**WYKŁAD 1**

**.NET** - Jest to rozwiązanie rozwijane od początku lat 2000 przez firmę Microsoft. Prace nad platformą rozpoczęły się już w latach 1990. W roku 2000 Microsoft,

Hewlett-Packard i Intel rozpoczęły pracę nad ustandaryzowaniem pewnych rozwiązań

pod nazwą CLI (ang. Common Language Infrastructure). CLI to otwarta specyfikacja, będąca

standardem ISO i ECMA.

**CLR** - CLR (ang. Common Language Runtime) to komponent platformy .NET, który

zarządza uruchamianiem aplikacji napisanych w języku pośrednim. Język IL (ang.

Intermediate Language) lub CIL (ang. Common Intermediate Language), a w nieco starszych

wersjach .NET Framework określany był jako MSIL (ang. Microsoft Intermediate Language). CLR jest najważniejszym elementem platformy .NET. Jego odpowiedzialnością jest

uruchamianie i zarządzanie aplikacjami pisanymi w języku pośrednim. W momencie

uruchomienia aplikacji jej uruchomieniem (przydziałem pamięci i czasu procesora itd.)

zazwyczaj zarządza system operacyjny. W przypadku aplikacji napisanej w języku

pośrednim jej uruchomienie zajmuje się CLR. Innymi słowy kod napisany w **języku**

**pośrednim** nie jest kodem maszynowym, więc nie może być bezpośrednio uruchomiony

przez system operacyjny. Niezależnie z jakiego korzystamy języka - podczas jego uruchamiania każda aplikacja

kompilowana jest do języka CIL (ang. Common Intermediate Language, Intermediate

Language lub kiedyś MSIL - Microsoft Intermediate Language). Jest to język pośredni, który

możemy porównać z byte code’em znanym Wam z języka Java. Język ten możemy

podejrzeć. Przypomina jednak bardziej assembler i z pewnością bezpośrednie

programowanie w nim byłoby niezwykle mało wydajne i uciążliwe.

**CLR** implementuje mechanizm nazwany w skrócie **GC** (ang. Garbage Collector),

który znany jest nam z wirtualnej maszyny Javy. Dzięki temu CLR odciąża nas od wielu

zadań związanych z zarządzaniem i zwalnianiem pamięci, chroniąc nas przed trudnymi do

zdiagnozowania błędami, które spowodowane są wyciekami pamięci. CLR implementuje

“leniwe” podejście do zarządzania pamięcią.

Zasób **zarządzalny** określa “to” czym zarządza CLR. Pełnię “władzy” nad

zasobem zarządzalnym ma CLR. Decyduje o tym, kiedy dany zasób zwolnić i w jego gestii

leży cały cykl życia danej zmiennej.

Zasób **niezarządzalny** określa zasoby, których cykl życia nie leży w gestii CLR (a

przynajmniej nie w całości). Przykładem sytuacji, w której korzystamy z zasobu

niezarządzalnego jest odczytywanie danych z pliku lub komunikacja z bazą danych.

**W jaki sposób rozpoznać, czy zasób jest niezarządzalny?** Jest na to prostu sposób.

Większość tego typu zasobów implementuje interfejs **IDispose** (wszystkie interfejsy w

.NET mają w nazwie prefiks “I” - jest to domyślna konwencja). Interfejs ten definiuje jedną

metodą Dispose, które służy zwolnieniu zasobów wykorzystywanych przez dany obiekt.

Zazwyczaj, jeśli klasa, którą wykorzystuje implementuje ten interfejs - oznacza to, że jest

to zasób niezarządzalny. Innymi słowy jest to zasób, który powinniśmy zwolnić tak szybko

jak jest to możliwe (czyli uruchomić metodę Dispose).

**Bezpieczeństwo typów** znacie już z pewnością z języka Java (czyli potrzeba

deklarowania typu zmiennej przed jej użyciem). Oznacza to, że operacje jakie możemy

wykonać na danej zmiennej są zależne od jej typu. Dzięki temu już na etapie kompilacji

możemy otrzymać sensowną informację o błędzie (ang. compile-time error). Ogranicza to

liczbę popełnianych przez nas błędów i przyspiesza pracą nad kodem aplikacji.

Innym terminem, który warto wspomnieć w kontekście “statycznych typów” jest

**enkapsulacja**. W skrócie jednym z jej mechanizmów jest wykorzystanie słówek

**public/private**. Dzięki temu możemy wyraźnie oddzielić “wnętrze” klasy od jej publicznego

API. Będzie to dla nas istotne w późniejszych wykładach.

