**Zarządzanie projektem informatycznym**

|  |  |  |  |
| --- | --- | --- | --- |
| **Kierunek studiów:** | *Informatyka* | **Rok studiów:** | *III* |
| **Numer grupy:** | *L5* | | |
| **Rok akademicki:** | *2014/2015* | **Semestr:** | *VI* |

**Sprawozdanie z wykonania projektu:**

**Projekt zespołowy systemu informatycznego**

|  |  |  |
| --- | --- | --- |
| **Lp.** | **Nr indeksu** | **Imię i nazwisko** |
| 1. | 86316 | Radosław Szymański |
| 2. | 84135 | Damian Szymański |
| 6. | 84139 | Kamil Ślusarczyk |
| 4. | 84123 | Artur Stelmach |

|  |  |  |
| --- | --- | --- |
| **Termin zajęć:** | | **Prowadzący:** |
| dzień: | *Wtorek* | mgr Przemysław Strzelczyk |
| godzina: | *17:15* |

# Założenia i cele programu

Głównym założeniem programu było stworzenie aplikacji umożliwiającej dokonywanie zakupów poprzez sieć. Charakterystyczne dla projektu było zrezygnowanie z interfejsu graficznego. Powodem tego była chęć skupienia się jedynie na abstrakcji systemu. Chcieliśmy wykonać grupowy projekt symulujący pracę w niewielkich firmach programistycznych, koncentrując się jedynie na tym co stanowi istotę programowania obiektowego.

# System kontroli wersji

System kontroli wersji to oprogramowanie służące do śledzenia zmian głównie w kodzie źródłowym oraz pomocy programistom w łączeniu zmian dokonanych w plikach przez wiele osób w różnych momentach czasowych. Nasza grupa, z uwagi na wcześniejsze doświadczenie, korzysta z systemu Git. Do obsługi tego systemu używany jest klient Source Tree.

Git - rozproszony system kontroli wersji. Stworzył go Linus Torvalds jako narzędzie wspomagające rozwój jądra Linux. Git stanowi wolne oprogramowanie i został opublikowany na licencji GNU GPL w wersji 2.s

# Użyte narzędzia

Z uwagi na wcześniejsze doświadczenia i zainteresowania aplikację postanowiliśmy napisać w środowisku .Net. Wybraliśmy platformę aplikacyjną do budowy aplikacji internetowych opartych na wzorcu Model-View-Controller (MVC) ASP.NET MVC. Jest to środowisko i model, który umożliwia szybką pracę, jednocześnie nie zamykający możliwości programiście.

Użyliśmy również Enity Framework, który jest narzędziem typu ORM (Object Relational Mapping), pozwalającym odwzorować relacyjną bazę danych za pomocą architektury obiektowej. Skorzystaliśmy również z wzorca Dependency Injection poprzez jego implementację - Ninject.

