POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Andrzej, Wiktor Nowak
Nr albumu s10018

PROGRAMOWANIE ZORIENTOWANE BEHAWIORALNIE, JAKO RECEPTA NA PROBLEMY ZWIĄZANE Z TTD

Praca magisterska napisana
pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, luty 2014

Streszczenie

Niniejsza praca zawiera analizę podejść do testowania oprogramowania, jako niezbędnego elementu procesu wytwórczego produktu, jakim jest dowolna aplikacja pracująca w środowisku komputerowym. W rezultacie określone zostały najlepsze praktyki, mechanizmy i procedury, których zastosowanie podczas tworzenia oprogramowania, będzie miało istotny i pozytywny wpływ na jakość efektu końcowego w postaci dowolnego rodzaju aplikacji, czy systemu informatycznego.

Na podstawie określonych w części teoretycznej zasad dotyczących praktyk testowania oprogramowania (TDD, BDD) zostały zdefiniowane wymagania dotyczące rozszerzenia aplikacji Microsoft Visual Studio wspierającego stosowanie praktyk zawartych w ramach doktryny BDD, a następnie zrealizowane samo rozszerzenie.

Spis treści

1. Wstęp

Proces tworzenia oprogramowania komputerowego jest wieloetapowy i często długotrwały, a sam produkt końcowy nierzadko bardzo złożony. Owoc współpracy kilkunastu lub kilkudziesięciu osób, aby był funkcjonalny i wolny od wad musi być nie tylko dobrze zaplanowany, a oczekiwania klienta dobrze określone, ale przede wszystkim musi być gruntowanie przetestowany. Testowanie aplikacji powinno się odbywać na każdym etapie jej tworzenia za pomocą testów jednostkowych, integracyjnych, systemowych, czy akceptacyjnych. Ciężar niniejszej pracy skupiony jest na testach jednostkowych i metodykach określających sposób pracy podczas budowania kodu oraz testów, jak i ich utrzymania.

Obecnie najpopularniejszymi metodykami, dla złożonych projektów są *Test Driven Development* oraz *Behavioral Driven Development*, które operują przede wszystkim na poziomie testów jednostkowych. Dla wspomnianych metodyk jest dostępnych szereg narzędzi wspierających programowanie w oparciu o nie, lecz każde z nich jest specyficzne i niekoniecznie uniwersalne.

1.1. Cel pracy

Celem pracy jest określenie najlepszych praktyk w tworzeniu złożonego oprogramowania w oparciu o TDD i BDD oraz na ich podstawie przygotowania wymagań funkcjonalnych dla narzędzia wspierającego pracę osób zaangażowanych w proces wytwórczy oprogramowania od analizy i specyfikacji, przez, implementację, po testy.

1.2. Rozwiązanie przyjęte w pracy

Na potrzeby tej pracy został przygotowany prototyp narzędzia wspierającego osoby odpowiedzialne za przygotowywanie i opisywanie testów opartych o TDD i BDD oraz implementujących je programistów. Prototyp został przygotowany dla platformy Visual Studio firmy Microsoft działającej w środowisku .NET na systemach operacyjnych z rodziny Windows.

1.3. Rezultaty pracy

Wynikiem poniższej pracy jest koncepcja rozwiązania wspomagającego pracę na styku zespołu programistycznego z przedstawicielami klienta lub analitykami specyfikującymi wymagania funkcjonalne aplikacji. W ramach koncepcji stworzone zostało narzędzie dla jednego z popularnych środowisk programistycznych ukazujące sposób działania i korzyści wynikające z zastosowania BDD.

Prototyp składa się z kilku elementów:

- Narzędzie wspomagające pisanie opisów funkcjonalności w postaci scenariuszy testowych
- Parser stworzonego opisu scenariuszy
- Generator kodu C#, który będzie zawierał odpowiednie klasy i metody do implementacji przez programistę.

1.4. Organizacja pracy

W pierwszej kolejności zostały opisane metodyki wytwarzania oprogramowania i ich ewolucja na przestrzeni lat.

Następnie przedstawiona została metodyka BDD wraz z jej założeniami. Przeprowadzona została analiza stanu sztuki, gdzie zaprezentowano narzędzia wraz z opisem części wspólnych i wyróżniających. W podsumowaniu wskazane zostały wady obecnych rozwiązań.

W rozdziale 3 zaprezentowana został głęboka analiza problemu i rozwiązanie eliminujące wady dostępnych obecnie narzędzi.

W procesie analizy problemu zostały wybrane i opisane technologie wykorzystane przy realizacji rozwiązania przyjętego w pracy, które wyeliminuje wskazane niedoskonałości.

Rozdział 5 poświęcony został prezentacji prototypu implementującego założenia przedstawione w pracy dyplomowej.

Na koniec w podsumowaniu wskazane zostały trudności, jakie wystąpiły podczas realizacji prototypu, wykryte wady. Wskazane też zostały ścieżki dalszego rozwoju.

2. Metodyki testowania oprogramowania

Tworzenie oprogramowania na potrzeby biznesu rozpoczęło się około 50 lat temu w czasie, kiedy kod był wykonywany na pierwszych maszynach, na których posiadanie pozwolić sobie mogły tylko najbogatsze firmy. Z powodu małej dostępności mocy obliczeniowej komputerów, kod był często tworzony na kartce, a następnie uruchamiany na serwerze innej firmy, która sprzedawała do niego dostęp w określonym czasie. W tym okresie testowanie było bardzo ograniczone i trudne ze względu na to, że pomiędzy kolejnymi uruchomieniami testowanej aplikacji mijało nawet do kilku dni.

Przez kolejne lata na rynku pojawiały się pierwsza i kolejne generacje komputerów osobistych, które umożliwiły dostęp do narzędzi programistycznych bez konieczności używania serwerów. Dzięki temu możliwym stało się testowanie małych kawałków kodu zaraz po ich wytworzeniu, jednak nie powstała w tamtym czasie żadna powszechna metodyka wytwarzania oprogramowania, obejmująca kolejne etapy.

2.1.1 Metodyka wodospadu

Pierwszym szerzej znanym podejściem do tworzenia programowania, w tym również do testowania była metodyka wodospadu (waterfall) [Bender, James; McWherter, Jeff], która zakłada sekwencyjne podejście do zadań w obrębie cyklu tworzenia oprogramowania:

1. Opracowanie wymagań biznesowych przez odpowiedni zespół
2. Drobiazgowe zaprojektowanie całego systemu przez zespół architektów
3. Realizacja projektu przez dział programistów
4. Przeprowadzenie testów oprogramowania przez zespół testerów.

Proces testowania oprogramowania w takim przypadku, jest bardzo długi, kosztowny i mało wydajny i opiera się na ręcznym testowaniu aplikacji. Po pierwszym podejściu do testów zgłaszane są duże ilości błędów, przy których programista musi się zagłębić w kod, aby wykryć, gdzie leży błąd. Dodatkowo scenariusze testowe wg., których przebiegają testy są często pisane przez samych programistów, co może powodować i często powoduje, że nie zawierają one wszystkich przypadków, które mogą wystąpić. Dodatkowo programista tworząc taki scenariusz, wie jak pracuje kod aplikacji, a użytkownik końcowy takiej wiedzy nie będzie posiadał, a jedynie informacje, jak działa interfejs do tej aplikacji.

Metodyka wodospadu była używana w trakcie realizacji przeróżnych projektów łącznie z tymi, których okres realizacji był dłuższy niż 18 miesięcy. Niestety w ich przypadku często efekt końcowy, zaprojektowany od początku do końca w jednym punkcie czasowym na początku realizacji, nie odpowiadał aktualnym potrzebom klienta po całym procesie. Całość była wydłużana poprzez kolejne iteracje dostosowywania systemu, lub klient dostawał produkt, który nie odpowiadał już jego potrzebom.

Obecnie ta metodyka jest nadal używana, jednak zaleca się ją jedynie do projektów, których czas trwania jest nie dłuższy niż 6 miesięcy. Również docelowy kształt systemu oraz zakres wymagań powinny być znane i dobrze określone [Ganeshan].

2.1.2 Iteracyjne podejście do metodyki wodospadu

Ograniczenia metodyki wodospadu spowodowały, że niektóre firmy tworzące oprogramowanie dzieliły projekt na kilka lub kilkanaście mniejszych. Do każdego z nich była stosowana metodyka wodospadu, co powodowało to, że projekty były krótsze i mniejsze. Nie mniej sama metodyka nie uległa drastycznej zmianie, ale testowanie poszczególnych „kawałków" aplikacji, wprowadzanie zmian do założeń było łatwiejsze.

Takie podejście jest nazywane *Iteracyjnym*, lub *Inkrementalnym* [Bender, James; McWherter, Jeff].

2.1.3 Zwinne metodyki programowania

Nowe podejście do programowania i testowania było niezbędne ze względu na ciągle ulegające zmianie wymagania. Wymagania każdego projektu trwającego powyżej roku będą się zmieniały w trakcie jego realizacji, dlatego podejście znane z metodyki wodospadu musiało ewoluować.

W roku 2001 na spotkaniu przedstawicieli środowisk programistycznych został wypracowany manifest zwinnego programowania (*Agile Programming/Agile Methodologies*). Główne założenia obejmują [Agile]:

- osiągnięcie satysfakcji klienta poprzez szybkość wytwarzania oprogramowania,
- działające oprogramowanie jest dostarczane okresowo (raczej tygodniowo niż miesięcznie),
- podstawową miarą postępu jest działające oprogramowanie,
- późne zmiany w specyfikacji nie mają destrukcyjnego wpływu na proces wytwarzania oprogramowania,
- bliska, dzienna współpraca pomiędzy biznesem a developerem,
- bezpośredni kontakt, jako najlepsza forma komunikacji w zespole i poza nim,
- ciągła uwaga nastawiona na aspekty techniczne oraz dobry projekt (design),
- prostota,
- samozarządzalność zespołów,
- regularna adaptacja do zmieniających się wymagań.

Powyższy manifest nie jest metodyką samą w sobie. Jest to lista najważniejszych cech i założeń, których spełnienie przez konkretną metodykę pozwala zaklasyfikować ją do jednego szeregu z innymi metodykami zwinnego programowania, jak komercyjne *Scrum, Extreme Programming (XP), Feature Driven Developement, Clear Case, Adaptive Software* [Bender, James; McWherter, Jeff].

Jest wiele metodyk, które można zaklasyfikować do zwinnych, m. in. *Test Driven Development* jest jedną z nich.

2.2. Metodyka TDD

Metodyka *Test Driven Develoopment* (TDD) wyrosła z wcześniej stosowanej *Extreme Programming (XP). XP* zakłada, *TDD* również, że podstawową zasadą tworzenia oprogramowania jest testowanie na jak najwcześniejszym etapie, co oznacza w praktyce, że testy powinny być wykonywane na jak najmniejszych porcjach kodu, czyli na jednostkach.

2.2.1 Test jednostkowy

Metodyka *TDD* bazuje na testach jednostkowych, stąd potrzeba określenia one są:

Definicja 1 [Roy]

Test jednostkowy jest kawałkiem kodu (zazwyczaj metodą), która wywołuje inny kawałek kodu i sprawdza poprawność pewnych założeń. Jeżeli założenia są błędne, wtedy test kończy się niepowodzeniem. Jednostką, której dotyczy test jest metoda lub funkcja.

Zgodnie z powyższą definicją test jednostkowy odnosi się do metody lub funkcji, natomiast przy ogromnej popularności programowania obiektowego, często niemożliwym jest testowanie poszczególnych metod w oderwaniu od obiektu, w którym są umieszczone. Stąd, aby zapewnić skuteczność testów jednostkowych i ich niezależność od reszty kodu niezwiązanego z daną klasą, poszczególne testy muszą określać dodatkowo stan początkowy obiektu.

Przykładem może być prosta klasa, gdzie jedna z metod zwracająca opis na podstawie identyfikatora w różnych językach. Klasa ta zwróci różne opisy dla jednego identyfikatora w zależności, jak ma zainicjowaną zmienną określającą język. Jeżeli test tej metody nie będzie definiował zmiennej języka, może się okazać, że przy niewielkiej zmianie domyślnej wartości w klasie, okaże się, że testy nie będą działały poprawnie, a nawet rezultaty mogą być różne w trakcie kolejnych uruchomień.

Równie istotną aspektem jest separacja testowanego kodu od reszty kodu aplikacji. Szczególnie istotne jest to w architekturze *Model View Controller* (*MVC*), który zakłada, że klasa *Controllera* będzie odwoływała się do modelu, celem pobrania, dodania lub modyfikacji danych. Tutaj pojawia się kilka problemów:

- Uruchamianie testów w różnych lokalizacjach i maszynach ze względu na pracę zespołową
- Oderwanie źródła danych od samego kodu testu. Przyjmując bazę relacyjną za źródło danych, można łatwo zauważyć, że jest szereg warunków jak dostępność bazy, uprawnienia użytkownika, jej schemat, uprawnienia użytkownika. Dodatkowo taka baza danych może ulegać naturalnemu procesowi zmian podczas rozwoju oprogramowania, co może niekorzystnie wpływać na testy jednostkowe.
- Jednoczesne uruchamianie testów w lokalnych wersjach projektu przez rożnych programistów
- Niedostępność kodu lub elementów aplikacji, do których się odwołuje kod ze względu na rozdzielenie prac między różne zespoły lub trywialnie ze względu na harmonogram prac.

Dlatego tak istotne jest udawanie innych elementów systemu na poziomie testu za pomocą zaślepek (z angielskiego *mock* objects). Można to zobrazować na przykładzie klasy do autentykacji

użytkownika. Jedna z metod tej klasy przyjmuje login i hasło jako argumenty i zwraca wartość *prawda* w przypadku, kiedy taki użytkownik widnieje w bazie i posiada wskazane hasło oraz *fałsz* w przypadku, kiedy nie ma takiego użytkownika, lub ma inne hasło. Testowana metoda, żeby móc zwrócić wynik musi sięgnąć do bazy. Dla ułatwienia jej parametry są zapisane na sztywno w kodzie. W takim przypadku test będzie obejmował również inne elementy jak sprawność łącza sieciowego, dostępność bazy danych, zawartość bazy danych itd. Test jednostkowy zgodnie z definicją ma testować jedynie najmniejszy możliwy kawałek kodu, czyli samą metodę, dlatego w teście należy umieścić swojego rodzaju zaślepkę. Dodatkowym atutem takiego podejścia jest przyspieszenie testów, ponieważ odbywają się one w obrębie jednej maszyny i nie sięgają do zasobów zdalnych, co zawsze trwa dłużej

2.2.2 Krokowość działań według metodyki TDD

Cykl tworzenia oprogramowania w metodyce *TDD* zakłada następujące kroki dla każdej nowej funkcjonalności o znanych wymaganiach [Adzic]:

1. Utworzenie testu dotyczącego nowej funkcjonalności ma podstawie opisu powstałego w wyniku analizy zapotrzebowania. Zgodnie z definicją utworzony kod, powinien zawierać wywołanie kodu testowanej funkcjonalności. W związku z tym, że test jest tworzony przed implementacją funkcjonalności, powinien on zwrócić błąd. W zależności od użytego języka może to być błąd, wyjątek, błąd kompilatora. W przeciwnym wypadku, gdyby test się powiódł, oznaczałoby, ze taka funkcjonalność jest już zaimplementowana lub nazwa wybranego obiektu lub metody już istnieje.

2. Dodanie reszty kodu testu opierając się na wymaganiach w postaci przypadków użycia (*use case*) i scenariuszy użytkownika *(user story*). Napisanie testów wymaga od programisty zrozumienia wymagań biznesowych zanim zacznie pisać kod implementacji samej funkcjonalności.

3. Programista uruchamia wszystkie testy. Wszystkie powinny zakończyć się niepowodzeniem. W przeciwnym wypadku te, które zakończyły się sukcesem są błędnie napisane i w efekcie bezużyteczne.

4. Napisanie implementacji funkcjonalności i kolejne uruchamianie testów aż do momentu, kiedy kod będzie prawidłowy i wszystkie testy zakończą się sukcesem.

5. Uruchomienie wszystkich testów, które zakończą się pozytywnie. W tym momencie programista jest pewien, że nowa zaimplementowana przez niego funkcjonalność spełnia wymagania.

6. *Refactoring* – zamiana kodu implementacji tak, aby był czytelny i łatwy w utrzymaniu. Ewentualnie kod może wymagać optymalizacji, aby dział sprawniej.