Innym mechanizmem, który warto wspomnieć jest **local type inference**. Jest to

mechanizm, który pozwala kompilatorowi “wydedukować” typ wyrażenia po lewej stronie

wyrażenia na podstawie wyrażenia po prawej stronie.

Platforma C# pozwala nam również korzystać z tzw. **unsafe code**. Takim

“niebezpiecznym kodem” jest sytuacja, w której korzystamy z zasobów/bibliotek będących

poza kontrolą CLR. Często określane są one komponentami COM (ang. Component Object

Model). Jest to standard definiowania interfejsów programistycznych zdefiniowany przez

Microsoft. Jest wykorzystywany do niskopoziomowej komunikacji (np. w przypadku

DirectX). Zazwyczaj w przypadku komunikacji z nieco starszymi aplikacjami (często

aplikacjami spadkowymi) następuje potrzeba komunikacji z takimi interfejsami.

**Base Class Libraries (BCL)** to biblioteki zawierające różnego rodzaju klasy, które implementują najbardziej popularne funkcjonalności i algorytmy. Tego rodzaju biblioteki mogą znaleźć

zastosowanie w dowolnych aplikacjach i ich znajomość jest niezwykle ważna. Będą nam

niemal zawsze niezbędne w pracy nad dowolną aplikacją.

**Framework Class Libraries (FCL)** obejmuje biblioteki zawierające klasy, które mogą być przydatne w zależności

od rodzaju aplikacji nad jaką pracujemy: aplikacja webowa, aplikacja desktopowa,

sieciowa usługa itd.

**Windows Presentation Foundation (WPF)** Zestaw bibliotek związanych z tworzeniem aplikacji desktopowych. WPF

prezentuje podejście nieco “nowocześniejsze” niż Windows Forms. Pozwala m.in. na

definiowanie widoków z pomocą języka XAML (ang. Extensible Application Markup

Language), pozwalającym na niezwykle wygodne i szybkie tworzenie interfejsów

graficznych i nie tylko.

Jest to preferowane podejście w przypadku tworzenia nowych aplikacji desktopowych.

Chociażby z tego powodu, że pozwala na pisanie aplikacji, których wygląd jest niezależny

od rozdzielczości urządzenia wyświetlającego

**ASP.NET** Jest to zbiór bibliotek pozwalający tworzyć różnego rodzaju aplikacje webowe (np.

Webforms, ASP.NET MVC, WebApi). Aplikacje tworzonej w tej technologii zazwyczaj

wymagają serwera IIS (ang. Internet Information Services), który umożliwia wygenerowanie

treści HTML, CSS, JS, czyli języków, które mogą być zinterpretowane przez przeglądarkę.

**Windows Communication Foundation (WCF)** określa biblioteki służące do tworzenia

różnego rodzaju usług sieciowych. Jest to niezwykle elastyczne rozwiązanie, które

pozwala na komunikacji systemów heterogenicznych. Może służyć zarówno do

stworzenia API korzystającego z protokołu HTTP - np. w stylu REST (ang. Representational

State Transfer) lub korzystające z dowolnego innego protokołu.

**ADO.NET (ang. ActiveX Data Objects for .NET)** to biblioteki, które z racji tego

przedmiotu poświęcamy wyjątkową uwagę. Obejmuje biblioteki, które służą komunikacji

z różnymi źródłami danych w tym relacyjnymi bazami danych.

**LINQ (ang. Language Integrated Query)** - język będący częścią języka C# (jednego z

dostępnych języków platformy .NET). Jest to język zapytań, który operuje na kolekcjach

obiektów (np. ArrayList’a w Javie). Został wprowadzony w wersji języka C# 3.5 i stara się

rozwiązać problem określany jako problem “różnicy impedancji”. Krótko mówiąc jego

składnia przypomina język SQL, ale pozwala nam operować na różnego rodzaju źródłach

danych (plikach XML, relacyjnych bazach danych, i innych) w sposób w pełni obiektowy.

Sam LINQ zależnie od źródła danych wykorzystywany jest zazwyczaj z innymi

bibliotekami - np. EF (ang. Entity Framework) w celu komunikacji z relacyjną bazą danych.

**Języki platformy .NET**

**C#** - statycznie typowany język obiektowy, którego składnia podobna jest do Javy. Będzie

to główny język, z którego będziemy korzystać. Mimo to z pewnością sama składnia

języka C# często oferuje nieco więcej udogodnień niż Java. Obecnie C# rozwija się

niezwykle szybko i wiele rozwiązań dopiero później pojawia się w innych językach. Jest to

oczywiście spowodowany tym, że język rozwijany jest przez jedną firmę (Microsoft), która

nie musi konsultować zmian ze społecznością (jak to zazwyczaj ma miejsce w języku Java).