# Wymagania niefunkcjonalne:

• Funkcjonowanie w systemie Windows 7 i nowszych

• Umożliwienie obsługi sklepu użytkownikom niezwiązanym z informatyką

# Interfejs tekstowy

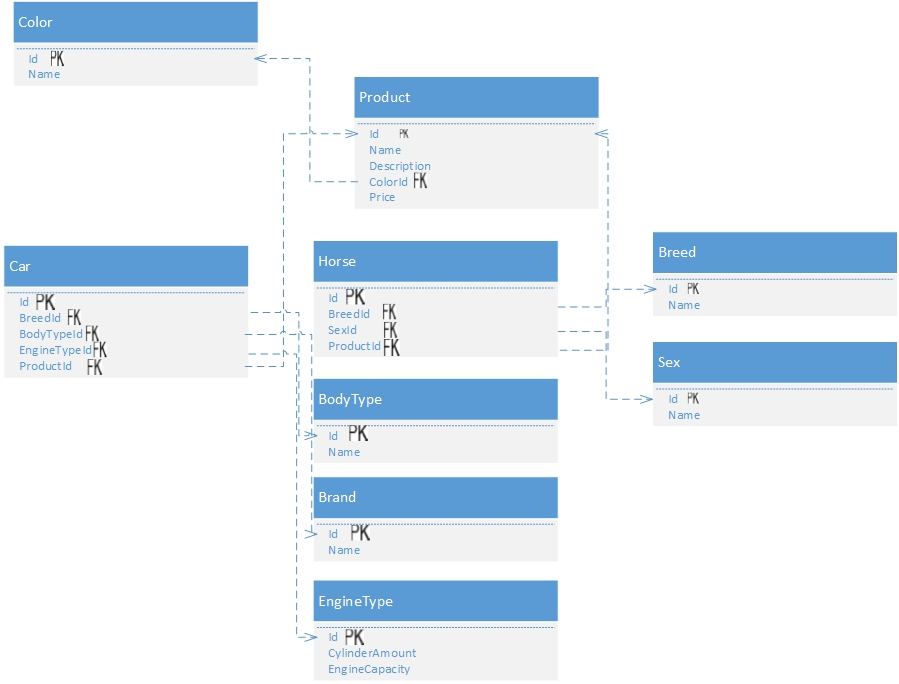
Jednym z głównych założeń projektu było zrezygnowanie z graficznego interfejsu użytkownika. Należało, więc stworzyć interfejs, który nie tylko pozwoli na sprawną nawigację w sklepie, ale także sprawi, że klienci będą tutaj wracać z uśmiechem na twarzy. W tym celu stworzony został interfejs tekstowy, którego uzupełnieniem są komunikaty głosowe.

Po określeniu wymagań funkcjonalnych stworzono bazę komend. Nawigacja odbywa się poprzez wpisywanie ich w aplikacji klienckiej. Dla użytkowników niezaznajomionych z aplikacją przygotowano pomoc, w której znajdują się wszystkie polecenia wraz z ich opisem oraz przykładem użycia.

Świadomym krokiem było nie stworzenie funkcjonalności poprawiania składni. Spowodowane to było nie tylko brakiem czasu i środków, ale przede wszystkim chęcią nakłonienia klientów do skrupulatnego i poprawnego wpisywania komend, co z całą pewnością przełoży się na sukcesy w życiu zawodowym i prywatnym, dzięki czemu zadowolenie z aplikacji sklepowej wzrośnie jeszcze bardziej.

Brak graficznego interfejsu nie oznacza istnienia jedynie teksu. Po połączeniu z serwerem wyświetlane zostaje niezwykłe logo sklepu, Koń i Samochód wykonane w technologii ASCII ART. Raz ujrzane nie znika już z pamięci podobnie jak piersi 20 letniej, nagiej dziewczyny, która lipcowym porankiem przechadza się po zroszonej łące a obok której idzie o dwa lata młodsza siostra, jednakowoż naga a której wzrok zdaje się mówić „Nas jest dwie a Ty jeden, ale to nic nie szkodzi”. Podobnie komunikaty głosowe sprawiają, że zakupy nie są jedynie monotonną czynnością a rytuałem, który z powodzeniem może zastąpić wizytę w filharmonii czy operze.

# Diagram bazy danych



Rys.1. Diagram bazy danych

# Funkcjonalności programu

Podczas projektu aplikacji założono, iż będzie ona obsługiwana z poziomu konsolowej linii poleceń. Założenie to miało kluczowy wpływ na sposób realizacji zadania - zarówno jeśli chodzi o przyjęty sposób interakcji z użytkownikiem jak i formę prezentacji treści przez program.

Wszystkie funkcjonalności są przetwarzane przez parser w zależności od wpisanego przez użytkownika tekstu. Komendy są zawarte w słowniku komend w klasie CommandParser. Więcej informacji na jej temat zawarto w punkcie 7.2.

