się, choć w minimalistycznej formie. Testerka uznała interfejs nie tylko za prosty i czytelny, ale i graficznie ładny. Poza tym zgłosiła garść pomysłów i poprawek, od drobnych typu wyświetlenie czasu podróży przed jej rozpoczęciem lub możliwość nadania nazwy karawanie, po duże, typu sugestie by dodać możliwość tworzenia magazynów, czy wprowadzenie dziennika podróży.

5. Podział pracy

Projekt tworzony był przez czteroosobowy zespół, którego członkowie wymienieni byli na stronie tytułowej pracy. Każdy członek zespołu odpowiedzialny był za oddzielną sferę programu, co umożliwiało początkowo samodzielną pracę każdego z członków zespołu, której efekty zostały następnie zintegrowane ze sobą i złączone w jedną aplikację - po jej scaleniu nastąpiły testy i poprawki, jak i dodawanie ostatnich funkcjonalności.

Kiedrowicz Kamil

Do zadań tej osoby należało przede wszystkim zaimplementowanie oraz debugowanie klasy podroz odpowiedzialnej za poruszanie się karawan między miastami, która znajduje się w namespace Caravans.matma. Zaimplementowanie wymagało użycia algorytmu Dijstry do wyznaczania najmniejszej odległości pomiędzy wierzchołkami grafu reprezentowanymi na mapie jako miasta. Wykonanie tego zadania wymagało utworzenia klasy Tablicadoliczenia znajdującej się w namespace Caravans.matma, która była użyta do utworzenia tablicy wykorzystanej przy obliczaniu odległości. Potrzebne było także lepsze zobrazowanie gdzie dane miejscowości się znajdują i w jaki sposób są połączone, co z kolei wymagało poprawy jakości mapy oraz widoczności nazw danych lokacji.

Kolejnym zadaniem, jakim ta osoba wykonała jest zaprogramowanie i usuwanie błędów związanych z funkcjami obsługującymi kupno oraz sprzedaż towarów w klasie handel, która znajduje się w namespace Caravans.matma.

Kirsanau Aliaksei

Był odpowiedzialny za bazodanową sferę gry. Jest więc autorem przede wszystkim bazy danych, modeli i metod przesyłających dane pomiędzy jednym a drugim, a także wczytujących dane z pliku - słowem, wszystkie klasy w namespace Caravans.model są niemalże w całości jego autorstwa - wyjątkiem jest kilka metod w klasie Modele, np. dodajKarawane(string a, string b). Warto tutaj nadmienić, że zarówno baza danych, jak i modele, zostały napisane przez niego ręcznie.

Dodatkowo, jest też autorem fabularnej części gry. To on narysował mapę, w formie prostego szkicu, to on jest odpowiedzialny za nazwy miast, ich położenie i charakterystykę. Przekłada się to na stworzenie startowej treści tabel Lokacje i Trakty,

jak również Karawana i TowarWKarawanie. Jako dopełnienie jest on pomysłodawcą tytułu gry.

Rybak Mark

Jego głównym zadaniem było utworzenie graficznego interfejsu użytkownika gry. Wszystkie pliki .xaml znajdujące się w namespace Caravans są jego, niemalże wyłącznego autorstwa - wyjątkiem jest plik WaggonShop.xaml i znajdujące się w nim metody obsługujące listę karawan. Odpowiedzialny jest także za klasy częściowe znajdujące się w plikach xaml.cs, w tym samym namespace. W tym przypadku utworzył on klasy i napisał zdecydowaną większość ich kodu, aczkolwiek niemały wkład w ich tworzenie mieli także pozostali członkowie grupy.

Na potrzeby poszczególnych okien utworzył także kilka metod w klasie przekaznik w namespace Caravans.matma.

Wardziński Marcin

Jego zasadniczym polem działania był matematyczny mózg programu, algorytmy obliczające ceny i odpowiedzialne za szeroko pojętą symulację żyjącego świata. Zalicza się do tego produkcja i zużycie towarów, zmiany populacji, rozwój miast i zachodzące w nich losowe stany i wydarzenia.

Przełożyło się to na stworzenie większości klas z namespace Caravans.matma - ceny, czas, miasto i towar. Uczestniczył także w tworzeniu klasy częściowej Zakupy, implementując fragment pobierający dane o towarach, zwłaszcza tych zawartych w obiektach klasy ceny. Jego autorstwa jest też część metod w klasie przekaźnik w namespace Caravans.matma, przede wszystkich tych tworzących fabularne opisy miasta (np dajBogactwo(string id) czy dajStany(string id)).

Dodatkowym elementem, który stworzył w związku z tym, była startowa zawartość części tabel w bazie danych - przede wszystkim tabel Miasta, Towary, TowaryWMiescie i Stany

Osobnym zagadnieniem, za które jest odpowiedzialny, jest obsługa okna warsztatu, czyli klasa częściowa Warsztat z namespace Caravans i klasa warsztat z namespace Caravans.matma, a także metoda dodajKarawane(string a, string b) z klasy Modele w namespace Caravans.model. Pociągnęło to za sobą także lekką modyfikację okna przeglądu karawan, polegającą na dodaniu listy karawan, z której gracz wybiera którą karawanę chce w danym momencie obsługiwać. Dokonał jej własnoręcznie,

zarówno na poziomie GUI (plik WaggonShop.xaml) jak i kodu go obsługującego (klasa częściowa WaggonShop w namespace Caravans).

Jest on także odpowiedzialny za większość testów przeprowadzanych na scalonym programie, w tym testy integracyjne i użytkownicze, a także za poprawkę części wykrytych w ten sposób błędów.

6. Bibliografia

1. Dokumentacja języka C#: https://docs.microsoft.com/pl-pl/dotnet/csharp/ [dostęp 05.06.2018].

2. Dokumentacja języka T-SQL: https://docs.microsoft.com/en-gb/sql/t-sql/ [dostęp 05.06.2018].

3. Dokumentacja języka XAML: https://docs.microsoft.com/pl-pl/visualstudio/designers/designing-xaml-in-visual-studio [dostęp 05.06.2018].

4. Dokumentacja framework'a ADO.NET: https://docs.microsoft.com/pl-pl/dotnet/framework/data/adonet/ [dostęp 05.06.2018].

5. Slajdy dr. inż. M. Żabińskiej dotyczące wymagań funkcjonalnych i niefunkcjonalnych: http://www.ujk.edu.pl/ifiz/pl/files/lectures/Inzynieria_oprogramowania/UJK-IO-FazaOkrWym.pdf [dostęp 05.06.2018].

Oświadczenie

Wyrażamy zgodę / nie wyrażamy zgody* na udostępnienie osobom zainteresowanym naszej pracy dyplomowej dla celów naukowo-badawczych.
Zgoda na udostępnienie pracy dyplomowej nie oznacza wyrażenia zgody na kopiowanie pracy dyplomowej w całości lub w części.
* *niepotrzebne skreślić*

.....	.....
data	podpis
.....	.....
data	podpis
.....	.....
data	podpis
.....	.....
data	podpis