Oczywiście można to uznać zarówno za wadę jak i zaletę.

**F#** - języka funkcyjny, związany głownie z analiza danych i ich przetwarzaniem. Z

pewnością wygodny w sytuacji, kiedy chcemy zająć się ML (ang. Machine Learning), AI (ang.

Artificial Intelligence) lub transformacją danych.

**Visual Basic** - obiektowy język programowania, którego składnia przypomina znacznie

bardziej język naturalny. Z tego powodu może być łatwiejszy do wykorzystania dla osób

rozpoczynających naukę programowania. Obecnie język raczej nie wykorzystywany do

nowych aplikacji.

I wiele innych... Jest np. Wersja Ruby’iego i Python’a na platformę CLR. Niegdyś istniała

wersja języka Java o nazwie J# na platformę CLR - jednak w tamtym czasie Sun

Microsystems pozwał firmę Microsoft.

**WYKŁAD 4 – Komunikacja z bazą danych**

ADO.NET – zbiór bibliotek, które pozwalają nam komunikować się z różnymi źródłami danych. Obejmuje szereg klas dzięki którym możemy łączyć się z różnymi bazami danych

Obiekt SQLCommand pozwala nam wyspecyfikować jaki typ interakcji chcemy użyć z bazą danych

**Wykład 6 – Entity Framework**

**Entity Framework** – jest to obiektowo-relacyjny mapper który pozwala developerom .NET pracować z bazami danych za pomocą obiektów .NET. Eliminuje on potrzebę dla większości kodu potrzebującego dostępu do danych którą programiści najczęściej muszą pisać

Zalety:

* Wieloplatformowość
* Tworzenie zapytań bez sql-a (LINQ)
* Modelowanie – zwyczajne klasy c# - POCO (Plain Old CLR Object), czyste obiekty, pracujemy w pełni obiektowo
* Śledzenie zmian w obiektach
* Generowanie polecenia INSERT, UPDATE, DELETE
* Współbieżność – pomiędzy pobraniem danych a zmianą w tych danych ktoś może też tych danych używać
* Transakcje – domyślne wykonywanie komend
* Caching
* Konwencja ponad konfigurację
* Konfiguracja pozwala na konfiguracje modelu używając adnotacji czy fluent API
* Migracje

Jak działa EF?

1. Mapowanie
2. Generowanie kodu SQL
3. Śledzenie zmian
4. Zapisywanie zmian w bazie

Klasa kontekstowa - czyli DbContext

Podejścia do pracy z BD:

Database-First - czyli baza już istnieje

Code-First - dla nowych aplikacji, BD jest generowana z kodu

Model-First (rzadko używane) – wersja mieszana, z narysowanego modelu

**Model MVC (Model View Controller)**

**Kontroler** – najważniejszy element; są to klasy obsługujące żądanie użytkownika; logika biznesowa, pobieranie czegoś z bazy danych, walidacja itd.

**Model** – klasy POCO modelujące coś z bazy danych (np. klasa Student itp.)

**Widok** – sposób generowania kodu HTML; tu jest html css, nie ma tu żadnej logiki biznesowej. Widok to interfejs

**Silnik widoku / Viev engine** – silniki do generowania widoku; sposób na to w jaki sposób powiązać np. kod c# i html i jak wygenerować dynamicznie kod html

**Różnica impedancji** - To dysonans między "światem obiektowym" a "światem relacyjnym"; określa ona problem połączenia języka zapytań z językiem programowania

**Testy software’u**

Cele testów:

* Znalezienie bugów
* Zademonstrowanie, że produkt funkcjonuje
* Walidowanie jakości software’u używając minimum kosztów (testy kosztują); jak najmniej kosztów, jak najwięcej jakości
* Generowanie wysokiej jakości test case’ów

Cykl życia tworzenia testów:

1. Zbieramy wymagania
2. Planowanie testów
3. Tworzenie test case’ów
4. Tworzenie środowiska pod testowanie
5. Wykonywanie testów
6. Zamknięcie cyklu testowania

Test case – opis wykonania jakiegoś konkretnego przypadku biznesowego

Test case zawiera:

* Identyfikacja którego wymagania dotyczy
* Kryteria, kiedy test się powiódł, a kiedy nie
* Konkretne kroki wykonywania testu
* Przykładowe dane

Weryfikacja – sprawdzenie czy software pasuje do specyfikacji (czy dobrze budujemy produkt?)