Komunikaty wyświetlane przez program są pobierane z klasy CommunicatesKinds i zostaną opisane w punkcie 7.3.

Poniżej zaprezentowano funkcjonalności aplikacji z podziałem na dostępne z poziomu administratora oraz klienta. Przy opisie przyjęto założenie, że dana operacja powiodła się i nie opisano szczegółowo możliwych ścieżek postępowania na wypadek niepowodzenia. Należy jednak zaznaczyć, że zamieszczono kod odpowiadający zarówno za przeprowadzanie operacji jak i obsłużenie wyjątków.

# 6.1 Funkcjonalności programu dostępne dla wszystkich

**6.1.1. Logowanie**

Podstawowa funkcjonalność systemu, która umożliwia autoryzację i autentykację użytkownika w celu umożliwienia dalszych operacji.

|  |
| --- |
| if (loggedUser != null)  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.AlreadyLogged);  var login = parameters[0];  var password = parameters[1];  var user = userBaseRepository.GetUserByLogin(login);  if (user == null)  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.LoginNotFound);  if (user.Password != password)  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.IncorrectPassword);    loggedUser = user;  Play(global::System.Reflection.MethodBase.GetCurrentMethod().Name);  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.LoginAccepted); |

Listing.1. Kod: logowanie

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Sprawdzenie, czy użytkownik nie jest już zalogowany
2. Jeśli znaleziono użytkownika o określonych danych następuje pobranie go z bazy.
3. Odtworzenie odpowiedniego utworu muzycznego.
4. Komunikat o udanym logowaniu.

**6.1.2. Wylogowanie**

Podstawowa funkcjonalność systemu umożliwiająca zakończenie pracy z systemem.

|  |
| --- |
| if (CheckIfLogged(out checkCommunicate))  return checkCommunicate;  loggedUser = null;  Play(global::System.Reflection.MethodBase.GetCurrentMethod().Name);  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.LogoutAccepted); |

Listing.2. Kod: kończenie pracy

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Sprawdzenie, czy użytkownik jest zalogowany
2. Następuje przypisanie do zmiennej *loggedUser* wartości *null*.
3. Odtworzenie odpowiedniego utworu muzycznego.
4. Komunikat o udanym wylogowaniu.

**6.1.3. Wiadomość zwrotna o aktualnym stanie użytkownika**

Funkcjonalność ta pozwala na otrzymanie informacji o aktualnie zalogowanym użytkowniku.

|  |
| --- |
| (global::System.Reflection.MethodBase.GetCurrentMethod().Name);  return loggedUser != null  ? CommunicatesFactory.GetCommunicate(CommunicatesKinds.LoggedAs) + loggedUser.Name  : CommunicatesFactory.GetCommunicate(CommunicatesKinds.NotLogged); |

Listing.3. Kod: zalogowany użytkownik

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Odtworzenie odpowiedniego utworu muzycznego.
2. Wyświetlenie loginu oraz roli użytkownika, który wpisał komendę

**6.1.4. Czyszczenie konsoli**

Funkcjonalność umożliwiająca na wyczyszczenie okna poleceń

|  |
| --- |
| Console.Clear();  Play(global::System.Reflection.MethodBase.GetCurrentMethod().Name);  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.Cleaned); |

Listing.4. Kod: czyszczenie okna

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Odtworzenie odpowiedniego utworu muzycznego.
2. Wyczyszczenie konsoli.
3. Wyświetlenie odpowiedniego komunikatu.