Tylko tak zdefiniowany cykl pracy programisty, gdzie najpierw tworzony jest kod związany z testem na podstawie założeń biznesowych, a dopiero potem kod implementujący funkcjonalność, daje pewność, że: [Bender, James; McWherter, Jeff]

- programista dobrze rozumie wymagania klienta, jeszcze zanim przystąpi do swojej pracy, zatem jest duża spójność kodu z wymaganiami.

- funkcjonalności zawierają dokładnie tyle kodu, ile jest niezbędne, aby wypełnić warunki sprawdzane przez testy. Dzięki temu nie ma niepotrzebnego kodu, a im mniej kodu, tym mniejsza szansa na jego wadliwość. Dodatkowo kod jest łatwiejszy w utrzymywaniu.
- biblioteki i API tworzone w oparciu o TDD są prostsze, a dzięki temu, że programista je tworzący jest ich pierwszym użytkownikiem, często są bardziej sensowne i spójne,
- tworzenie testów przez programistę wymusza na nim więcej komunikacji z klientem lub analitykiem, przez co, tworzony kod przyjmuje i zwraca dokładnie te dane, jakie są niezbędne.
- testy tworzone i utrzymywane przez programistów podczas dodawania lub modyfikowania funkcjonalności świetnie nadają się do testów regresji

2.2.3 Niedoskonałości TDD i wynikające z tego problemy

Mimo dużego nacisku położonego w *TDD* na komunikację pomiędzy programistami, a biznesem, nie jest ona wystarczająca. *TDD* zaleca używanie opisowych nazw metod zawierających testy, przykładem może być test funkcjonalności zliczającej wystąpienia ciągu znaków w dłuższym tekście. Metoda testowa mogłaby się nazywać `niechZnajdzieJednoZdanieWSłowaTworząZdanie` i sprawdzać, czy w tekście *„Słowa tworzą zdanie.”* testowana funkcjonalność znajdzie dokładnie jedno wystąpienie słowa *„zdanie”*. Na tak prostym przykładzie widać, że nazwa metody dostarcza podstawowych informacji, ale trudno będzie w niej zawrzeć wszystkie szczegóły danej funkcjonalności jak:

- Informacje o stanie systemu w momencie rozpoczęcia testu (kontekst) np.:
- Zalogowany użytkownik
- Stan bazy danych przed operacją
- Stan niektórych zmiennych, albo sesji
- Opis akcji wywołującej użycie metody np.:
- Klient zwraca towar
- Obiekt biznesowy żąda wydruku dokumentu
- Opis oczekiwanego rezultatu
- Stan magazynu zwiększa się o ilość zwróconego towaru
- Dokument wydrukowano z sukcesem na drukarce o nazwie

Nie sposób jest zawrzeć powyższe informacje w nazwie metody, a jeśli nawet się to uda, nazwa nie będzie jednoznaczna i czytelna. Zatem wiele informacji będzie zapisanych za pomocą języka programowania nieczytelnego dla większości osób. To zdecydowanie utrudnia uczestnictwo biznesu w tworzeniu i akceptacji testów. Przez to nie ma pewności, czy programista poprawnie zrozumiał wymagania stawiane danej funkcjonalności.

Jednocześnie bardzo częstym problemem jest poprawne zrozumienie przez programistę wymagań dotyczących funkcjonalności stworzonych na podstawie przeprowadzonej analizy. Opis w takim dokumencie często zawiera nieadekwatną liczbę szczegółów, która może prowadzić do niepełnego zrozumienia niesionej przez analizę informacji, lub też zaciemnienie obrazu. Stąd wynika jeszcze jedna

potrzeba nierozwiązana przez *TDD*, mianowicie ujednolicenie i usystematyzowanie za pomocą pewnej stałej struktury tekstu opisującego funkcjonalność.

Podczas próby wdrożenia *TDD*, problemem może się okazać brak wystarczających argumentów dla kadry zarządzającej. Tworzenie testów jest czasochłonne i w pierwszej chwili może się wydawać, że proces wytworzenia oprogramowania będzie przez to dłuży i droższy. W dodatku brak konieczności stosowania i wewnętrzny charakter metodyki nie pomagają w przekonaniu osób odpowiedzialnych w firmach do wprowadzenia *TDD*. Dlatego mimo wielu zalet *TDD* i oczywistych zysków, jakie idą za tą metodyką, w wielu firmach jej wdrożenie staje się niemożliwe.

2.3. Metodyka BDD

Metodyka *Behavioral Driven Development* (*BDD*) jest silnie związana z *TDD* i również kładzie duży nacisk na:

- Odpowiednią komunikację pomiędzy zespołem programistów a twórcami wymagań
- Stosowania testów jednostkowych i kolejności tworzenia poszczególnych elementów (testy najpierw, później implementacja samej funkcjonalności)

BDD jednak odróżnia się istotnie *TDD* jednym najważniejszym założeniem. Treść testów i założenia dotyczące walidacji funkcjonalności powinny być tworzone, a co najmniej akceptowane przez twórców wymagań, czyli analityka, lub też samego odbiorcy systemu.

2.3.1 Język naturalny w BDD

Główną cechą *BDD* odróżniającą od innych metodyk jest założenie, że testy powinny być opisane za pomocą języka możliwie zbliżonego do języka naturalnego.

Definicja 2 [Świdziński]

Język naturalny jest to dwuklasowy system znaków wykorzystywany przez daną grupę społeczną do porozumiewania się o wszystkim.

W Definicja 2 zwrot „dwuklasowy system” oznacza, że język składa się ze znaków prostych i złożonych, a sposób łączenia pierwszych w drugie to gramatyka. Z innej strony można język naturalny określić, jako sposób komunikacji międzyludzkiej pozwalający wyrazić wszystko.

Naturalność przekazu w ramach opisu wymagań, przypadków użycia została przeniesiona również na opis testów jednostkowych, a co za tym idzie możliwość wymiany tych opisów pomiędzy każdym członkiem zespołu abstrahując od jego umiejętności programistycznych lub opanowanych technologii. Idąc dalej, takie podejście umożliwia włączenie klienta w tworzenie i akceptację testów jednostkowych i minimalizację ryzyka nieporozumień na styku zespół rozwijający produkt, a analityk lub klient.

Język naturalny ma kilka bardzo ważnych z punktu widzenia *BDD* cech:

- Kontekstowość, co oznacza, że w zależności od otoczenia danego zwrotu lub słowa zależy jego znaczenie
- Niejednoznaczność, zatem jedno słowo może mieć wiele znaczeń
- Dziedzinowość, która oznacza, że słowa odnoszące się do pewnego obszaru wiedzy nie zawsze mają to samo znaczenie, jak w języku potocznym.Jest to pewne uszczegółowienie niejednoznaczności.
- Ponadto język naturalny jest niesłychanie trudny do analizy i przetworzenia na konkretne konstrukcje programistyczne.

To wszystko sprawia, że opis w języku naturalnym bez narzuconych żadnych ram i konstrukcji może okazać się trudny do zrozumienia przez osoby niezwiązane z analizą, bądź nieznających specyfiki projektu. Szczególnie programiści są osobami, które mają na pewno z początku ograniczoną wiedzę o charakterze danej dziedziny, a ich zadaniem jest implementacja kodu, a nie nauka np. księgowości. Również grono testerów może nie posiadać odpowiednich kwalifikacji, aby móc w pełni merytorycznie odpowiadać za specyfikowanie lub interpretację scenariuszy testowych. Do tego dochodzi niesłychana elastyczność języka naturalnego i nieskończona liczba konstrukcji zdaniowych, która utrudnia automatyczną analizę tekstu i wykorzystanie narzędzi wspierający proces wytwórczy oprogramowania. Dlatego *BDD* zakłada wykorzystanie języka możliwe zbliżonego naturalnego, a nie języka naturalnego

2.3.2 Specyfikowanie testów za pomocą języka naturalnego

Metodyka zakłada, że każdy przypadek testowy powinien być opisany w sposób zbliżony do opisu scenariusza użytkownika (*user story*) za pomocą trzech głównych elementów:

- Tytuł – nazwa każdej dla opisywanej historii (funkcjonalności)
- Opis – krótki wstęp mówiący o:
- Kto – rola, albo obiekt biznesowy, lub też aktor systemu
- Jakie efekty ma przynieść użycie funkcjonalności
- Jaka jest biznesowa wartość funkcjonalności, inaczej cel
- Scenariusz - opis każdego możliwego przypadku, jaki jest brany pod uwagę dla funkcjonalności, czyli kryteriów akceptacji.
- Określenie wstępnych warunków lub warunków, które określają stan początkowy przed rozpoczęciem scenariusza
- Określenie elementu inicjującego scenariusz, czyli akcji w wyniku, której powinien zostać osiągnięty efekt
- Określenie efektu wyjściowego za pomocą jednego lub więcej warunków.

Wykorzystywanie przypadków użycia w specyfikacji określającej funkcjonalność jest kluczowe z tego względu, że język naturalny nie jest jednoznaczny, ponadto jest kontekstowy. Dlatego do poprawnego zrozumienia nawet prostego opisu może okazać się niezbędna wiedza z zakresu danej dziedziny, a użyte zwroty mogą być błędnie rozumiane przez programistę. Metodyka *BDD* szeroko korzysta z przypadków użycia narzucając konieczność konstruowania tych opisów w języku dziedzinowym.

Język dziedzinowy służy do łatwego opisywania dobrze określonej dziedziny, która w przypadku metodologii *BDD* to przypadki użycia, a szerzej opis funkcjonalności. Wymagania wobec takiego języka to przede wszystkim

- Prostota – korzystać z niego będzie szerokie grono osób, gdzie nie wszystkie będą biegłe w programowaniu, a tym bardziej w ściśle określonej technologii projektu Składnia nie może być skomplikowana.
- Dopasowanie do opisywanej dziedziny. Język powinien umożliwiać opisanie dowolnego przypadku użycia.
- Możliwość dalszego przetwarzania – stworzone opisy mogą być wtedy szybko rozwijane do prostych konstrukcji programistycznych, które zwiększą produktywność zespołu

2.3.3 Język dziedzinowy Gherkin

Twórca *BDD* Dan North w 2006 roku [North] zaproponował proste i skuteczne rozwiązanie problemu poprzez zastosowanie prostej składni z kilkoma słowami kluczowymi. Rozwiązanie było natywne dla języka angielskiego z dość prostą składnią i kilkoma słowami kluczowymi. Na Listing 1 zaprezentowany jest przykład w języku polskim:

Listing 1 - założenia *BDD* dotyczące języka opisującego funkcjonalność

```
Opis: Zwroty wracają na stan magazynu

W celu utrzymania aktualnego stanu magazynu
Jako właściciel sklepu
Chcę dodawać zwrócone produkty z powrotem na stan magazynu

Scenariusz 1: Zwrócone produkty powinny wrócić na stan magazynu
Dany klient wcześniej kupił czarny sweter ode mnie
I aktualnie mam na stanie trzy czarne swetry na stanie
Kiedy klient zwraca sweter
Wtedy powinienem mieć cztery czarne swetry na stanie magazynu

Scenariusz 2: Wymieniony produkt powinien być zwrócony do magazynu
Dane jest, że klient kupił niebieską część garderoby
I mam dwie niebieskie części garderoby na stanie magazynu
I mam trzy czarne części garderoby na stanie magazynu
Kiedy klient zwraca niebieską część garderoby w celu wymiany na czarną
Wtedy powinienem mieć trzy niebieskie części garderoby na stanie magazynu
I dwie czarne części garderoby na stanie magazynu
```

Zaprezentowany opis zawiera scenariusze testowe dotyczące jednej funkcjonalności. Powyżej pogrubione zostały słowa kluczowe: Opis, Scenariusz W celu, Jako, Chcę, Dana/Dany/Dane, Kiedy, Wtedy, I, które określają logiczną strukturę testu. Dzięki zastosowaniu słów kluczowych możliwym jest wyodrębnienie poszczególnych elementów niezbędnych do określenia testu jednostkowego, czyli określenia stanu początkowego poszczególnych zmiennych i stanu systemu, wykonywanych akcji, warunków walidujących poprawność wyniku:

	Scenariusz 1	Scenariusz 2
Warunki początkowe	Użytkownik: właściciel sklepu; Dokonana transakcja kupna czarnego swetra przez klienta; Na stanie są trzy czarne swetry;	Użytkownik: właściciel sklepu; Dokonana transakcja kupna niebieskiej części garderoby przez klienta; Na stanie są dwie niebieskie części garderoby Na stanie są trzy czarne części garderoby
Akcje do wykonania	Klient zwraca czarny sweter	Klient wymienia niebieską część garderoby na czarną
Warunki walidujące	Na stanie są 4 czarne swetry	Na stanie są trzy niebieskie i dwie czarne części garderoby

Tabela 1 – rozbicie przykładu z Listing 1 na wymagania testu jednostkowego

Zaprezentowany wcześniej opis jest opisem słownym wykorzystującym zdania języka naturalnego i jest zrozumiały dla osób, które niekoniecznie są związane z programowaniem. Ta cecha umożliwia aktywne uczestniczenie klienta biznesowego w procedurze budowy testów, co podniesie trafność określonych wymagań, ich walidacje przed rozpoczęciem implementacji, dookreślenie funkcjonalności. W efekcie programiści będą budowali dokładnie to, czego oczekuje klient i nie będą tracili czasu, na zmianę już gotowych funkcjonalności ze względu na błędne określenie potrzeb klienta lub ich błędną interpretację.

Stosowanie *BDD* poprzez wykorzystanie języka naturalnego do opisu funkcjonalności stanowi żyjącą, na bieżąco aktualizowaną dokumentację, którą można wykorzystać w dalszych etapach budowy systemu. Przy odpowiednim nacisku na testy, będą one zawsze aktualne, gdyż w przypadku konieczności wprowadzenia modyfikacji do funkcjonalności modyfikacja musi obejmować również testy.

Kiedy programista chce dokładnie dowiedzieć się, jak dana funkcjonalność działa wystarczy, że zajrzy do przygotowanych scenariuszy testowych i nie musi polegać na dodatkowej dokumentacji, która często jest niekompletna, lub nieodpowiadająca stanowi obecnemu. Dokumentacja może też służyć osobom wdrażanym w kod, którego nie tworzyli, bądź do odtwarzania wiedzy sprzed dłuższego okresu czasu, kiedy programista nie miał styczności z danym kawałkiem kodu.

2.4. Obecne rozwiązania

Obecnie dostępny jest szereg rozwiązań umożliwiających tworzenie oprogramowania w oparciu o BDD w środowisku Microsoft Visual Studio. Rozwiązania te pozwalają opisywać scenariusze testowe za pomocą API zbliżonego do języka naturalnego, a następnie implementować i wykonywać testy za pomocą wybranych frameworków.

2.4.1 Cucumber

Cucumber jest narzędziem linii poleceń pierwotnie przeznaczonym dla Ruby, które umożliwia definiowanie scenariuszy testowych za pomocą języka *Gherkin* oraz ich wykonywanie. Język *Gherkin* oparty jest o słowa kluczowe: *Feature, Background, Given, When, Then, And, But, *, Scenario*. Na Listing 2 przykład w języku polskim scenariusza zapisanego za pomocą:

Listing 2 - Przykładowy opis funkcjonalności opisanej za pomocą języka *Gherkin*

```
Opis: Zwroty wracają na stan magazynu

W celu utrzymania aktualnego stanu magazynu
Jako właściciel sklepu
Chcę dodawać zwrócone produkty z powrotem na stan magazynu

Scenariusz: Zwrócone produkty powinny wrócić na stan magazynu
Zakładając, że klient wcześniej kupił czarny sweter ode mnie
I aktualnie mam na stanie trzy czarne swetry na stanie
Kiedy klient zwraca sweter
Wtedy powinienem mieć cztery czarne swetry na stanie magazynu
```

Język polski jest jednym z kilkudziesięciu wspieranych przez narzędzie. Oczywiście najpopularniejszym jest język angielski, ze względu na to, że języki programowania są w tym języku i jest on często podstawowym dla dokumentacji oprogramowania. Zatem naturalnym będzie, że duża część programistów użyłaby właśnie angielskiego do opisu funkcjonalności i mogłoby to wyglądać tak [Agile]:

Listing 3 – Przykładowy opis funkcjonalności opisanej za pomocą języka *Gherkin*

```
Story: Returns go to stock

In order to keep track of stock
As a store owner
I want to add items back to stock when they're returned

Scenario 1: Refunded items should be returned to stock
Given a customer previously bought a black sweater from me
And I currently have three black sweaters left in stock
When he returns the sweater for a refund
Then I should have four black sweaters in stock
```

Tak skonstruowany opis zgodnie z wytycznymi języka *Gherkin* może być przetworzony przez narzędzie *Cucubmer* na klasę testową wraz ze zalążkami odpowiednich metod. Celem takiej operacji jest ułatwienie pracy programistów przez wygenerowanie części kodu, a jednocześnie zapewnienie, że każda metoda została umieszczona w odpowiednim pliku z kodem.

