# 1. 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.

# 2. Funkcjonalności programu

Podczas projektu aplikacji założono, iż będzie ona obsługiwana z poziomu konsolowej lini 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 RADKA.

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

Poniżej zaprezentowano funckjonalności 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.

# 2.1 Funkcjonalności programu dostępne dla wszystkich

2.1.1. 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);

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.

2.1.2. 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);

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.

2.1.3. 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);

Poniżej przedstawiono kroki, które realizuje powyższy kod:

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

2.1.4. Czyszczenie konsoli

Funkcjonalność umożliwiająca na wyczyszczenie okna poleceń

Console.Clear();

Play(global::System.Reflection.MethodBase.GetCurrentMethod().Name);

return CommunicatesFactory.GetCommunicate(CommunicatesKinds.Cleaned);

Poniżej przedstawiono kroki, które realizuje powyższy kod:

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

2.1.5. 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);

2.1.6. 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);

Poniżej przedstawiono kroki, które realizuje powyższy kod:

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

2.1.7. 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;

}

Poniżej przedstawiono kroki, które realizuje powyższy kod:

1. Nie wiem.
2. No nie wiem.

2.1.9. Zakończenie pracy programu

Funkcjonalność umożliwia zakończenie pracy programu.

IsWorking = false;

return CommunicatesFactory.GetCommunicate(CommunicatesKinds.Thanks);

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.

# 2.2 Funkcjonalności programu dostępne dla administratora

2.2.1. 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);

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.

2.2.2. 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);

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.

2.2.3. 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);

}

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.

# 2.3 Funkcjonalności programu dostępne dla klienta

2.3.1. 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);

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.

2.3.2. Usuwanie produktu z koszyka

Podstawowa funkcjnalność kliencka, która umożliwa 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);

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.

2.3.3. 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();

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.

2.3.4. Podsumowanie zakupów i złożenie zamówienia

Funkcjonalność umożliwie 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);

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.