**6.1.5. Pomoc**

Funkcjonalność ułatwiająca korzystanie ze sklepu internetowego. Po wywołaniu odpowiedniej komendy ukazuje się spis wszystkich dostępnych komend i parametrów, które te komendy mogą przyjmować. Widoczny jest również krótki opis komendy.

|  |
| --- |
| return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ShowHelp); |

Listing.5. Kod: pomoc

**6.1.6. Zatrzymanie odtwarzania muzyki**

Funkcjonalność umożliwiająca zatrzymanie odtwarzania utworów, przyporządkowanych do danej komendy. Z racji wysokich walorów artystycznych naszych nagrań komenda jest uważana za całkowicie zbędną i szkodliwą.

|  |
| --- |
| player.Stop();  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.MusicComunikatesStop); |

Listing.6. Kod: zatrzymanie muzyki

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Zatrzymanie odtwarzanego utworu muzycznego.
2. Wyświetlenie odpowiedniego komunikatu.

**6.1.7. Wyświetlenie produktów zdefiniowanego typu o określonej wartości wybranego parametru.**

|  |
| --- |
| try  {  var compareModel = FilterHelper.GetCompareModel(filtersDictionary);  var filtredCars = GetFiltredCars(compareModel);  if (filtredCars.Count() == 0)  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductNotFound);  var stringresult = filtredCars.Aggregate(CommunicatesFactory.GetCommunicate(CommunicatesKinds.FoundCars),  (current, car) => current + (car.ToString() + "\n"));  return stringresult;  }  catch (InvalidValueExeption ex)  {  return "C&H Shop > " + ex.Message;  } |

Listing.7. Kod: wyświetlanie produktów

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Generowanie modelu do porównania.
2. Filtrowanie rekordów na podstawie wygenerowanego modelu porównawczego.
3. W przypadku znalezienia rekordów, spełniających kryteria następuje ich wylistowanie.
4. W przypadku braku wyników następuje przechwycenie wyjątku.

**6.1.9. Zakończenie pracy programu**

Funkcjonalność umożliwia zakończenie pracy programu.

|  |
| --- |
| IsWorking = false;  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.Thanks); |

Listing.8. Kod: kończenie pracy programu

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Ustawienie zmiennej, odpowiadającej za pracę systemu na wartość *false.* Jest ona następnie obsłużona w instancji systemu w pliku
2. Wyświetlenie odpowiedniego komunikatu.

# 6.2 Funkcjonalności programu dostępne dla administratora

**6.2.1. Dodawanie produktu typu Koń**

Podstawowa funkcjonalność administratora, służąca do dodania produktu typu Koń do sklepu.

|  |
| --- |
| if (CheckIfLogged(out checkCommunicate)) return checkCommunicate;  if (!CheckIfNotAdmin(out checkCommunicate)) return checkCommunicate;  var horse = new Horse()  {  Name = parameters[0],  Description = parameters[1],  ColorId = Convert.ToInt32(parameters[2]),  Price = Convert.ToInt32(parameters[3]),  BreedId = Convert.ToInt32(parameters[4]),  SexId = Convert.ToInt32(parameters[5]),  };  if (ValidationHelper.isValid(horse))  {  dbContext.Products.Add(horse);  dbContext.SaveChanges();  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductAddedToShop);  }  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductAddedToShopFail); |

Listing.9. Kod: dodawanie produktu Koń

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Weryfikacja, czy osoba wywołująca komendę jest adminem
2. Tworzenie obiektu typu Koń na podstawie otrzymanych parametrów.
3. Walidacja obiektu oraz dodanie go do bazy danych
4. Wyświetlenie odpowiedniego komunikatu.

**6.2.2. Dodawanie produktu typu Samochód**

Podstawowa funkcjonalność administratora, służąca do dodania produktu typu Samochód do sklepu.

|  |
| --- |
| if (CheckIfLogged(out checkCommunicate)) return checkCommunicate;  if (!CheckIfNotAdmin(out checkCommunicate)) return checkCommunicate;  var car = new Car()  {  Name = parameters[0],  Description = parameters[1],  ColorId = Convert.ToInt32(parameters[2]),  Price = Convert.ToInt32(parameters[3]),  BrandId = Convert.ToInt32(parameters[4]),  BodyTypeId = Convert.ToInt32(parameters[5]),  EngineTypeId = Convert.ToInt32(parameters[6])  };  if (ValidationHelper.isValid(car))  {  dbContext.Products.Add(car);  dbContext.SaveChanges();  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductAddedToShop);  }  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductAddedToShopFail); |

Listing.10. Kod: dodawanie produktu Samochód

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Weryfikacja, czy osoba wywołująca komendę jest adminem
2. Tworzenie obiektu typu Samochód na podstawie otrzymanych parametrów.
3. Walidacja obiektu oraz dodanie go do bazy danych
4. Wyświetlenie odpowiedniego komunikatu.