W powyższych przykładach słowa kluczowe *In order, As, I want* zostaną pominięte, ponieważ stanowią one jedynie dodatkowy opis, zatem ich implementacja nie jest konieczna. Natomiast na podstawie pozostałych słów kluczowych *Cucumber* automatycznie wygeneruje klasę testową zawierającą następujące elementy [Wynne, Matt; Hellesoy, Aslak]:

- Blok kodu odpowiedzialny za nadanie kontekstu funkcjonalności, czyli ustawienie wartości początkowych. Opis jakie mają być te wartości następuje bezpośrednio za *Given*, lub związanym z nim *And*. Każdy taki blok będzie opatrzony komentarzem zawierającym w naszym przykładzie odpowiednio:
    - Wymaganie dotyczące klienta, który kupił wcześniej sweter

Listing 4 – Przykład zastosowania słowa kluczowego Given w Gherkin

```
Given "a customer previously bought a black sweater from me" do
# TODO: implementacja niezbędnych czynności
end
```

- Wymaganie dotyczące obecnego stanu magazynu

Listing 5 - Przykład zastosowania słowa kluczowego Given *w* Gherkin

```
Given "I currently have three black sweaters left in stock" do
# TODO: implementacja niezbędnych czynności
end
```

- Blok kodu odpowiedzialny za wywołanie testowanej funkcjonalności zdefiniowany w opisie za pomocą *When*

Listing 6 - Przykład zastosowania słowa kluczowego Then *w Gherkin*

```
When "he returns the sweater for a refund" do
# TODO: implementacja niezbędnych czynności
End
```

- Weryfikację poprawności wyniku zwróconego przez funkcjonalność opisaną przez *Then*

Listing 7 - Przykład zastosowania słowa kluczowego Then w Gherkin

```
Then "I should have four black sweaters in stock" do
# TODO: implementacja niezbędnych czynności
End
```

Wygenerowana klasa zwiera jedynie zalążki klas i nie posiada implementacji, za co odpowiedzialny jest programista. Wspomniana klasa jest zapisana w języku programowania Ruby w oparciu, o który działa cały *Cucumber*. *Cucumber* jest też odpowiedzialny za wykonywanie testów i informowanie programisty o jego wynikach, czyli realizuje wsparcie dla *TDD* i *BDD* praktycznie w całości i nie wymaga dodatkowych narzędzi.

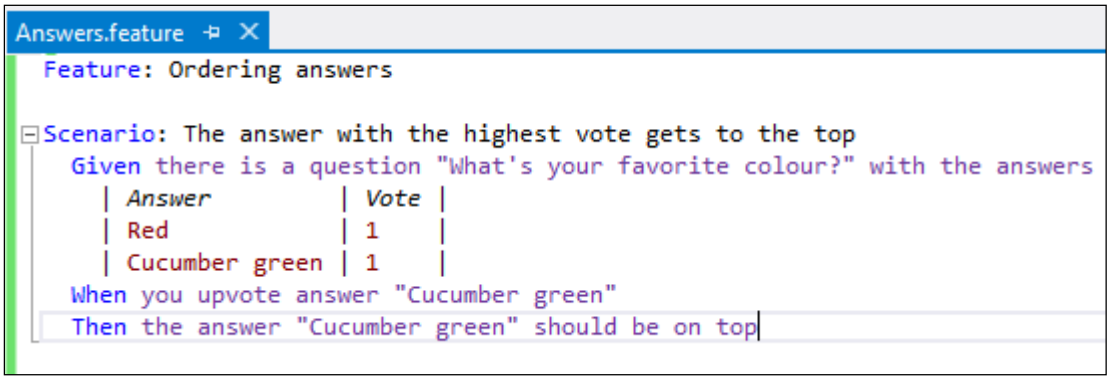
W przypadku połączenia *Cucubmer* z platformą *.NET* koniecznym jest instalacja dodatkowych narzędzi integrujących *.NET* z Ruby. Dużą niedogodnością jest konieczność pracy za pomocą zewnętrznych narzędzi w stosunku do *Visual Studio* i z poziomu linii poleceń. Dodatkowo należy posiadać doświadczenie z językiem Ruby.

Cucmber jest udostępniany w modelu licencyjnym *MIT*.

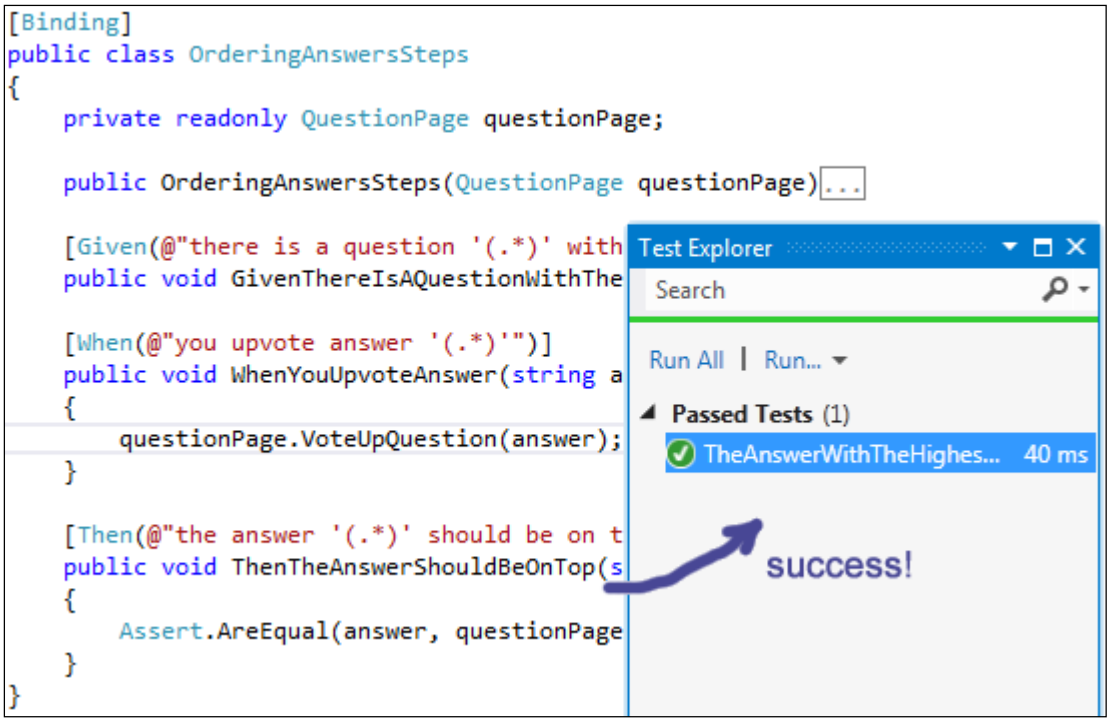
2.4.2 SpecFlow

Kolejnym frameworkiem umożliwiającym wykorzystanie zalet *BDD* jest *SpecFlow*, który korzysta z tej samej składni, co *Cucumber*, czyli z języka *Gherkin*, a to dlatego, że *SpecFlow* jest częścią tej samej rodziny, ale w przeciwieństwie do *Cucumber* w pełni się integruje ze środowiskiem *Visual Studio* [Sanderson, 2010]:

- Możliwość tworzenia nowych plików z poziomu Visual Studio
- Wsparcie dla debuggera poprzez możliwość ustawiania break-pointów
- Możliwość implementacji testów w dowolnym języku platformy .NET
- Podczas kompilacji tworzony jest plik zgodny z NUnit zawierający testy jednostkowe, zatem istnieje możliwość zastosowania dowolnego środowiska dla testów jednostkowych. Na oficjalnej stronie można znaleźć informację o wsparciu dla *NUnit, Ms Test, xUnit, mbUnit, SpecRun.*



Rysunek 1 - Plik opisujący funkcjonalność w narzędziu *SpecFlow* [SpecFlow]



Rysunek 2 - Plik wynikowy zawierający wygenerowany kod [SpecFlow]

SpecFlow jest otwartym narzędziem udostępnianym w modelu licencyjnym *BSD*.

2.4.3 StoryQ

StoryQ to kolejne narzędzie przeznaczone dla wszystkich chcących programować zgodnie z *BDD*, ale w przeciwieństwie do rozwiązań prezentowanych w poprzednich podrozdziałach definiowanie funkcjonalności i scenariuszy odbywa się za pomocą języka C# z wykorzystaniem odpowiedniej metody i przekazaniu jej odpowiednich parametrów. W tym samym pliku należy dodatkowo przygotować odpowiednie metody implementujące akcje opisane przez słowa kluczowe *Given*, *When*, *Then* [Writing your first StoryQ test, 2010]:

Listing 8 - Przykładowy opis funkcjonalności za pomocą *SpecFlow* [SpecFlow]

```
[TestMethod]
public void TestMethod1()
{
  new Story("Trying out StoryQ")
    .InOrderTo("see how StoryQ works")
    .AsA("developer")
    .IWant("to try a simple example")
    .WithScenario("first example")
    .Given(ThisIsMyFirstTry)
    .When(IRunThisMethod)
    .Then(IWillSeeWhatHappens)
}

public void ThisIsMyFirstTry() { throw new NotImplementedException(); }
public void IRunThisMethod() { throw new NotImplementedException(); }
public void IWillSeeWhatHappens() { throw new NotImplementedException(); }
```

StoryQ jest dystrybuowany w modelu licencyjnym *MIT*, a jego kod źródłowy dostępny na stronie projektu.

2.4.4 Wady dostępnych rozwiązań

Cucumber i *SpecFlow* to dwa narzędzia korzystające z składni języka *Gherkin*, który umożliwia opis poszczególnych funkcjonalności wraz ze scenariuszami testowymi w sposób zbliżony do języka naturalnego. Jednak *Cucumber* nie jest natywnym rozwiązaniem dla *Visual Studio*, dodatkowo wykorzystuje własne narzędzia do zarządzania i wykonywania testów jednostkowych, co sprawia, że jest mało elastyczny. To powoduje, że programiści niezwiązani z językiem Ruby bardzo niechętnie podchodzą do pomysłu zmiany narzędzia. Dodatkową wadą jest konieczność mostkowania pomiędzy językami dostępnymi w *.NET* a Ruby.

SpecFlow jest przygotowany do pracy z *Visual Studio*, dodatkowo pozwala na używanie dowolnego narzędzia do zarządzania testami np. *NUnit*. Takie podejście sprawia, że łatwiej go wdrożyć. Dużą wadą *SpecFlow* jest sposób, w jaki generuje on klasę testową i zalążki poszczególnych metod testowych. Efektem wyjściowym jest klasa z metodami, które programista musi wypełnić implementacją testów. W rezultacie niemożliwym jest użycie opisu jednej funkcjonalności wraz ze scenariuszami więcej niż raz, co stanowi duże utrudnienie w przypadku, kiedy testy jednostkowe są również wykorzystywane do testów akceptacyjnych i integracyjnych. Problem jest większy, jeżeli

rozważy się konieczność utrzymywania testów ze względu na zmiany wymagań, ponieważ dla różnych rodzajów testów są różne opisy tej samej funkcjonalności. O nich wszystkich trzeba pamiętać i kolejno wprowadzać często identyczne zmiany. Problemów może też przysporzyć aktualizacja pliku zawierającego opis funkcjonalności. Przy ponownym generowaniu zostanie nadpisany plik, który wcześniej mógł być już wypełniony implementacją testów.

StoryQ w odróżnieniu od wcześniej omawianych rozwiązań nie jest przygotowany do obsługi opisów funkcjonalności w języku zbliżonym do naturalnego. To jest jego ogromną wadą, ponieważ tworzenie dokumentacji jest utrudnione, a składnie jest mało zrozumiała dla osób niebędących programistami, a uczestniczącymi w procesie akceptacji wymagań.

2.4.5 Podsumowanie

W dalszych rozdziałach niniejszej pracy dyplomowej zostanie opracowana koncepcja, która będzie pozbawiona wad obecnie stosowanych rozwiązań i wprowadzi nowe, do tej pory niedostępne możliwości pracy związane z *BDD*.

1. Narzędzie będzie w pełni zintegrowane ze środowiskiem programistycznym poprzez
    a. Możliwość tworzenia plików z opisami funkcjonalności z poziomu aplikacji: menu głównego, menu kontekstowych
    b. Możliwość edycji plików wewnątrz środowiska programistycznego
    c. Nie będzie wymagało żadnych dodatkowych narzędzi jak w rozwiązaniu *Cucubmer*

2. Narzędzi będzie umożliwiało podświetlanie i podpowiadanie składni za pomocą mechanizmów środowiska programistycznego

3. Narzędzie wyeliminuje niedogodności związane z generowaniem zbyt szczegółowego kodu, jak to ma miejsce w *SpecFlow* i *Cucumber*. Dzięki temu jeden opis funkcjonalności będzie mógł posłużyć przy konstruowaniu opisów dla funkcjonalności w testach jednostkowych, akceptacyjnych, czy integracyjnych.
    a. Zamiast zwykłych klas będą automatycznie generowane abstrakcyjne klasy typu `interface`.
    b. Będzie możliwość tworzenia klas implementujących powyższe interfejsy przeznaczonych dla konkretnego typu testów.

4. Do opisu funkcjonalności zostanie zastosowany dziedzinowy język bliski naturalnemu, jak to ma miejsce w przypadku *Cucumber*, czy *SpecFlow*, który będzie można łatwo zastąpić językiem lokalnie stosowanym np. polskim, czy niemieckim.

Głównym i najważniejszym zadaniem stawianym przed prototypem jest automatyczne generowanie klas na podstawie opisów w języku zbliżonym do naturalnego. Język ten powinien mieć określoną składnię, która będzie umożliwiała tworzenie poprawnych zdań w dowolnym języku lokalnym. Ważne jest, by język zawierał słowa kluczowe, które poprawnie umieszczone w tekście będą umożliwiały tworzenie wcześniej wspomnianych klas. Świetnym przykładem takiego języka jest *Gherkin* wykorzystany w *Cucumber* i *SpecFlow*.

Interesującą funkcją dla programisty może być też umieszczanie w opisie funkcjonalności już gotowych zestawów danych wejściowych oraz oczekiwanych rezultatów. W ten sposób nie trzeba pisać

nieskończonej listy scenariuszy, a wystarczy podać kilkanaście, lub kilkadziesiąt przykładów dotyczących jednego scenariusza. Przykłady umieszczone bezpośrednio w opisie funkcjonalności mogą być ujęte w formie tabeli.

3. Koncepcja rozwiązania

Bieżący rozdział poświęcony został szczegółowemu opisowi rozwiązania, które umożliwi w pełni korzystanie z założeń *BDD* w ramach procesu wytwórczego oprogramowania. W kolejnych podrozdziałach omówione zostały ogólny zarys, założenia funkcjonalne i niefunkcjonalne.

3.1. Ogólny zarys

W tym podrozdziale opisane zostały ogólne wymagania dotyczące niezbędnych informacji w opisach, specjalnego języka na potrzeby tychże opisów, efekt ich analizy w postaci plików wypełnionych kodem oraz pozostałych wymagań związanych z *BDD*.

3.1.1 Usystematyzowanie opisu

Opis będzie powstawał w języku, który będzie zbliżony do naturalnego, stąd wynika konieczność jego usystematyzowania pod kątem struktury, a przede wszystkim treści, jaką ma ze sobą nieść.

Określenie, jak ma być budowany tekst, jest bardzo istotne ze względu na późniejszy odbiór i interpretację przez osoby niezaznajomione ze specyfiką i procesami danego klienta. Usystematyzowanie polega na ścisłym zdefiniowaniu, w jakiej kolejności są opisywane kolejne elementy wymagań i w jakie informacje muszą się w tym opisie znaleźć. Dobrym przykładem jest wykorzystanie zdań zaczynających się od `Jako, Chcę, W celu.` Ten prosty sposób narzuca pewną skondensowaną formę i zapewnia, że pokryty będzie stały zakres informacji.

3.1.2 Język

Język musi być prosty, zrozumiały dla przeciętnej osoby z umiejętnościami analitycznymi, ma pozwalać na elastyczny i pełny zapis historii (zbioru scenariuszy testowych) określającej funkcjonalność.