Walidacja – czy produkt robi to co chce, aby użytkownik robił (czy budujemy dobry produkt?)

Metody testowania:

1. Black-box testing – nie mamy wiedzy na temat kodu, traktujemy go jak czarną skrzynkę; Testy bazują na wymaganiach i specyfikacji; w takich testach biorą udział np. użytkownicy, ale nie tylko
2. White-box testing – testowanie mając pełną wiedzę co się dzieję w środku kodu; często zajmują się nim programiści.

White-box testing dzieli się na dwie kategorie:

1. Statyczny
2. Code walk through – czyli sprawdzanie po kolei kodu
3. Formal inspections – czyli przeglądanie dokumentacji
4. Desk checking – czyli rozpisywanie np. na kartce poszczególnych kroków danego algorytmu
5. Strukturalny
6. Control flow testing – czyli IF-y, sprawdzamy każdą ścieżkę uruchomienia w kodzie; czysto techniczne testy, często nie mające nic wspólnego ze sprawdzeniem wymagań klienta
7. Basic path testing – sprawdzanie głównej ścieżki
8. Loop testing – czyli sprawdzanie pętli
9. Data flow testing – testowanie przepływu danych i tego jak dane się zmieniają

Przykłady testowania:

* Alpha testing – testowanie przed wysłaniem do klienta (często na maszynie wirtualnej, bez urządzeń)
* Acceptance testing (UAT – user acceptance test) – test który aplikacja musi przejść
* Ad-hoc testing – osoba, która nie zna kodu, specyfikacji itp. siada i testuje bez informacji o tej aplikacji
* Accessebility testing – testowanie dostępności dla osób z dysfunkcjami (osoby niewidome, upośledzone umysłowo, starsze osoby itp.)
* Beta testing – testowanie na prawdziwym sprzęcie, realni użytkownicy testują aplikację
* Back-end testing – testowanie wszystkiego co nie ma interfejsu graficznego
* Browser compatibility testing – testowanie czy aplikacja działa na przeglądarkach
* Backward compatibility testing – testowanie wstecznej kompatybilności (np. sprawdzanie czy nowa aplikacja działa ze starą bazą danych itp.)
* Boundary value testing – sprawdzanie skrajne i typowe wartości (np. najmniejszą liczbę, typową i skrajnie największą)
* Comparsion testing – porównywanie różnych wersji aplikacji (np. czy nowa wersja jest szybsza od starej itp.)
* End-to-end testing – testowanie na środowisku produkcyjnym, czyli w takim środowisku w jakim naprawdę będziemy używać aplikacji (np. na realnym telefonie itp.)
* Exploratory testing – nieformalne testy doświadczonego testera, który nie zna kodu (podobne do ad-hoc)
* Functional testing – jedne z najważniejszych, testy ogólne dotyczące wymagań użytkowników
* Non-functional testing – testowanie np. czy aplikacja wspiera różne przeglądarki, albo czy aplikacja wytrzyma np. 1000 użytkowników na raz
* Monkey testing – często automatyczne, testowanie „jak małpa”
* Integration testing – testowanie kilku modułów na raz (np. baza danych i klasa aplikacji)
* Load testing – czyli ile jest w stanie wytrzymać nasza aplikacja (np. aplikacja wytrzyma np. 1000 użytkowników na raz)
* Stress testing – podobne do load testingu, czyli np. sprawdzamy większą ilość użytkowników niż nasza aplikacja zakłada (np. czy aplikacja w ogóle będzie działała)
* Recovery testing – np. odtwarzanie aplikacji z backupu; sprawdzanie przypadku, kiedy aplikacja pada
* Regression testing – testowanie czy po dodaniu nowego kodu pozostałe funkcje działają
* Risk-based testing (RBT) – testowanie najważniejszych funkcji
* Security testing – ważne testy, np. sprawdzanie czy używamy https, testy socjotechniczne (np. przychodzi pan sprzątacz i sprawdzamy, czy zostanie wpuszczony do serwerowni)
* Unit testing – testujemy jedną rzecz np. testujemy tylko jedną metodę i tylko kod w niej bez użycia bazy danych (test jednostkowy)
* Usability testing – czy aplikacja jest w ogóle sensowna (może się okazać, że aplikacja jest do niczego nie potrzebna, bo pracownicy, którzy mają używać tej aplikacji nie chcą jej używać)