**6.2.3. Usunięcie produktu**

Podstawowa funkcjonalność administratora, służąca do usuwania produktów ze sklepu.

|  |
| --- |
| if (CheckIfLogged(out checkCommunicate)) return checkCommunicate;  if (!CheckIfNotAdmin(out checkCommunicate)) return checkCommunicate;  int id;  if (Int32.TryParse(parameters[0], out id))  {  var product = dbContext.Products.Where(x => x.Id == id).FirstOrDefault();  if (product != null)  {  dbContext.Products.Remove(product);  dbContext.SaveChanges();  Play(global::System.Reflection.MethodBase.GetCurrentMethod().Name);  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductDeltedFromShop);  } |

Listing.11. Usuwanie produktu

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Weryfikacja, czy osoba wywołująca komendę jest adminem.
2. Próba wyszukania w bazie obiektu o zdefiniowanym id i usunięcie go.
3. Wyświetlenie odpowiedniego komunikatu.
4. Odtworzenie odpowiedniego utworu muzycznego.

# 6.3 Funkcjonalności programu dostępne dla klienta

**6.3.1. Dodawanie produktu do koszyka**

Podstawowa funkcjnalność kliencka, która umożliwa dodawanie produktów o zdefiniowanych parametrach do koszyka.

|  |
| --- |
| if (CheckIfLogged(out checkCommunicate)) return checkCommunicate;  if (CheckIfNotAdmin(out checkCommunicate)) return checkCommunicate;  List<int> intParams = new List<int>(parameters.Count);  for (int i = 0; i < parameters.Count; i++)  {  int temp;  if (Int32.TryParse(parameters[i], out temp))  {  intParams.Insert(i, temp);  }  else  {  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.IncorrectParameter) + i;  }  }  var id = intParams[0];  var ammount = intParams[1];  var user = (Client)loggedUser;  if (user.Cart == null)  {  user.Cart = new Cart();  }  var product = productRepository.FindBy(x => x.Id == id).FirstOrDefault();  if (product == null)  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductNotFound);  user.Cart.Products.AddRange(product, intParams[1]);  Play(global::System.Reflection.MethodBase.GetCurrentMethod().Name);  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductAddedToCart); |

Listing.12. Kod: dodawanie produktu do koszyka

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Weryfikacja, czy osoba wywołująca komendę jest klientem.
2. Weryfikacja, czy liczba parametrów wpisana przez użytkownika jest odpowiednia.
3. Jeśli jest to pierwszy obiekt wybrany przez użytkownika następuje utworzenie obiektu typu *Cart,* który jest odwzorowaniem koszyka.
4. Wyszukanie obiektu o zdefiniowanym przez użytkownika id i dodanie wybranej ilości tego typu produktów do koszyka.
5. Wyświetlenie odpowiedniego komunikatu.
6. Odtworzenie odpowiedniego utworu muzycznego.