Należy zwrócić uwagę, że język ten będzie funkcjonował w wielu obszarach procesu wytwórczego oprogramowania. Przede wszystkim opis ten będzie tworzony przez analityka, który z natury rzeczy często nie jest specjalistą w dziedzinie programowania. Tak stworzone opisy powinien weryfikować klient, ponieważ to on najczęściej ma szeroki zakres wiedzy merytorycznej. Równie dobrze, za weryfikację może być odpowiedzialny inżynier, wewnątrz firmy, który również ma ograniczoną znajomość języka programowania. Weryfikacja może też zakończyć się dodaniem nowych testów. Wreszcie opis ten będzie funkcjonował wśród programistów, którzy operują różnymi technologiami. Istotne jest, by każda grupa osób była w stanie nie tylko zrozumieć opis, ale żeby każdy z nich mógł sam modyfikować i tworzyć własne opisy. Dlatego język musi być możliwie prosty z ograniczoną składnią i minimalną semantyką. Wszystkie użyte słowa języka powinny być bliskie lub identyczne z ich znaczeniem w języku naturalnym.

3.1.3 Edytor

Zapewni łatwość tworzenia opisów w środowisku programistycznym za pomocą kontroli składni, podpowiadaniu, informowaniu o błędach i wskazywaniu potencjalnych przyczyn

Kluczowym i poruszanym wcześniej zagadnieniem jest łatwość użycia narzędzia i jakość obsługi (*user experience)*, które będą miały olbrzymi wpływ na to, czy dane narzędzie, a razem z nim cała metodologia *BDD*, zostanie wprowadzona do projektu. Nie trudno zauważyć, że po dostarczeniu nawet prostego języka, może się okazać, że jego użycie w formie czysto tekstowej spowoduje masę błędów, o których użytkownik nie będzie wiedział, a które doprowadzą do tego, że jego praca będzie musiała być poprawiana przez niego, lub osoby pracujące z jego. Zatem wsparcie ze strony narzędzia do edycji musi być możliwie szerokie, a przy tym szybkie i nienachalne.

3.1.4 Analiza opisu funkcjonalności

Umożliwi przetworzenie tekstu zawierającego historię na gotowe twory programistyczne, które przyspieszą proces implementacji testów

Najistotniejszym ogniwem w procesie wytwarzania oprogramowania są programiści, których praca często w projektach informatycznych zajmuje najwięcej czasu. Ich rola w projekcie nie jest do przecenienia, ale zarazem ich zdanie na temat używanych metodyk i narzędzi jest bardzo istotne. Koniecznością jest, by wdrażane narzędzie niosło dla nich wartość dodaną, bo inaczej nigdy się do niego nie przekonają. Dlatego w narzędzie musi z opisu tekstowego generować uporządkowany i łatwy do użycia kod. Narzędzie powinno wykonywać możliwe dużo akcji w taki sposób, aby programista nie musiał szukać odpowiednich funkcji, pamiętać o dodatkowych opcjach, czy spędzać długich godzin nad skonfigurowaniem środowiska tak, by pewne rzeczy działy się automatycznie. Narzędzie powinno samo generować odpowiednie interfejsy programistyczne, dbać o ich aktualizację przy edycji opisu i zapewniać, że wszystkie warunki opisane w aplikacji będą wyraźnie zaznaczone w kodzie, jako takie do implementacji.

3.1.5 Spójność z BDD

Narzędzie powstałe zgodnie z koncepcją będzie wspierało BDD w jak najszerszym zakresie, co oznacza, że poza elementami już wyszczególnionymi należy wskazać poniższe:

- Oczywistym wymaganiem jest, żeby narzędzie było zgodne z założeniami metodologii BDD i wspierało ją w jak najszerszym zakresie. Poza elementami wykazanymi w poprzednich punktach wskazać należy na:
- Opis funkcjonalności ma zgodnie z pierwszym słowem w *Behavioral Driven Development* określać jej zachowanie, a nie sposób, w jaki ma działać. Ten element musi znaleźć swoje odbicie, w tym jak język będzie usystematyzowany i jakich słów kluczowych będzie używał.
- Nazwy metod tworzonych na podstawie opisu funkcjonalności muszą opisywać zadanie, dzięki temu, kiedy dany test kończy się niepowodzeniem, od razu widać, co poszło nie tak.
- Historia powinna być dokumentacją funkcjonalności. Stąd potrzeba przygotowania tak języka, aby umożliwiał dodatkowe komentarze oraz dodatkowe opisu pozwalające na grupowanie, lub tagownie poszczególnych funkcjonalności lub też samych scenariuszy.

3.2. Założenia funkcjonalne

Poniższe punkty opisują zakres funkcji, jakie winny się znaleźć w narzędziu zgodnie z przyjętą koncepcją rozwiązania.

3.2.1 Opis funkcjonalności

Każdy specyfikowana funkcjonalność przez klienta lub analityka musi być odpowiednio opisana. We wcześniejszych fragmentach zostało wskazane, że najlepszym opisem dla programisty są przypadki użycia, ale nie tylko. Równie istotnym jest opisanie czynności (funkcji), roli (użytkownika) i celu tak, by powstał kontekst. Zatem udany opis powinien zawierać następujące elementy składowe:

1. Nazwę funkcjonalności (jedno lub co najwyżej kilka słów opisujących)
2. Kontekst funkcjonalności poprzez
    a. Rolę (kto będzie używał danej funkcjonalności)
    b. Czynność (jaka akcja jest do podjęcia)
    c. Cel działania
3. Scenariusze testowe, które składać się będą na całą historię, gdzie każdy będzie zawierał
    a. Nazwę konkretnego przypadku (scenariusza) testowego
    b. Warunki początkowe, jakie musi test zapewnić
    c. Akcję do podjęcia celem uzyskania rezultatu
    d. Określenie warunków walidujących poprawne działanie

Każda funkcjonalność może być opisana jednym takim opisem, lub wieloma. Wynika to z tego, że liczba scenariuszy testowych może być tak duża, że będzie się je dało pogrupować w związku z kontekstem systemu, czyli warunkami, w jakich dany zestaw testów się odbywa. Innym kryterium podziału może być część narracyjna, gdzie różne role korzystają z tej samej funkcji w różnym celu. Grupy testów mogą się też odwoływać do różnych typów testów od jednostkowych, przez integracyjne do obciążeniowych, czy akceptacyjnych. Jednocześnie grupowanie w mniejsze pliki zwiększa przejrzystość i czytelność opisów, a zarazem powoduje, że poszczególne scenariusze lepiej korespondują z nazwą funkcjonalności. Każdy z opisów jest tworzony w osobnym pliku o innej nazwie.

Opis kontekstu będzie składał się z prostych konstrukcji zdaniowych, jest to sposób wymuszenia istotnych informacji, bez wchodzenia w szczegóły implementacyjne. Osoba piszące wymagania skupia się tylko na nich, a nie wchodzi w techniki realizacji, co pozwala na późniejszym etapie na proponowanie rozwiązań alternatywnych.

Po określeniu podstawowych informacji, czyli przestrzeni nazw i części narracyjnej wprowadzającej kontekst, przyszedł czas na scenariusze testowe, albo raczej warunki akceptacji działania określonego kawałka kodu. Scenariusz testowy jest najbardziej złożonym elementem w całym opisie i jednocześnie jest najważniejszym elementem, ponieważ definiuje kolejne kroki testu. Scenariusz określa w kolejności warunki początkowe, akację i warunki walidacyjne. Każdy z tych elementów niesie informację, co należy zaimplementować w kodzie testu, aby ten rzeczywiście weryfikował poprawność działania funkcjonalności.

3.2.2 Wielojęzyczność

Projekty informatyczne prowadzone są w większości, jak nie we wszystkich krajach świata, przez co język, w jakim będzie wymagana dokumentacja może być dowolny. Nawet zakładając Polskę, jako kraj, w jakim będzie prowadzony projekt, można znaleźć firmy, gdzie dokumentacja będzie tworzona w języku polskim, jak i takie, gdzie będzie ona w języku angielskim. Możliwość „lokalizacji” jest w

obliczu powyższego niezbędna. Wsparcie powinno być na poziomie samego opisu funkcjonalności, jak i na poziomie komunikatów i opcji narzędzia.

3.2.3 Opis stanu początkowego

Zakładając, że jedna funkcjonalność może być opisana w kilku plikach, gdzie te ostatnie będą zawierały pogrupowane scenariusze testowe tak, by miały one jak największą część wspólną okaże się, że warunki początkowe testowanego obiektu będą wspólne dla wszystkich scenariuszy w pliku. Z tego względu konieczną jest implementacja w narzędziu możliwości opisania warunków początkowych raz dla całego zestawu scenariuszy testowych. W ten sposób uniknie się powielania tych samych sekwencji tekstu, opis będzie bardziej zwięzły i treściwy, a co za tym idzie, jego interpretacja i przyswojenie będą dokładniejsze.

3.2.4 Dane i zmienne w scenariuszach testowych

Testy jednostkowe mogą polegać na tym, że jeden test jest sprawdzany z wieloma różnymi kombinacjami danych wejściowych i odpowiadających im rezultatów. Z drugiej strony założenia TDD i BDD mówią jasno, że za wszelką cenę należy unikać korzystania z dodatkowych źródeł danych, ze względu na konieczność uruchamiania testów w różnych miejscach i co za tym idzie rozdzielenie testu od danych za pomocą, których funkcjonalność jest testowana.

Powyższy akapit jednoznacznie wskazuje konieczność zaimplementowania w narzędziu łatwego przekazywania oznaczania danych, jako zmienne, a jednocześnie umożliwiać wskazywanie odpowiednich kombinacji zmiennych wejściowych i wyjściowych używanych w ramach testów. Zatem musi istnieć taka konstrukcja w języku, która pozwoli zaznaczyć w tekście słowa, pod które będą podstawiane różne wartości. Przy czym typy danych powinny być maksymalnie proste i obejmować `string`, `double` i `integer`. Musi też być prosty sposób tworzenia tabel z wartościami, gdzie pierwszy wiersz będzie zawierał nazwy zmiennych użytych wcześniej w scenariuszu, a kolejne wiersze poszczególne wartości. Dzięki temu programista implementując kod będzie mógł skorzystać pętli, a w niej podstawiać pod funkcję przyjmującą zestaw parametrów wartości z kolejnych wierszy wspomnianej tablicy.

3.2.5 Edytor

Narzędzie powinno wspierać użytkownika podczas edycji plików opisujących funkcjonalność i poszczególnych scenariuszy testowych. Odpowiednimi środkami do realizacji tego zadania będą:

- Kolorowanie składni tak, by od razu widać było typy i funkcję poszczególnych elementów tekstu. Musi być możliwość akcentowania kolorem każdego ze słów kluczowych. Odpowiedni kolor powinny też przyjmować zmienne, wartości i inne elementy, które będą miały przełożenie na automatycznie generowany kod.
- Podpowiadanie kontekstowe, które w zależności od tego, co użytkownik w danym momencie pisze, wskazuje pasujące do kontekstu elementy składni oraz żeby pozwalało na dokończenie słowa, bez wprowadzania go w całości (auto-uzupełnianie).
- Szybka informacja podawana użytkownikowi, kiedy chce dowiedzieć się więcej w danej chwili na temat elementu języka zastosowanego w kodzie. W ten sposób użytkownik nie musi sięgać do zewnętrznej w stosunku do aplikacji dokumentacji, a posiadając podstawową wiedzę, resztę

może uzupełnić na bieżąco. Takie podpowiedzi powinny się pojawiać po najechaniu kursorem w edytorze tekstu nad konkretne słowo.

- Podkreślanie fragmentów tekstu, które mogą powodować błędy kompilacji, czyli nie są poprawne. To narzędzie powinno walidować maksymalną liczbę kryteriów, które brane są pod uwagę podczas walidacji treści na etapie generowania pliku wyjściowego. Takie podejście pozwala szybko poprawić składnię i jednocześnie uniknąć analizowania okienka z błędami.

3.2.6 Generator

Końcowym etapem tworzenia opisu funkcjonalności jest jego przełożenie na konkretne konstrukcje programistyczne, które spowodują minimalny nakład pracy programisty w związku z implementacją testów i jednocześnie zminimalizują ryzyko pominięcia istotnych części scenariuszy. Generator musi

- walidować poprawność składni,
- wskazywać miejsca, gdzie są błędy
- tworzyć pliki wyjściowe w wybranym języku programowania w przypadku, kiedy walidacja tekstu zakończyła się poprawnie.

Pliki uzyskane w wyniku użycia generatora powinny być tak skonstruowane, aby można było ich użyć w ramach różnych typów testów (jednostkowe, akceptacyjne, integracyjne). Jest to spowodowane tym, że założenia testów, czyli scenariusze często nie ulegają zmianie dla różnych typów testów, jednak ich implementacja jest różna, choćby przy testach integracyjnych nie należy już zaślepiać funkcjonalności, które nie są zaimplementowane przez testowany kod.

W przypadku języka C# wspierającego interfejsy naturalnym wyborem jest właśnie wykorzystanie do tego definicji `interface`.

3.2.7 Organizacja plików projektu w środowisku programistycznym

Ważne jest, by pliki w projekcie były odpowiednio uporządkowane i to na dwóch płaszczyznach. Organizacja powinna być poziomie katalogów i pliki powinny być odpowiednio przechowywane, ale równie ważne jest by z poziomu samego narzędzia programistycznego pliki były odpowiednio uporządkowane, ale nie muszą być koniecznie ułożone w ten sam sposób co w katalogach.

Narzędzie podczas generowania plików, powinno decydować o ich nazwach, a dbając o brak konfliktów nazw umieszczać je w podkatalogach o takiej samej nazwie, co plik funkcjonalności. Dodatkowo, aby nie mnożyć bytów w oknie projektu w narzędziu programistycznym powinna być wykorzystana możliwość „podpięcia” tych plików pod plik źródłowy z opisem funkcjonalności. W ten sposób mamy na podstawowym poziomie pliki opisujące funkcjonalności, a pod nimi w dopiero ich „liście” w formie plików z kodem. Z kolei struktura na dysku będzie nieco odmienna, ponieważ dla każdej funkcjonalności będzie katalog o takiej samej nawie, a w nim dopiero wygenerowane pliki.

3.3. Założenia niefunkcjonalne (jakościowe)

Poniżej zostały zaprezentowane wymagania, które nie dotyczą poszczególnych funkcji narzędzie, jednak również powinny być wzięte pod uwagę.

3.3.1 Wybór środowiska

Na potrzeby prototypu wykorzystane musi być *Visual Studio 2012* firmy *Microsoft*. Pełna implementacja koncepcji może być przeprowadzona dla więcej niż jednego środowiska.

3.3.2 Cechy użytkowe edytora

Edycja plików musi być płynna, użytkownik nie może „czekać" z wpisywaniem treści. Jednocześnie funkcje kolorowania, podpowiadania i wskazywania „wątpliwego" pod kątem poprawności kodu powinny działać praktycznie w czasie rzeczywistym z niewielkimi opóźnieniami pozwalającymi użytkownikowi płynną pracę.

3.3.3 Prostota

W związku z tym, że narzędzie będzie musiało zostać wdrożone zarówno wśród programistów, jak i osób niekoniecznie związanych z samym programowaniem, składnia i sposób użycia muszą być łatwe i intuicyjne.

3.4. Podsumowanie

Opisana w tym rozdziale koncepcja obejmuje swoim zakresem zestaw narzędzi, który będzie rozszerzał funkcjonalność środowiska programistycznego, tak by wprowadzanie metodologii *BDD* było łatwe do uzasadnienia na każdym poziomie od analityków, przez programistów, do klienta i zarządu wykonawcy. Cel zostanie osiągnięty dzięki takim rozwiązania, jak spójność narzędzi, realizacja postulatów BDD, jakość obsługi.

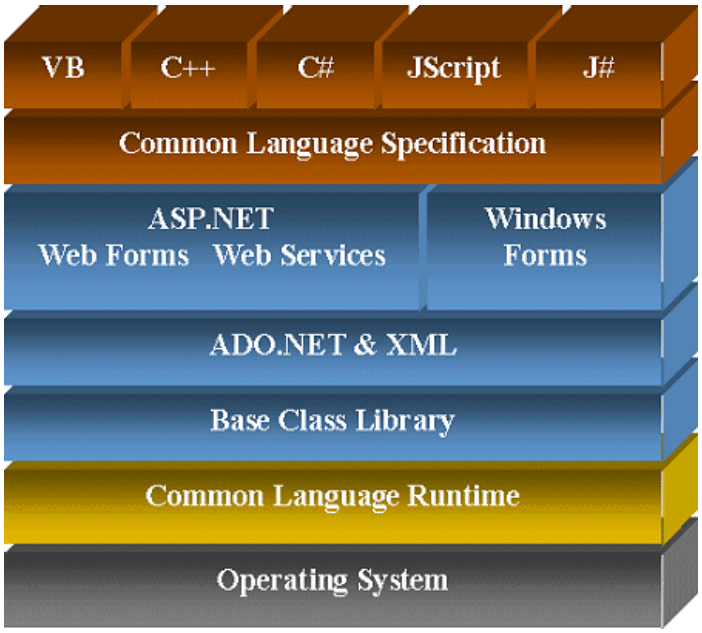
4. Zastosowane narzędzia

W tym rozdziale omówione zostały zastosowane technologie i narzędzia użyte do budowy prototypu.