**6.3.2. Usuwanie produktu z koszyka**

Podstawowa funkcjonalność kliencka, która umożliwia usuwanie produktów o zdefiniowanych parametrach z koszyka.

|  |
| --- |
| if (CheckIfLogged(out checkCommunicate)) return checkCommunicate;  if (CheckIfNotAdmin(out checkCommunicate)) return checkCommunicate;  List<int> intParams = new List<int>(parameters.Count);  for (int i = 0; i < parameters.Count; i++)  {  int temp;  if (Int32.TryParse(parameters[i], out temp))  {  intParams.Insert(i, temp);  }  else  {  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.IncorrectParameter) + i;  }  }  var id = intParams[0];  var ammount = intParams[1];  var user = (Client)loggedUser;  if (user.Cart == null || user.Cart.Products.Count() == 0)  {  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.EmptyCart);  }  if (user.Cart.Products.Exists(x => x.Id == id))  {  var product = user.Cart.Products.Find(x => x.Id == id);  for (int i = 0; i < ammount; i++)  {  if (product != null)  {  user.Cart.Products.Remove(product);  }  }  }  else  {  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductNotFound);  }  Play(global::System.Reflection.MethodBase.GetCurrentMethod().Name);  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.ProductRemovedFromCart); |

Listing.13. Kod: usuwanie produktu o zadanych parametrach z koszyka

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Weryfikacja, czy osoba wywołująca komendę jest klientem.
2. Weryfikacja, czy liczba parametrów wpisana przez użytkownika jest odpowiednia.
3. Sprawdzenie, czy koszyk nie jest pusty.
4. Sprawdzenie, czy koszyk zawiera obiekt o zdefiniowanym przez użytkownika id
5. Usunięcie wybranej ilości tego typu produktów z koszyka.
6. Wyświetlenie odpowiedniego komunikatu.
7. Odtworzenie odpowiedniego utworu muzycznego.

**6.3.3. Podgląd koszyka zakupów**

Funkcjonalność pozwalająca klientowi na podglądnięcie zawartości jego koszyka.

|  |
| --- |
| if (CheckIfLogged(out checkCommunicate)) return checkCommunicate;  if (CheckIfNotAdmin(out checkCommunicate)) return checkCommunicate;  var user = (Client)loggedUser;  return user.Cart == null ? CommunicatesFactory.GetCommunicate(CommunicatesKinds.EmptyCart) : user.Cart.ToString(); |

Listing.14. Kod: zawartość koszyka

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Weryfikacja, czy osoba wywołująca komendę jest klientem.
2. Zwrócenie zawartości koszyka z opisem bądź wyświetlenie informacji o pustym koszyku.

**6.3.4. Podsumowanie zakupów i złożenie zamówienia**

Funkcjonalność umożliwiająca zakończenie zakupów i złożenie zamówienia.

|  |
| --- |
| if (CheckIfLogged(out checkCommunicate)) return checkCommunicate;  if (CheckIfNotAdmin(out checkCommunicate)) return checkCommunicate;  var user = (Client)loggedUser;  if (user.Cart == null)  {  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.EmptyCart);  }  var order = new Order() { Client = user };  var productLists = new List<ProductList>();  foreach (var product in user.Cart.Products)  {  if (productLists.Exists(x => x.ProductId == product.Id))  {  var productList = productLists.Find(x => x.ProductId == product.Id);  productList.Amount++;  }  else  {  productLists.Add(new ProductList() { Amount = 1, Order = order, ProductId = product.Id });  }  }  dbContext.Orders.Add(order);  dbContext.ProductLists.AddRange(productLists);  dbContext.SaveChanges();  user.Cart = null;  Play(global::System.Reflection.MethodBase.GetCurrentMethod().Name);  return CommunicatesFactory.GetCommunicate(CommunicatesKinds.OrderIsInProgress); |

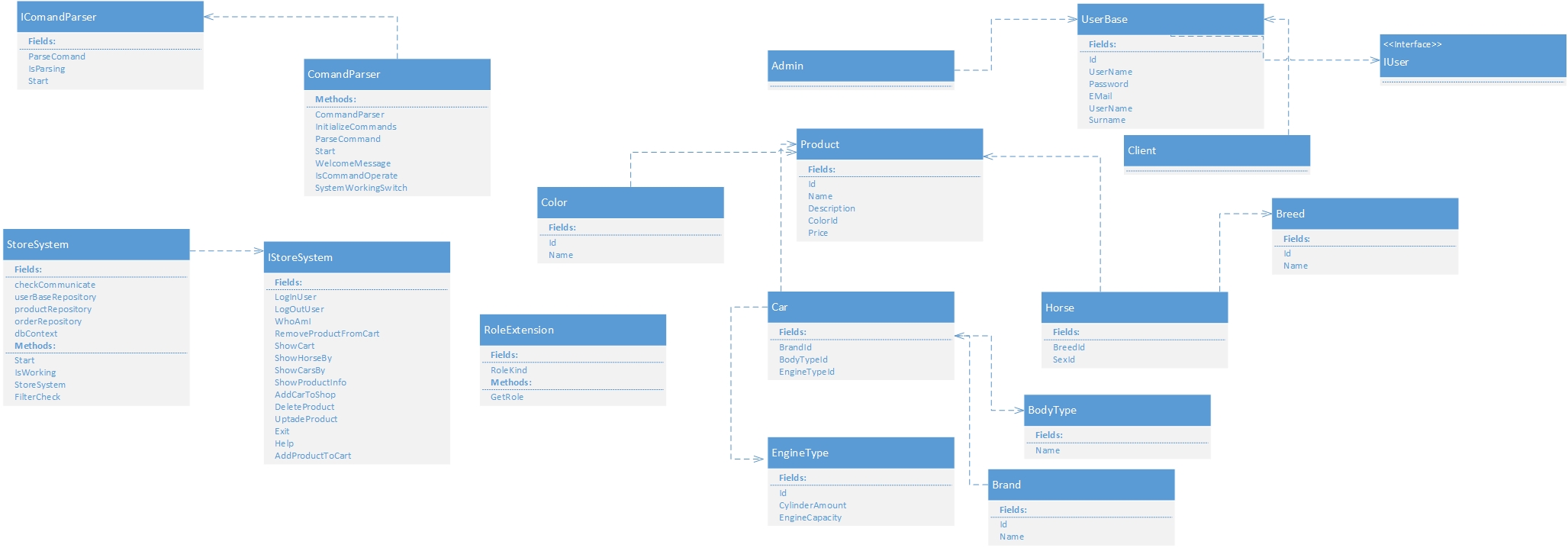
Listing.15. Kod: zakończenie zakupów i złożenie zamówienia

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Weryfikacja, czy osoba wywołująca komendę jest klientem.
2. Weryfikacja, czy koszyk nie jest pusty
3. Dodanie zamówienia do listy *ProductList*. Jest to wykonywane w celu umożliwienia śledzenia zamówień przez administratora.
4. Zapisanie w bazie zdefiniowanego zamówienia.
5. Wyświetlenie odpowiedniego komunikatu.
6. Odtworzenie odpowiedniego utworu muzycznego.

# Klasy systemu

* 1. **Diagram klas**



Rys.2 Diagram klas

* 1. **Comand Parser**

CommandParser to narzędzie odpowiedzialne za odczytywanie komend wpisywanych przez użytkownika. Jego główną składową jest klasa CommandParser. Jej elementem jest comandsDictionary, którego kluczami są obsługiwane komendy a wartościami delegaty do odpowiednich metod w klasie StoreSystem. Dzięki takiemu sposobowi możliwe jest łatwe rozszerzanie funkcjonalności systemu. Każda komenda wpisywana przez użytkownika składa się identyfikatora oraz opcjonalnych parametrów. Jej weryfikacja przebiega wieloetapowo. Najpierw sprawdzane jest czy komenda nie jest pusta. Następnie określa się czy komenda jest obsługiwana przez system oraz czy została podana odpowiednia liczba parametrów. Poprawność zawartości parametrów sprawdzania jest dopiero w klasie SystemStore. Commad Parser odpowiedzialny jest też za wyświetlanie komunikatów o nawiązywaniu połączenia oraz o udanym bądź nieudanym połączeniu z serwerem a także wiadomość powitalną.