4.1. .NET

Platforma *.NET* obejmuje zarządzalne środowisko uruchomieniowe *CLR* (*Common Language Runtime*) oraz szeroki zestaw bibliotek pozwalających na budowanie aplikacji uruchamianych w systemach operacyjnych firmy *Microsoft*. Sama platforma *.NET* jest oderwana od języka programowania i można tworzyć na nią oprogramowanie z wykorzystaniem taki języków jak C++/CLI, C#, F#, J#, Delphi 8 dla .NET, Visual Basic .NET, gdzie w każdym z nich można się odwołać do kodu napisanego w jednym z pozostałych.

Założeniem *.NET jest, by* tworzenie aplikacji odbywało się w oparciu o języki obiektowe i zapewniało przenośność kodu pomiędzy różnymi platformami sprzętowymi i pomiędzy różnymi systemami operacyjnymi, choć już w bardziej ograniczonym stopniu.



Rysunek 3 - Architektura .NET

Efekt został osiągnięty przez kompilowanie kodu aplikacji zamiast do kodu maszynowego, do kodu pośredniego (*Microsoft Intermediate Language* — w skrócie *MSIL*). Kod *MSIL* może być traktowany jako gotowa aplikacja i w ten sposób rozpowszechniany. Dopiero przy pierwszym uruchomieniu na danej maszynie *CLR* wywołuje kompilacje aplikacji w postaci *MSIL* do kodu maszynowego, jego optymalizację dla danego procesora i odpowiada za jego uruchomienie. Przykład w oparciu o język C# zaprezentowany jest na Rysunek 4.

Platforma *.NET* dostarcza następujące elementy:

- Zarządzanie pamięcią, które zwalnia programistę z pamiętania o zwalnianiu zasobów, gdyż CLR zrobi to za niego
- Wspólne typy podstawowe, które dla wszystkich wspieranych języków są takie same przez co nie ma braku kompatybilności między nimi.
- Zbiór bibliotek usprawniających pracę programistów
- Zbiór technologii pozwalający na programowanie różnych typów aplikacji od desktopowych, przez serwisy webowe, do aplikacji rozproszonych
- Możliwość wykorzystania w pisanym kodzie dowolnego innego kodu napisanego w jednym z wspieranych przez *.NET* języków programowania
- Możliwość wykorzystywania odpowiedniej wersji *.NET,* ale i możliwość instalowania różnych wersji jednocześnie tak, by aplikacje napisane pod konkretne wydania mogły działać jednocześnie w tych wydaniach *.NET* , na które były pisane
- Wsparcie dla pisania kodu na różne systemy operacyjne [MicrosoftNET].

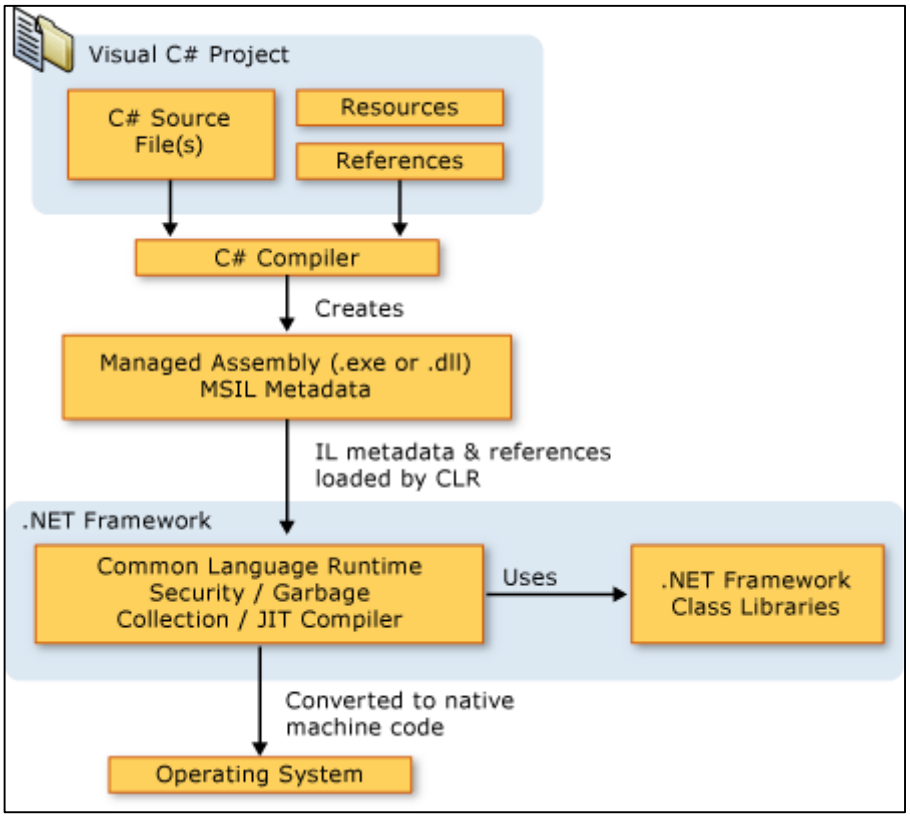
4.2. Język programowania C#

Obiektowy język programowania C# mocno rozwijany przez firmę *Microsoft* przeznaczony do budowania aplikacji w środowisku *.NET* [MicrosoftC#].

Główne cechy C#:

- Nowoczesny język programowania o przeznaczeniu ogólnym
- Zorientowany obiektowo
- Obsługujący właściwości klas i zdarzenia (*Events)*
- Bazujący na koncepcji programu, jako zbiorze klas z jednym punktem wejściowym
- Brak konieczności zarządzania pamięcią, zapewnia *.NET*
- Proste w użyciu typy ogólne –*generics*
- Obsługa dla *LINQ* i wyrażeń *lambda*
- Wszystkie klasy dziedziczą po `System.Object`, co zapewnia metody wewnętrzne również w typach prostych
- Atrybuty pozwalające na przekazywanie metadanych o typach w czasie wykonywania kodu
- Refleksja pozwalająca na analizowanie kodu podczas jego wykonywania
- Dynamiczne tworzenie kodu podczas wykonywania programu i dołączanie go do uruchomionego kodu
- Bogate wsparcie do tworzenia graficznych interfejsów użytkownika poprzez zbiór bibliotek
- Silna integracja z systemem Windows

Język C# jako język wysokopoziomowy szeroko czerpie z języków *C* i C++, ale największe podobieństwo wykazuje do języka *JAVA.*



Rysunek 4 - Zasada działania kodu C# w oparciu o platformę .NET
Źródło: [MicrosoftC#]

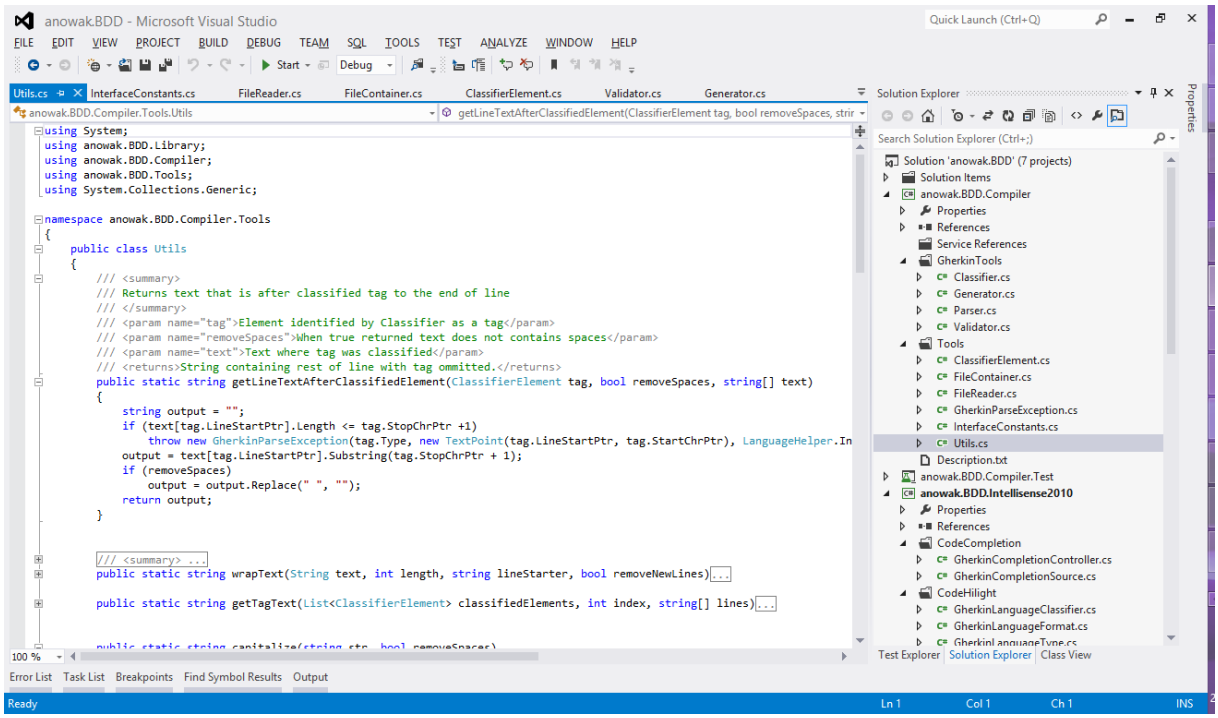
4.3. Microsoft Visual Studio

Oprogramowanie *Visual Studio* jest obecnie jednym z najbardziej zaawansowanych edytorów pozwalającym na pracę z różnego rodzaju językami programowania, ale przede wszystkim jest to zestaw narzędzi pozwalających na budowanie aplikacji na platformy z rodziny *Micrososft*:

- *Windows*
- *Widnows CE (Ebedded)*
- *Windows Phone*
- *.NET*
- *Silverlight*
- *XBOX*

Wspomniane środowisko programistyczne jest często stawiane za wzór dla innych ze względu na szybkość i łatwość pracy z interfejsem graficznym, dlatego zostało ono wybrane, jako miejsce uruchomienia prototypu w ramach pracy magisterskiej

Prototyp został przygotowany dla wersji 2012, która była najnowszą stabilną wersją narzędzia podczas rozpoczynania prac programistycznych. W tej chwili jest już dostępna nowsza wersja oznaczona jako 2013 [MicrosoftVS].



Rysunek 5 - Przykładowy widok narzędzia *Visual Studio*

Visual Studio było też jedynym środowiskiem wykorzystywanym do edycji kodu w ramach realizacji niniejszej pracy dyplomowej, a to dzięki rozbudowanym funkcjom, do których należą:

- Edytor kodu wraz z Intellisense
- Debugger, który pracuje na poziomie kodu źródłowego jaki na poziomie architektury sprzętowej
- Narzędzie typu *Designer* do budowania GUI
- Wsparcie do budowania aplikacji WWW w oparciu o kod C#, serwer ISS i silnik Razor
- Zarządzanie źródłami danych
- Narzędzie do projektowania schematów baz danych
- Wbudowane wsparcie dla testów jednostkowych

4.3.1 Intellisense

Intellisense jest silnikiem w obrębie *Microsoft Visual Studio*, który pozwala na stosowanie mechanizmów ułatwiających tworzenie aplikacji przez m.in.:

- *List Members* - Listy elementów klasy lub obiektu jak metody i parametry
- *Parameter Info* - Informacje dotyczące parametrów przekazywanych metodom

- *Quick Info* – informacja dotycząca elementu kodu
- *Code Colorization* – podświetlanie kodu w kolorach zwiększając czytelność
- *Complete Word* – podpowiadanie elementów składni

Narzędzie *Intellisense* pozwala tez na budowanie rozszerzeń dla *Microsoft Visual Studio*, które mogą pozwolić na stosowanie wyżej wymienionych ułatwień dla dowolnego języka programistycznego. Obecnie natywnie wspierane są języki C++, C#, J#, Visual Basic, XML, HTML, XSLT.

Intellisense wspiera poza podstawowymi funkcjami powyżej specyficznie niektóre języki:

- C# - m.in. automatyczne generowanie kodu i wskazywanie najczęściej używanych członków klasy
- Dla Visaul Basic, JScript, T-SQL – m.in wskazówki dotyczące składni, zwijanie zawartości, gotowe kawałki kodu (*snippets)*, inne. [MicrosoftIS]

4.3.2 Visual Studio SDK

Zestaw bibliotek pozwalający na rozszerzanie środowiska programistycznego o dodatkowe możliwości poprzez [MicrosoftEVS].:

- Dodawanie komend, okienek
- Rozszerzania edytora
- Możliwość napisania rozszerzenia wspierającego nowy język
- Wsparcie dla rozszerzeń umożliwiających korzystanie z nowych źródeł danych
- Własne schematy projektów
- Możliwość użycia różnych narzędzie do zarządzania wersjami kodu
- Opcje debuggera, lub możliwość napisania swojego

4.4. Managed Extensibility Framework MEF

Zestaw narzędzi programistycznych pozwalający na łatwe i proste opieranie kodu aplikacji lub narzędzi na rozszerzeniach bez konieczności ich konfiguracji i używania sztywnych zależności. Pozwala programistom tworzącym rozszerzenia na łatwe ich opakowywanie i wykorzystanie w aplikacji, lub pomiędzy aplikacjami

MEF pozwala na tworzenie aplikacji, które będzie następnie można rozszerzać bez konieczności ponownej kompilacji. Biblioteka przede wszystkim pozwala na odejście od sztywnej rejestracji rozszerzeń na rzecz wykrywania ich poprzez kompozycję. W prosty sposób definiowane są elementy, które kod pobiera (importuje) i jakie udostępnia (eksportuje). Na tej podstawie jedna strona zasila danymi wskazane w drugiej stronie za pomocą `import` zmienne. Następnie pobiera z rozszerzenia na

tej samej zasadzie dane z atrybutem `export` i wypełnia nimi swoje zmienne oznaczone jako `import`. W takim przypadku, po określeniu odpowiednich interfejsów bardzo łatwo jest rozszerzyć pracę aplikacji o kawałek kodu.

Zasada działania *MEF* opiera się na popycie i podaży poszczególnych stron (części obiektu w postaci klasy, obiektu, właściwości). Obie strony, które mają ze sobą pracować, muszą być odnajdywalne i każda ze stron musi zaspokajać popyt drugiej na podstawie umowy najczęściej zawarte w formie interfejsu, który implementują obie strony. *MEF* zadba o dynamiczne przeszukiwanie i odnajdywanie stron w znanej lokalizacji, jeżeli tylko zajdzie taka potrzeba. W przypadku odnalezienia pasującego kodu silnik kompozycji dołączy go do tego już wykonywanego oczywiście, jeżeli obie strony spełnią wymagania umowy.

MEF jest dostępny w pakietach *.NET* od wersji 4.0 i nie jest zależny od innych technologii, tzn. można go użyć w dowolnej aplikacji niezależnie czy jest to program linii poleceń, aplikacja graficzna, czy rozproszona [Mef].

5. Realizacja prototypu

Ten rozdział został poświęcony prototypowi, który powstał w ramach niniejszej pracy dyplomowej na bazie narzędzi z poprzedniego rozdziału i koncepcji rozwiązania z rozdziału 3.

5.1. Zastosowany język

Głównym elementem narzędzia powstającego w ramach koncepcji z punktu widzenia użytkownika jest edytor i to co będzie za jego pomocą edytowane. Nawet najlepszy edytor nie będzie w stanie sprawić, by źle dobrany, lub po prostu nieintuicyjny język był wygodny i chętnie używany. Ten podrozdział został poświęcony charakterystyce języka, jaki został wykorzystany w ramach prototypu.