* 1. **Store System**

Klasa StoreSystem zawiera w sobie całą logikę biznesową sklepu. Odpowiedzialna jest za takie działania jak weryfikacja, autoryzacja i zarządzanie użytkownikami, obsługę koszyka, finalizowanie zamówień, wyszukiwanie i wyświetlanie informacji o produktach w sklepie a także dodawanie nowych i edytowanie już istniejących. W tej klasie dokonuje się też zmian w bazie danych. Ponadto wywołuje ona metody odpowiedzialne za zarówno tradycyjne komunikaty tekstowe jak i zupełnie unikatowe w aplikacjach sklepowych komunikaty głosowe.

* 1. **Comunicates factory**

Comunicates Factory to klasa odpowiedzialna za przekazanie do wyświetlenia odpowiedniego komunikatu. Składa się z pola typu dictionary communicatesDictionary oraz metody GetCommunicate. Kluczami w słowniku są rodzaje komunikatów np. LoginAccepted wyświetlany po udanym logowaniu. Typy komunikatów przechowywane są za pomocą typu wyliczeniowego ComunicatesKinds. Wartościami słownika są zmienne typu string, które zostają wyświetlone użytkownikowi. W tym przykładzie jest to „Zalogowano pomyślnie.”. Metoda GetCommunicate zwraca „wyłuskany” ze słownika komunikat o kluczu otrzymanym poprzez parametr.

# Model TCP/IP

Model TCP/IP - teoretyczny model warstwowej struktury protokołów komunikacyjnych. Model TCP/IP został stworzony w latach 70. XX wieku w DARPA, aby pomóc w tworzeniu odpornych na atak sieci komputerowych. Potem stał się podstawą struktury Internetu.

Model logiczny stosu protokółów TCP/IP składa się z czterech warstw:

* warstwy interfejsu sieciowego
* warstwy internetowej
* warstwy transportowej
* warstwy aplikacji.

Warstwa interfejsu sieciowego (ang. Network Interface layer) jest odpowiedzialna za przekazywanie i odbieranie pakietów z kanału transmisyjnego. Stos protokółów TCP/IP został przy tym tak zaprojektowany, aby uniezależnić się od rodzaju kanału transmisyjnego, formatu ramki fizycznej czy architektury sieciowej

Warstwa internetowa (ang. Internet layer) jest odpowiedzialna za adresowanie, podział na pakiety oraz routing.

Warstwa transportowa (ang. Transport layer) gwarantuje poprawna komunikacje miedzy komputerami w sieci oraz przepływ danych miedzy warstwa internetowa a warstwa aplikacji. Do podstawowych protokółów warstwy transportowej należą:

* Protokół TCP
* Protokół UDP

Warstwa procesowa czy warstwa aplikacji (ang. process layer) to najwyższy poziom, w którym pracują użyteczne dla człowieka aplikacje takie jak np. serwer WWW czy przeglądarka internetowa.

# Testowanie i poprawki systemu

W trakcie procesu rozwoju aplikacji istotną fazą jest faza testów. W przypadku naszego programu testowanie było przeprowadzone przez osoby implementujące dane zagadnienie.

Wewnętrzne testy były przeprowadzane metodą testów użytkownika – podczas rozwoju aplikacji programiści na bieżąco sprawdzali jakość i bezawaryjność swoich rozwiązań wcielając się chwilowo w potencjalnego użytkownika.

Największy nacisk położono na przetestowanie poniższych zagadnień:

- testowanie stopnia realizacji wymagań funkcjonalnych przez autorów,

- testowanie procesów autoryzacji i autentykacji w aplikacji.

- testowanie jednoznaczności wykonywanych operacji w odniesieniu do poleceń wpisywanych do konsoli

- testowanie skończoności i poprawności wykonywanych operacji.