5.1.1 Słowa kluczowe - semantyka

Na potrzeby prototypu zostały użyte elementy języka *Gherkin*, a przede wszystkim zostały wykorzystane słowa kluczowe, od których rozpoczyna się kolejne linie

- *Feature*: (Funkcjonalność) – po słowie kluczowym występuje tekst opisujący nazwę historii (funkcjonlaności) aż do końca linii.
- *As* (Jako) – określa rolę jaka korzysta z danej funkcjonalności
- *I Want* (Chcę) – określa pożądany efekt
- *In Order* (W Celu) - określa szerszy kontekst wykonywanej czynności w *I Want*
- *Background*: (Tło) – poprzedza blok warunków *Given, And, But* opisujących początkowy stan systemu dla wszystkich *Scenario* w pliku
- *Scenario* (Scenariusz) – po słowie kluczowym występuje tekst opisujący nazwę konkretnego scenariusza, a w kolejnych liniach warunki *Given, When, Then, And, But*.
- *Given* (Zakładając) – słowo rozpoczyna linię zawierającą warunek, który musi być spełniony przed rozpoczęciem akcji *When*
- *When* (Kiedy) – słowo rozpoczyna linię zawierającą akcję do wykonania, żeby przejść od stanu początkowego, do stanu walidowanego
- *Then* (Wtedy) – słowo rozpoczyna linię zawierającą warunek walidujący poprawność wykonania scenariusza testowego
- *And* (Oraz) – słowo rozpoczyna linię zawierającą dodatkowe warunki. And przybiera znaczenie pierwszego poprzedzającego go słowa z zakresu *Given, When, Then*
- *But* (Ale) – słowo rozpoczyna linię zawierającą dodatkowe warunki. And przybiera znaczenie pierwszego poprzedzającego go słowa z zakresu *Given, When, Then*
- *Scenario Outline*: (Zarys Scenariusza) – po tych słowach kluczowych występuje tekst opisujący nazwę konkretnego scenariusza, identycznie jak w *Scenario*, z tą różnicą, że można w kolejnych liniach stosować zmienne zamiast wartości, których przykłady muszą być podane w tabeli za *Examples*.

- *Examples*: (Przykłady) – słowo kluczowe, które występuje jedynie dla Zarysu Scenariusza już za wszystkimi warunkami (Givem, When, Then, And, But) i poprzedza tabelę, gdzie nagłówek definiuje nazwy zmiennych użytych w warunkach, a następne wiersze określają poszczególne zestawy danych wejściowych i wyjściowych.

Każda linia w tekście poza komentarzami i tagami, o których mowa będzie dalej, rozpoczyna się od słowa kluczowego, co nadaje jej odpowiednie znaczenie i na podstawie tego słowa interpretowana jest zawartość linii. Słowo kluczowe może składać się z kilku słów w rozumieniu języka naturalnego.

5.1.2 Składnia

Kolejność słów kluczowych nie jest obojętna tak samo, jak część słów wyznacza bloki w sensie organizacji tekstu. Dla ułatwienia warto stosować wcięcia tekstu, choć ze względu na łatwość użycia nie są one brane pod uwagę.

- Każdy plik musi się zaczynać od linii zawierającej słowo `Feature`: (z pominięciem komentarzy i tagów, które mogą poprzedzać *słowa* `Feature`, `Scneario` i `Scenario Outline`).
- Następnie może wystąpić blok linii związanych z narracją w postaci `As`, `I Want`, `In Order`. Blok ten jest bardzo istotny, aby szerzej zrozumieć kontekst scenariuszy testowych przedstawionych w dalszej części tekstu, jednak ze względu na jego narracyjny charakter, nie jest on obowiązkowy i może zostać pominięty.

Listing 9 - Przykład opisu funkcjonalności.

```
Feature: Create Profile Form Validation
       As System administrator
       I want to be sure that input data provided by user in form is valid
       In order to have proper data in db and be able to use it for invoicing
```

- Również nieobowiązkowy blok dotyczy stanu początkowego systemu wspólnego dla wszystkich scenariuszy, czyli blok zaczynający się od `Background`. W tym bloku mogą wystąpić jedynie linie zaczynające się od `Given`, `And`, `But`, ponieważ na tym etapie, żadne czynności nie są wykonywane, a walidacja nie ma prawa się tu wydarzyć.

Listing 10 - Przykład opisu tła

```
Feature: Create Profile Form Validation
       As System administrator
       I want to be sure that input data provided by user in form is valid
       In order to have proper data in db and be able to use it for invoicing

       Background:
       Given User is using web site at "www.exampleSite.com/CreateProfile"
       And is using web browser localized for "pl-PL"
       But his browser does not support HTML5
```

- Wreszcie czas na bloki zawierające poszczególne scenariusze lub ich zarysy. Zatem mogą występować wymiennie bloki rozpoczęte przez `Scenario` lub `Scenario Outline`. Jednym warunkiem jest to, że w całym pliku musi wystąpić, choć jeden blok związany z opisem scenariuszy

- `Scenario` – Blok zawiera linie opisujące warunki rozpoczynające się od `Given`, `When`, `Then` rozszerzone o `And` i `But`, które mogę występować dowolnie, ale nie przed `Given`. Kolejność jest ważna, ponieważ `Then` nie może wystąpić przed `When`, a `When` przed `Given`, ale każde z nich musi wystąpić przynajmniej raz.

Listing 11 - Dwa identyczne opisy z użyciem różnych słów kluczowych

```
Scenario: Attempt to pay with debit card
       Given I have $100 in my bank account
       But my card is outdated
       When I want to pay $50
       Then my request should be denied
       And I should be told that my card is not valid

Scenario: Attempt to pay with debit card
       Given I have $100 in my bank account
       Given my card is outdated
       When I want to pay $50
       Then my request should be denied
       Then I should be told that my card is not valid
```

- `Scenario Outline` – Blok musi spełniać te same warunki jak `Scenario` z dokładnością do `Example`, które musi wystąpić po wszystkich liniach opisujących warunki. `Example` mówi o tym, że w następnych liniach będą wiersze tabeli, zawierające konkretne wartości dla użytych zmiennych w scenariuszu testowym.

Listing 12 - Przykład *Scenario Outline* z listą wartości dla użytych zmiennych

```
Scenario Outline: Attempt to pay with debit card
       Given I have <intial_amount> in my bank account
       And my card is valid
       When I want to pay <payment>
       Then my request should be <accepted>
       Then I should have on my bank account <decreased_amount>

       Examples:
       | initial_amount | payment | accepted | decreased_amount |
       | 123            | 250     | false    | 123              |
       | 123            | 120     | true     | 3                |
       | 150            | 250     | false    | 150              |
       | 149            | 149     | false    | 0                |
```

5.1.3 Pozostałe elementy

Zastosowany język jest bardzo elastyczny i pozwala dość szybko nauczyć się reguł nim rządzących. Poza omówionymi wyżej słowami kluczowymi definiującymi znaczenie linii są jeszcze dodatkowe elementy:

- Wartość – występuje w liniach opisujących warunki (`Given, When, Then, And, But`)
- `Int` – gdzie jest to liczba całkowita
- `Double` – gdzie jest to liczba z kropką, lub przecinkiem. To zależy od wybranego języka
- `String` – wartość ujęta w cudzysłowie ”to jest wartość zmiennej typu string”

- `PyString` – wartość typu string, obejmujące wiele wierszy. Wartość tego typu musi być rozpoczęta i zakończona linią zawierającą """ (trzy znaki cudzysłowia). Nazwa pochodzi z miejsca zaczerpnięcia takiej składni, czyli z języka programowania *Python*.

Listing 13 - Przykładu różnych typów wartości

```
Scenario: Filling the form with correct data
      Given I've inserted "anowak" username into username field
      And I've put that I am 24 years old
      And I'm 1,96 height in meters
      When I press submit
      Then I should see info that
      """
      anowak you have submitted following indo
      your age is 24
      your height is 1,96m
      """
```

- Nazwy wartości – są to słowa opisujące wartość. Występują one za każdym razem, jako pierwsze po wprowadzonej wartości. Ich użycie jest istotne ze względu na to, że programista będzie w kodzie posługiwał się zmiennymi, dzięki nazwom, zmienne w kodzie będą miały już odpowiednie nazwy.
- Zmienne – są to nazwy parametrów, pod które w trakcie wykonywania testu programista będzie musiał podstawić odpowiednie wartości. Konkretne wartości są za to podane w tabeli poprzedzonej słowem kluczowym `Examples:`. Doskonale widać to na Listing 14.
- Tabela – jest zbiorem wierszy, gdzie kolejne kolumny oddzielone są znakiem `|`. Tabela zawiera dwa rodzaje danych (również widoczne na Listing 14):
- Nagłówek gdzie do kolumn przypisywana jest odpowiednia nazwa zmiennej
- Pozostałe wiersze zawierające przykładowe dane, dla każdej zmiennej w zarysie scenariusza zgodnie z nagłówkiem tabeli.
- Komentarze – są to linie rozpoczynające się od znaku `#`, które pozwalają na wewnętrzne komentarze, lub uwagi dla testerów lub programistów. Komentarze nie powinny w żaden sposób zastępować nazw funkcjonalności lub kolejnych kroków testu.

Listing 14 - Przykładowe zastosowanie komentarzy, jako informacji wewnętrznej

```
# Scenariusz do sprawdzenia z biznesem, czy aby na pewno jest OK!
Scenario Outline: Attempt to pay with debit card
       Given I have <intial_amount> in my bank account
       And my card is valid
       When I want to pay <payment>
       Then my request should be <accepted>
       Then I should have on my bank account <decreased_amount>

       # Niezbędne jest dorzucenie testów przez Biznes i programistów
       # trzeba pamiętać o przypadkach szczególnych
       Examples:
       | initial_amount | payment | accepted | decreased_amount |
       | 123       | 250   | false  | 123        |
       | 123       | 120   | true   | 3         |
       | 150       | 250   | false  | 150        |
       | 149       | 149   | false  | 0         |
```

- Tagi – pozwalają one dowolnie oznaczać funkcjonalności i scenariusze za pomocą przyjętych w zespole lub projekcie reguł. Jest to element zwiększający elastyczność rozwiązania, który nie wpływa istotnie na rezultaty generatora.

Listing 15 - Zastosowanie tagów do kategoryzowania elementów opisu

```
@Oficjalny
Feature: Login Credentials Validation
  As user of the xxx system
  I want to be properly validated basing on my credentials
  In order to be sure nobody can see my data not having proper credentials

  @Jednostkowy @Integracyjny
  Scenario: Grant access to anowak with proper credentials
   Given I've put into dialog "anowak" user and my "haslo123" pass
   When I press log me in
   Then access to my data is granted
```

5.1.4 Podsumowanie

Zastosowany język jest przede wszystkim bardzo naturalny i prosty w użyciu. Łatwo jest się nauczyć kilku słów kluczowych i przyjąć, że każda linia musi się od jednego z takich słów zaczynać. Z drugiej strony elastyczność języka pozwala za pomocą opisów przybliżyć każdy scenariusz testowy ujęty w pewne ramy, przez co jest łatwiejszy do zrozumienia. Przyjęta struktura wymusza na twórcy opisów podawanie danych w charakterystyczny sposób, który sprawia, że informacje są bardziej precyzyjne i bardziej zbliżone do tego, jak je będzie implementował programista. Całość jest zgodna ze zwinnymi metodykami, na których bazuje *BDD*.

5.2. Elementy wspomagające edycję

Zgodnie z wymaganiami koncepcji narzędzie musi wspomagać osobę edytującą pliki zawierające opis funkcjonalności wraz ze scenariuszami tak, by minimalizować pomyłki, przyspieszać proces i sprawić, że narzędzie było przyjazne użytkownikowi.

5.2.1 Kolorowanie składni

Kolorowanie składni w środowiskach programistycznych jest dziś standardem, a to ze względu na to, że za pomocą kolorów można tekstu wydobyć mnóstwo informacji bez konieczności czytania i interpretowania słów, co zwiększa czytelność i przejrzystość kodu. Dodatkowym atutem kolorowania jest fakt, że edytor spodziewa się koloru w danym miejscu i jeżeli słowo nie zostanie ono podświetlone, najpewniej wystąpiła w nim literówka. Edytor od razu widzi odstępstwo od normalnego zachowania środowiska i zaczyna weryfikować poprawność, jeszcze przed próbą walidacji w trakcie generowania.

W *Visual Studio 2012* i nowszym kolorowanie składni dla dowolnego typu plików można osiągnąć przez napisanie odpowiedniego rozszerzenia bazując na bibliotekach *MEF*.

W pierwszym kroku trzeba zdefiniować dostawcę usługi kolorowania tekstu, czyli odpowiednie klasy stworzone w projekcie:

Listing 16 - Kod odpowiadający za rejestrację kodu dostarczającego funkcjonalność podświetlania tekstu

```
Export(typeof(ITaggerProvider))]
 [ContentType("Gherkin")]
 [TagType(typeof(ClassificationTag))]
 internal class GherkinClassifierProvider : ITaggerProvider
 {
  [Export]
  [Name("Gherkin")]
  [BaseDefinition("code")]
  internal static ContentTypeDefinition GherkinContentType = null;

  [Export]
  [FileExtension(".feature")]
  [ContentType("Gherkin")]
  internal static FileExtensionToContentTypeDefinition GherkinFileType = null;

  [Import]
  internal IClassificationTypeRegistryService ClassificationTypeRegistry = null;

  [Import]
  internal IBufferTagAggregatorFactoryService aggregatorFactory = null;

  public ITagger<T> CreateTagger<T>(ITextBuffer buffer) where T : ITag
  {
   ITagAggregator<GherkinTokenTag> gherkinTagAggregator =
aggregatorFactory.CreateTagAggregator<GherkinTokenTag>(buffer);

   return new GherkinLanguageClassifier(buffer, gherkinTagAggregator, ClassificationTypeRegistry) as
ITagger<T>;
  }
 }
```

W Listing 16 widać, jak oprogramowanie *Visual Studio*, które będzie używało rozszerzenia, jest informowane za pomocą atrybutu `Export` o typie przekazywanych danych (`ITaggerProvider`), czyli jaki interfejs implementuje rozszerzenie. W następnych liniach pojawia się „`code`” jako definicja bazowa. To oznacza, że rozszerzamy sposób kolorowania przeznaczony jest do plików zawierających kod. W końcu jest informacja, jakich typów plików będzie dotyczyła wtyczka (`.feature`). W kodzie następują też atrybuty Import, które oznaczają, co zostanie dostarczone przez *Visual Studio* wtyczce.

W kolejnym kroku należy opisać formaty, jakie będą klasyfikowane (Listing 17) oraz podać jak mają być wyświetlane (Listing 18**Błąd! Nie można odnaleźć źródła odwołania.**).

Listing 17 - Definicja typu, który będzie klasyfikowany w tekście

```
internal static class OrdinaryClassificationDefinition
  {
    [Export(typeof(ClassificationTypeDefinition))]
    [Name("Feature")]
    internal static ClassificationTypeDefinition feature = null;

…
```

Listing 18 - Określenie formatu podświetlenia dla klasyfikowanego typu

```
  [Export(typeof(EditorFormatDefinition))]
  [ClassificationType(ClassificationTypeNames = "Feature")]
  [Name("Feature")]
  //this should be visible to the end user
  [UserVisible(false)]
  //set the priority to be after the default classifiers
  [Order(Before = Priority.Default)]
  internal sealed class GherkinFeature : ClassificationFormatDefinition
  {
    /// <summary>
    /// Defines the visual format for the classification type
    /// </summary>
    public GherkinFeature()
    {
      //human readable version of the name
      this.DisplayName = "Gherkin Feature";
      this.ForegroundColor = Colors.MediumVioletRed;
    }

  }
```

W kolejnych krokach zaimplementowany został interfejs `ITagger` w klasie, która jest odpowiedzialna za klasyfikację poszczególnych elementów do podświetlenia. Klasa ta za pomocą `GetTags(NormalizedSnapshotSpanCollection spans)` zwraca zakresy tekstu oraz sklasyfikowany typ tekstu np. `Feature`. W tym rozwiązaniu klasyfikator tekstu pracuje na zakresach odpowiadającym liniom, identyfikując słowo kluczowe. W przypadku, kiedy za słowem kluczowym mogą wystąpić inne elementy podlegające podświetleniu (np. wartość, zmienna, lub nazwa zmiennej) wtedy reszta linii jest odpowiednio analizowana. Wyjątkiem od powyższej reguły są niektóre typy, jak `PyString`, tabela, które mogą obejmować sowim zakresem więcej niż jedną linię, dlatego analizę pojedynczej linii poprzedza sprawdzenie, czy należy ona do `PyString` lub do tabeli.

5.2.2 Szybka informacja

Funkcja informowania użytkownika o znaczeniu danego elementu również jest stałym elementem środowisk programistycznych i nie zabrakło jej w prototypie. W przypadku *Visual Studio* sposób dostarczenia edytorowi informacji o wtyczce jest analogiczny jak opisany w punkcie 5.2.1 Kolorowanie

składni. Zmienia się za to informacja przekazywana przez środowisko programistyczne wtyczce, gdyż nie jest to już zakres tekstu a pozycja wskaźnika myszy.

Najistotniejsze różnice w tym wypadku to konieczność implementacji dwóch metod widocznych na Listing 19:

Listing 19 - Metody związane z pozycją kursora przy Szybkiej Informacji (*QuickInfo*)

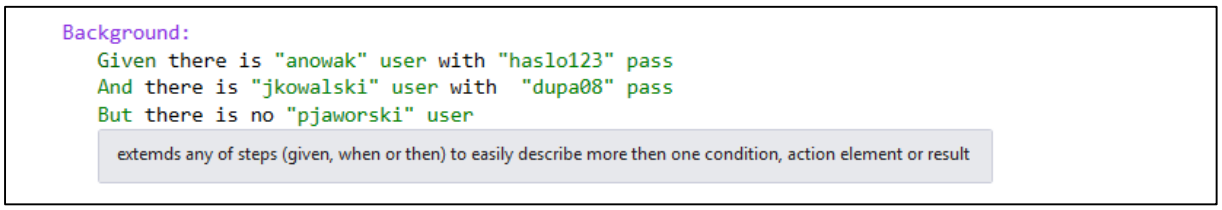
```
 private void OnTextViewMouseHover(object sender, MouseHoverEventArgs e)
 {
   SnapshotPoint? point = this.GetMousePosition(
    new SnapshotPoint(
     _textView.TextSnapshot,
     e.Position));

  if (point != null)
  {
   ITrackingPoint triggerPoint =
    point.Value.Snapshot.CreateTrackingPoint(
     point.Value.Position,
     PointTrackingMode.Positive);

   if // Find the broker for this buffer
 (!_componentContext.QuickInfoBroker.IsQuickInfoActive(_textView))
   {
    _session =
_componentContext.QuickInfoBroker.CreateQuickInfoSession(_textView, triggerPoint,
true);
    _session.Start();
   }
  }
 }

 private SnapshotPoint? GetMousePosition(SnapshotPoint topPosition)
 {
  // Map this point down to the appropriate subject buffer.
  return _textView.BufferGraph.MapDownToFirstMatch (
   topPosition,
   PointTrackingMode.Positive,
   snapshot => _subjectBuffers.Contains(snapshot.TextBuffer),
   PositionAffinity.Predecessor
   );
 }
```

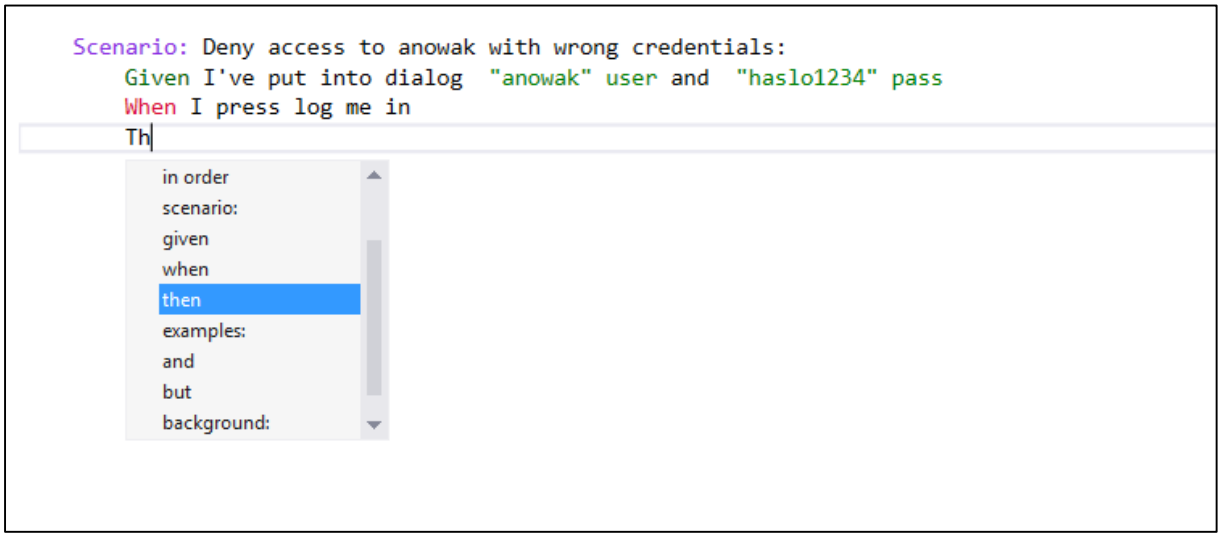
Użycie kodu z Listing 19, razem z prostym klasyfikatorem zwracającym typ wskazywanego słowa i zestawem informacji, jakie należy przekazać w Szybkiej Informacji w zależności od efektu pracy klasyfikatora jest zobrazowane na Rysunek 6.



Rysunek 6 - Przykładowa szybka informacja dla słowa kluczowego `But` pojawiająca się po i krótkim przytrzymaniu

5.2.3 Auto uzupełnianie

Funkcja podczas edycji tekstu podpowiada jak można zakończyć rozpoczęte słowo. Dzięki niej możliwe jest szybsze pisanie tekstu. Zamiast wstukiwać całe słowo, można wprowadzić pierwsze litery, następnie po wskazaniu podpowiedzi w kontekstowym menu, środowisko programistyczne samo dokończy wyraz. Funkcja ta też świetnie się sprawdza, jeżeli użytkownik nie zna jeszcze zbyt dobrze składni i poszukuje odpowiedniego elementu do użycia w tym miejscu. Dzięki tej funkcji zostaną mu wyświetlone wszystkie opcje.



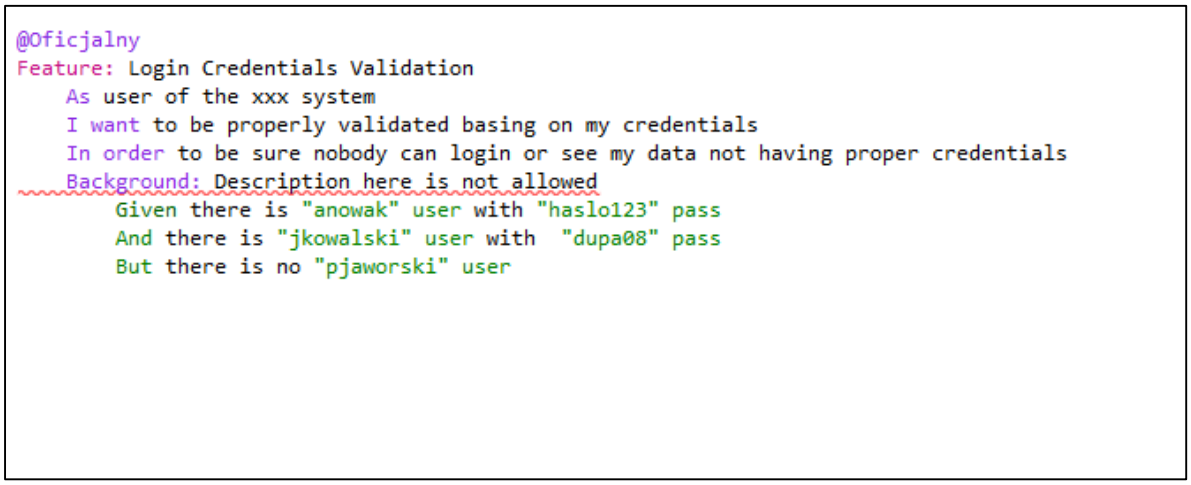
Rysunek 7 - Przykładowe zastosowanie auto-uzupełniania kodu

Realizacja tej funkcjonalności jest zbliżona do pozostałych elementów bazujących na MEF, ale warto zwrócić uwagę na to, kiedy podpowiadanie jest uruchamiane. Uruchomienie następuję poprzez naciśnięcie klawiszy *Ctrl* + *Spacja*, lub poprzez postawienie znaku spacji. Dodatkowo jest założony filtr, żeby podpowiedzi pojawiały się tylko i wyłącznie, jeżeli słowo, które ma być wpisane jest pierwsze w linii. Jest to wynik założeń dotyczących język, gdzie słowa kluczowe występować powinny jako pierwsze w linii. Niestety

5.2.4 Sygnalizowanie błędów

Ostatnią funkcją edytora wspomagającą użytkownika w tworzeniu plików z opisami scenariuszy testowych jest sygnalizacja błędów przez podkreślenie czerwoną, falistą linią obszarów tekstu, których walidacja w kontekście składni języka zakończy się niepowodzeniem. W prototypie funkcja została również udostępniona na zasadzie mechanizmów *MEF* tak, jak poprzednie komponenty. Natomiast walidacja składni dobywa się zupełnie inaczej, mianowicie walidacja idzie krok po kroku analizując każdą linię i sprawdza obecność wymaganych elementów.

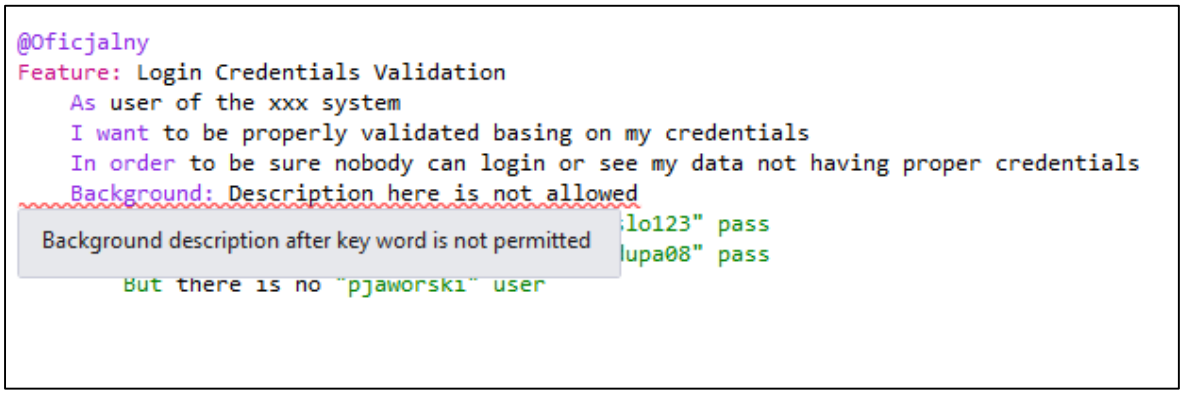
W trakcie walidacji sprawdzane jest, czy poza tagami i komentarzami pierwsza linia zawiera słowo kluczowe `Feature`, następnie czy za nim jest odpowiedni opis. W kolejnych krokach sprawdza obecność już nieobowiązkowego bloku narracji (`As, I Want, In Order`) i w razie, kiedy słowa kluczowe pojawiają się, a brakuje tekstu za nimi sygnalizuje to. Następnym analizowanym blokiem jest `Background`, gdzie nie może wystąpić opis za słowem kluczowym, a kolejne opisy warunków mogą być jedynie typu `Given`, rozszerzone przez użycie `And` i `But`.



Rysunek 8 - Przykładowe zaznaczenie błędu

Następnie w pętli są analizowane wszystkie scenariusze i ich zarysy w bardzo podobny sposób do analizy `Background` z dokładnością do wymaganego opisu scenariusza, większej liczby typów warunków w odpowiedniej ilości (`Given`, `When`, `Then`), możliwości wystąpienia słowa kluczowego `Examples` wraz z tabelą. Weryfikacja tabeli obejmuje liczbę kolumn w kolejnych wierszach.

Sygnalizowanie miejsca nie jest wystarczającym sposobem informowania użytkownika o problemie, bo zdarza się, że ten występuje w innym miejscu niż podkreślenie. Dlatego z narzędzie dostarcza dodatkowych opisów dostępnych w postaci Szybkiej Inforamcji (*QuickInfo*), co jest przedstawione na Rysunek 9.



Rysunek 9 - Informacja o przyczynie błędu

5.3. Generator

Nadrzędnym celem prototypu zrealizowanego w ramach niniejszej pracy dyplomowej jest wspieranie procesu opisywania funkcjonalności za pomocą dostarczonych i opisanych wyżej narzędzi, również od strony programistycznej. Narzędzie ma przekładać stworzone opisu na kod w wybranym języku programowania, w tym wypadku C#.

Wracając do założeń zaproponowanego rozwiązania, narzędzie ma wspierać proces zgodny z BDD, a to oznacza przede wszystkim to, że w momencie powstawania opisu funkcjonalności, a w tym poszczególnych scenariuszy testowych, sama funkcjonalność nie istnieje. Nie ma ani pliku, w którym znajdzie się odpowiedni kod, ani też tym bardziej nazwy klasy, która będzie testowana zgodnie ze scenariuszami. Dlatego na wyjściu generatora będą pliki zawierające deklaracje klas i metod, jednak nie będą one zawierały żadnej szczegółowej implementacji.

5.3.1 Efekt użycia generatora

Zadaniem generatora jest na podstawie pliku zawierającego opis funkcjonalności i mającego rozszerzenie `.feature` jest:

- Przygotowanie struktur programistycznych, które będą umożliwiały implementację różnych testów w zależności od ich charakteru (jednostkowe, integracyjne, akceptacyjne)
- Przygotowanie odpowiednich metod nazwanych w taki sposób, aby mówiły jak najwięcej o teście
- Przygotowanie takiej hierarchii struktur programistycznych, aby implementacja nie wymagała powtarzania, albo niepotrzebnego, wielokrotnego odwoływania się do kawałków kody implementujących np. tło dla wielu scenariuszy.
- Zapewnienie narzędzie i obiektów pomocniczych, które umożliwią w łatwy sposób obsługę przykładów z danymi wejściowymi i wyjściowymi oraz wartości typu `PyString`.
- W końcu struktury powinny być czytelne i krótkie, aby programista mógł łatwo odnaleźć się w kodzie.

Powyższe wymagania w prototypie zostały spełnione poprzez wykorzystanie interfejsów zamiast klas, co pozwoli po pierwsze na wielokrotne wykorzystanie każdego interfejsu. Środowiska programistyczne w przypadku tworzenia klas implementujących interfejsy umożliwiają za pomocą jednego kliknięcia wypełnienie klasy wszystkimi metodami, które muszą być zaimplementowane. Dzięki temu użycie interfejsów jest uzasadnione i bardzo wygodne. Dodatkowo interfejs ma przewagę nad klasą, ponieważ nie można go wypełnić implementacją, przez co początkujący z narzędziem programista nie ma szans się do niego zrazić poprzez ponowne wygenerowanie pliku i nadpisanie dotychczasowej pracy.

W celu zapewnienia przejrzystości kodu generowane pliki zostały podzielone na odpowiadające im kawałki opisu. W następnych podrozdziałach zostaną opisane:

- Interfejs główny
- Interfejsy scenariuszy
- Klasy pomocnicze

5.3.2 Interfejs funkcjonalności z tłem

Główna klasa interfejsu jest tworzona dla jednego pliku z opisem funkcjonalności i posiada:

- Nazwę pliku odpowiednią opisowi występującemu po słowie kluczowym `Feature`. Nazwa jest uzyskiwana za pomocą sprowadzenia wszystkich słów do małych liter, ustawieniu na

wielką pierwszej litery każdego słowa i usunięciu spacji. Następnie nazwa jest poprzedzana literą „I” celem zgodności z notacją przyjętą w C#.

- Przestrzeń nazw w interfejsie (`namespace`) uzyskaną na podstawie nazwy projektu i struktury katalogów zawierających analizowany plik *.feautre* zgodnie z tymi samymi zasadami, na jakich działa środowisko programistyczne. Czyli do nazwy projektu dokładane są nazwy kolejnych w drzewie katalogów poprzedzonych kropką.
- Podsumowanie interfejsu, wyświetlanym, jako jego opis, tworzonym na podstawie narracyjnej części opisu. Podsumowanie może być wykorzystane do stworzenia dokumentacji dostępnej z poziomu narzędzia programistycznego jako opis danego interfejsu. Zatem wszystkie linijki `As, I want, In order` są wprowadzanie w konstrukcję zaprezentowaną na Listing 20.

Listing 20 - Klasa interfejsu głównego

```
/// <summary>
/// As user of the tested system
/// I want to be properly validated basing on my credentials
/// In Order to be sure nobody can login or see my data not having proper credentials
/// </summary>
public interface ILoginCredentialsValidation
```

- Nazwa interfejsu jest analogiczna do nazwy pliku
- Zawartość interfejsu to poszczególne metody składające się z:
- Opisu w postaci podsumowania, które zawierającego pełną treść warunki (*`Given, When, Then`*)
- Odpowiedniego atrybutu (który może być wykorzystany do dalszej automatyzacji testów opisanej w podsumowaniu niniejszej pracy) wygenerowanego na podstawie opisu z poprzedniego punktu, gdzie wartości zostały zastąpione wyrażeniem regularnym.
- Definicji samej metody i parametrów, jeżeli dostępne. Parametry są określane na podstawie wartości i ich nazw, lub na podstawie zmiennych

Listing 21 - Zawartość interfejsu głównego

```
/// <summary>
/// Given there is "anowak" user with "haslo123" pass,
/// </summary>
[Given(@"there is (.*) user with (.*) pass")]
void givenThereIsUserWithPass(System.String user, System.String pass);

/// <summary>
/// Given there is no "pjaworski" user,
/// </summary>
[Given(@"there is no (.*) user")]
void givenThereIsNoUser(System.String user);
```

5.3.3 Interfejs scenariusza

Na podstawie każdego ze scenariuszy lub zarysów scenariuszy jest tworzony osobny interfejs, który dziedziczy po głównym interfejsie opisującym funkcjonalność. Program z kodem:

- Posiada nazwę pliku, która zawiera nazwę funkcjonalności, kropkę, nazwę danego scenariusza
- Zawiera
- Przestrzeń nazw identyczną z interfejsem głównym
- Podsumowanie interfejsu, które można wykorzystać, jako dokumentację
- Atrybut interfejsu mogący być wykorzystany przez narzędzie uruchamiające testy (więcej w podsumowaniu)
- Deklarację interfejsu, gdzie jego nazwa składa się z litery „I”, słów składających się na opis w pliku *feature*, z wielką literą na początku każdego słowa. Oczywiście nazwa jest pozbawiona znaków białych, po niej następuje deklaracja dziedziczenia po interfejsie głównym.

Listing 22 - Interfejs scenariusza

```
namespace projectName.Folder1.Folder11
{
  /// <summary>
  /// Deny access to anowak with wrong credentials
  /// </summary>
  [Scenario(@"Deny access to anowak with wrong credentials:")]
  public interface IDenyAccessToAnowakWithWrongCredentials :
ILoginCredentialsValidation
```

- Kolejnych deklaracji metod implementujących kolejne warunki `Given, When, Then`. Całość jest analogiczna do interfejsu głównego.

Listing 23 - Opis metod w interfejsie scenariusza

```
  /// <summary>
  /// I've put into dialog "anowak" user and "wrong" pass
  /// </summary>
  [Given(@"I've put into dialog (.*) user and (.*) pass")]
  void givenIvePutIntoDialogUserAndPass(System.String user, System.String pass);

  /// <summary>
  /// I press log me in
  /// </summary>
  [When(@"I press log me in")]
  void whenIPressLogMeIn();

  /// <summary>
  /// access to my data is denied
  /// </summary>
  [Then(@"access to my data is denied")]
 void thenAccessToMyDataIsDenied();
```

5.3.4 Klasy pomocnicze

Wyżej wymienione interfejsy nie zaspokajają potrzeby obsługi tabel i złożonych wartości tekstowych `PyString`. W tym celu przy interfejsach zapisywane są klasy pomocnicze, które w swych konstruktorach zawierają instrukcję wypełniające je danymi zgodnie ze opisem scenariusza.

Dla tabeli klasa pomocnicza wygląda tak:

Listing 24 - Klasa pomocnicza dla przykładów

```
public class DenyOrGrantAccessForSeveralCredentialsExamples :
anowak.BDD.Library.Examples
{
  public void DenyOrGrantAccessForSeveralCredentialsExamples()
  {
    Example ex = new Example();
    ex.Add("user", jkowalski);
    ex.Add("password", dupa08);
    ex.Add("granted", true);
    examples.Add(ex);
    Example ex = new Example();
    ex.Add("user", anowak);
    ex.Add("password", haslo1234);
    ex.Add("granted", false);
    examples.Add(ex);
  }

}
```

Nazwa klasy jest tworzona na podstawie scenariusza w analogiczny do klasy scenariusza sposób, jedynie nie jest dodawana liter „I” na początku, a na koniec nazwy doklejany jest przyrostek `Examples`. Całość dziedziczy po specjalnie przygotowanej klasie `Examples`, która zawiera kolekcję wierszy tabeli. Każdy wiersz tabeli składa się z kolekcji par klucza i wartości, gdzie kluczem jest nazwa kolumny. W ten sposób implementacja testu może opierać się na pętli `foreach`, co zdecydowanie skraca kod i ułatwia jego zrozumienie.

Wartość `PyString` jest udostępniana na tej samej zasadzie. Jednak nazwa klasy jest uzależniona od nazwy scenariusza i nazwy warunku (`Given`, `When`, `Then`) z tego względu, że możliwym jest, by wartość taka pojawiła się w scenariuszu więcej niż jeden raz oraz w kilku scenariusza. Stąd taka konstrukcja nazwy, by uniknąć ewentualnych konfliktów.

5.3.5 Integracja ze środowiskiem

Prototyp został przygotowany do współpracy z *Visual Studio* jako dwa niezależne elementy. Jednym z nich jest zestaw narzędzi wspierających edycję plików z rozszerzeniem `.feature`, a drugim jest generator wielu plików na podstawie jednego wsadowego. W tym celu została stworzona klasa `VsMultipleFileGenerator`, która implementuje dostępny w *Visual Studio SDK* interfejs `IVsSingleFileGenerator`. Problemy z tym związane (niezgodność słów *Multiple* i *Single*) jest opisana w rozdziale trudności.

Powyższa klasa została z kilkoma pomocniczymi wydzielona do osobnego projektu kompilowanego do postaci *DLL*. Aby poprawnie korzystać z narzędzia konieczne jest zarejestrowanie pliku *DLL* jako rozszerzenia w rejestrze. *Visual Studio* po odpowiedniej rejestracji pliku i zmianie

ustawień, przy każdym zapisie wywołuje to rozszerzenie przekazując mu informacje o zawartości pliku, położeniu, nazwie oraz o właściwościach projektu.

Generator po otrzymaniu powyższych informacji przetwarza zawartość pliku i w razie problemów zwraca wyjątek, który *Visual Studio* wyświetla jak zwykły wyjątek podczas kompilacji, dostępny w oknie „*Error List*". W razie udanego przetwarzania tworzy nowe pliki, które są podłączane w drzewie projektu, pod plik źródłowy z rozszerzeniem `.feature`. Na koniec przeprowadzane jest czyszczenie katalogu z wszystkich plików, które podłączone są pod plik źródłowy, a nie zostały wygenerowane przy tym uruchomieniu generatora. Jest to zabezpieczenie przed plikami powstałymi w wyniku przetwarzania wcześniejszych wersji pliku źródłowego, które po zmianach tam wprowadzonych nie mają prawa bytu.

6. Podsumowanie

W tym rozdziale ujęte zostały trzy obszary związane z realizacją prototypu zgodnie z koncepcją rozwiązania. Kolejno zostały omówione trudności, które pojawiły się w trakcie realizacji, wykryte wady założeń wynikających z koncepcji oraz kierunki dalszych działań związanych, które należy podjąć celem wprowadzenia narzędzia do szerszego użytku.

6.1. Trudności

W niniejszym podrozdziale przedstawione zostaną zagadnienia, które podczas realizacji prototypu W trakcie realizacji prototypu zgodnie z założeniami napotkany został szereg trudności związanych z wybranym środowiskiem, jak i trudnością w realizacji założeń.

6.1.1 Obsługa elementów wieloliniowych

Założeniem zastosowanego języka jest jego jednoliniowy charakter, gdzie interpretacja każdej linii nie zależy od pozostałych. Jednak od tej reguły został stworzony jeden wyjątek w postaci PyString, który jest elementem odpowiednio oznaczonym na początku i na końcu, jednak linie występujące pomiędzy średnikami muszą być analizowane kontekstowo. To zdecydowanie utrudnia pracę narzędziom z wiązanym z podkreślaniem, kolorowaniem i analizą kodu na bieżąco. Związane jest to z tym, że te narzędzia najczęściej oparte są na analizie edytowanej linii.

Zastosowanie PyString nie okazało się problemem przy realizacji elementów generatora, ponieważ tam, analiza jest zawsze od początku pliku i w każdym jej momencie znany jest kontekst.

6.1.2 Integracja generatora

Zaskoczeniem i powodem przejściowych trudności okazało się też podejście producenta środowiska programistycznego do opcjonalnych analizatorów kodu. Mogłoby się wydawać, że przy możliwości wykorzystania biblioteki *MEF*, która doskonale sprawdziła się przy zewnętrznych wtyczkach rozszerzających edytor, w przypadku generatora będzie podobnie. Niestety, w związku z brakiem dokumentacji, która wskazywałaby możliwość użycie *MEF*, konieczne było przygotowanie odpowiedniego narzędzia kompilowanego do *DLL*, którą następnie trzeba było rejestrować w *Visual Studio*.

6.1.3 Generator

Visual Studio jest narzędziem przygotowanym na obsługę rozszerzeń, jednak nie wszystko zostało odpowiednio udokumentowane, a nawet nie zostało przewidziane. O ile środowisko dostarcza procedurę, aby analizować zapisywany plik o odpowiednim rozszerzeniu przez zewnętrzną wtyczkę tak daje jedynie możliwość utworzenia na tej podstawie jednego pliku. W przypadku założeń tej pracy było to zdecydowanie niedopuszczalne.

Źródłem problemu był brak odpowiednika interfejsu `IVsSimpleFileGenerator`, który by zawierał w nazwie słowo `Mulit` zamiast `Simple`. Rozwiązaniem stało się stworzenie odpowiedniej klasy implementującej powyższy interfejs i odpowiednie jej rozszerzenie.

6.2. Możliwości rozwoju

Przedstawiony w niniejszej pracy prototyp jest odpowiedzią na zapotrzebowanie środowisk realizujących projekty informatyczne w oparciu o *BDD*. Jednoznacznie pokazuje również, że narzędzie tego typu spełnia swoją rolę i może się przyczynić do wzrostu efektywności procesu wytwórczego oprogramowania. Jednak na obecnym poziomie prototyp nie może być traktowany, jako w pełni dojrzałe narzędzie i należałoby ten projekt rozwinąć w następujących obszarach.

6.2.1 Rozszerzenie prototypu

Prototyp w obecnej postaci nie zawiera elementów związanych z:

- Weryfikacji zgodności liczby i nazw parametrów w zarysie scenariusza i w tabeli oraz samych wierszy tabeli pod kątem jednakowej liczby kolumn.
- Nie wszystkie zasady związane z konstrukcjami języka do opisywania funkcjonalności zostały zawarte w rozszerzeniach do pokazujących błędy.
- Automatyczne uzupełnianie kodu mogłoby obejmować podpowiadanie zmiennych przy tworzeniu tabeli.

6.2.2 Wsparcie środowiska testowego

Prototyp jest narzędziem, które zostało stworzone w odpowiedzi na problemy na styku biznes, a zespół programistów. Jednak do tego, aby efektywnie wspierać programistów należałoby przygotować wsparcie dla najczęściej wykorzystywanych środowisk testujących. Takie wsparcie powinno obejmować tworzenie zrębów klas implementujących poszczególne testy w taki sposób, aby nadane były im wszystkie niezbędne atrybuty do tego, by narzędzie uruchamiające testy, jak *NUnit* wiedziało, co z daną metodą zrobić. W obecnej chwili elementy te są pozostawione do implementacji programiście. Można wyobrazić sobie taką funkcjonalność jako odrębną wtyczkę, możliwe, że instalowaną w pakiecie, która dawałaby opcje w menu kontekstowych w narzędziu, a następnie tworzyła klasę, dodawałaby zręby klas implementujące odpowiednie interfejsy dodając do tego odpowiednie atrybuty.

6.2.3 Ponowne użycie kodu

Podczas analizy pewnej liczby przypadków testowych widać, że duża liczba warunków `Given, When, Then` mogą być różnie zapisane, ale odnosić się do tej samej funkcji. Związane jest to z dużą elastycznością języka naturalnego, który mimo włożenia go w ramy zgodne z założeniami niniejszej pracy nadal pozwala na konstrukcje zdaniowe identyczne znaczeniowe, a różne składniowo.

Można założyć, że warunki z Listing 25 byłby użyte w dwóch różnych scenariuszach.

Listing 25 – podobne konstrukcje zdaniowe o identycznym znaczeniu

```
Given there is "anowak" user
Given there is user with "anowak" login
Given in the system is "anowak" user registered
```

Powyższe wymagania mogłyby być realizowane przez jedną metodę. Stąd stosowanie w aplikacji odpowiednich atrybutów i ewentualna możliwość przypisywania więcej niż jednego atrybutu do metody. Jednak problem może być rozwiązany trywialnie poprzez zastosowanie takiego kodu

Listing 26 - Możliwość wykorzystania jednej metody dla wielu warunków

```
public void givenThereIsUser(string user)
 {
   //implementacja
 }

 public void givenThereIsUserWithLogin(string login)
 {
   givenThereIsUser(login);
 }
```

Patrząc na Listing 26 nie widać dużej wartości dodanej w implementacji mechanizmów, które będą poszukiwały odpowiednich metod po ich atrybutach.

6.2.4 Dokumentacja

Niesłychanie przydatnym narzędziem będzie też wtyczka, która pozwoli na przygotowanie kompletnej dokumentacji dotyczącej testów zawierającej odpowiednie scenariusze z pominięciem niepotrzebnych komentarzy. Całość powinna być możliwa do otworzenia w narzędziach typu *Open Office, Ms Office* i wszelkimi czytnikami *PDF*. Dzięki temu wymiana informacji pomiędzy klientem, a wykonawcą byłaby łatwiejsza. Idealnym rozwiązaniem byłby też raport z wykonanych testów, ile razy były udane, kiedy ostatni raz był nieudany, kiedy ostatni raz był wykonany dany test.

7. Bibliografia

[Adzic] G. Adzic, Specification by Example - How successful teams deliver the right software., 2011, Maning Publication Co., ISBN 9781617290084.

[Agile] "Twelve Principles of Agile Software" 5 2 2014 [Online]. Available: http://agilemanifesto.org/principles.html

[Bender, James] Bender, James; McWherter, Jeff, Professional Test Driven Development with C# - Developing Real-Word Applications with TDD, 2012, Wiley Publishing Inc., ISBN 978-0-470-64320-4.

[Geneshan] D. Ganeshan, „http://www.theserverside.com," 06 06 2011. [Online]. Available: http://www.theserverside.com/tip/Waterfall-versus-Agile-methods-A-pros-and-cons-analysis.

[Mef] Microsoft, „Microsoft .Net MEF," Microsoft, 31 01 2014. [Online]. Available: http://mef.codeplex.com/.

[MicrosoftC#] Microsoft, „Visual C# resources," 31 01 2014. [Online]. Available: http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx.

[MicrosoftEVS] Microsoft, „Extend Visual Studio," Visual Studio, 15 06 2013. [Online]. Available: http://msdn.microsoft.com/en-US/vstudio/vextend/.

[MicrosoftIS] Microsoft, „Using Intellisense," 29 10 2013. [Online]. Available: http://msdn.microsoft.com/pl-pl/library/hcw1s69b.aspx.

[MicrosoftNet] Microsoft, „Overview of the .NET Framework," 23 11 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx.

[MicrosoftVS] Microsoft, „#define VISUAL_STUDIO," 31 01 2014. [Online]. Available: http://msdn.microsoft.com/en-US/vstudio.

[North] D. North, „Introducing BDD" 10.01.2013 [Online]. Available: http://dannorth.net/introducing-bdd/.

[Roy] O. Roy, The Art of Unit Testing with Examples in .NET, 2009, Manning Greenwich, ISBN 978-1-933988-27-6.

[Sanderson] S. Sanderson, „http://blog.stevensanderson.com,” 3 3 2010. [Online]. Available: http://blog.stevensanderson.com/2010/03/03/behavior-driven-development-bdd-with-specflow-and-aspnet-mvc/.

[Specflow] Specflow, „Getting Started,” 10 10 2013. [Online]. Available: http://www.specflow.org/getting-started/.

[StoryQ] „Writing your first StoryQ test,” 8 7 2010. [Online]. Available: http://storyq.codeplex.com/wikipage?title=write%20your%20first%20StoryQ%20test&referringTitle=Home.

[Świdziński] M. Świdziński, „Wprowadzenie do językoznastwa strukturalnego,” 4 11 2013. [Online]. Available: http://www.mswidz.republika.pl/pliki/materialy/EGO1.pdf.

[Wynne, Matt, Hellesoy] Wynne, Matt; Hellesoy, Aslak, The Cucumber Book - Behavioral Driven Development for Testers and Developers, 2012 The Pragmatic Programmers, ISBN 13: 978-1-934356-80-7

8. Spis ilustracji

9. Listingi