Zaawansowane programowanie obiektowe - wykład 10

Opracowanie: dr Piotr Jastrzębski

Refleksje, atrybuty, programowanie dynamiczne

Pojęcia wstępne

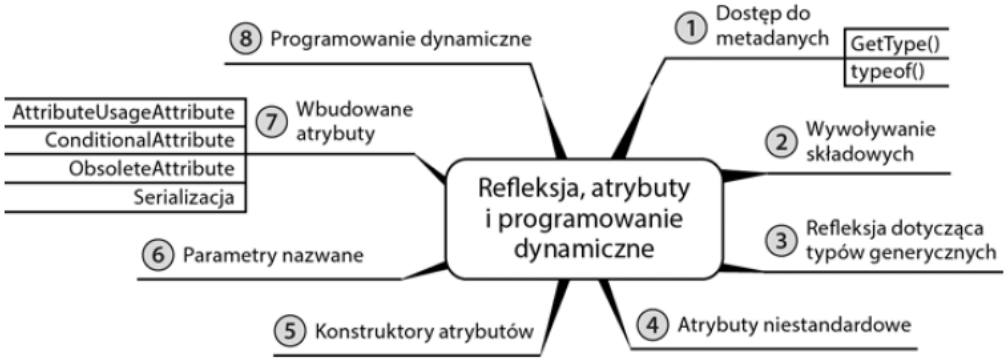
Mechanizm refleksji (czasem nazywane jako odzwierciedlenie) – pojęcie z dziedziny informatyki oznaczające proces, dzięki któremu program komputerowy może być modyfikowany w trakcie działania w sposób zależny od własnego kodu oraz od zachowania w trakcie wykonania. Paradygmat programowania ściśle związany z mechanizmem refleksji to programowanie refleksyjne.

Refleksja pozwala w łatwy sposób zarządzać kodem tak, jakby był danymi. Używa się jej najczęściej do zmieniania standardowego zachowania już zdefiniowanych metod lub funkcji, a także do tworzenia własnych konstrukcji semantycznych modyfikujących język. Z drugiej strony kod wykorzystujący refleksję jest mniej czytelny i nie pozwala na sprawdzenie poprawności składniowej i semantycznej w trakcie kompilacji (niewygodne śledzenie błędów).

Mechanizm ten jest częściej spotykany w językach wysokiego poziomu, zwykle opartych na maszynie wirtualnej.

Atrybuty – element składni języka programowania, który określa konkretną właściwość (znaczenie), nadaną wybranemu elementowi (obiektowi).

Uwaga! W niektórych kontekstach za atrybuty uznaje się pola, zmienne zdefiniowane w klasie. Te elementy programowania zostały już omówione na początku wykładu, w słowo atrybut będzie zawsze powiązane z konstrukcją podaną na początku tego slajdu.


Refleksja, atrybuty i programowanie dynamiczne
1 Dostęp do metadanych
GetType()
typeof()
2 Wywoływanie składowych
3 Refleksja dotycząca typów generycznych
4 Atrybuty niestandardowe
5 Konstruktory atrybutów
6 Parametry nazwane
7 Wbudowane atrybuty
AttributeUsageAttribute
ConditionalAttribute
ObsoleteAttribute
Serializacja
8 Programowanie dynamiczne

Rysunek 1:

Common Intermediate Language

Common Intermediate Language (z ang. Wspólny Język Pośredni, w skrócie CIL, lub IL) – język najniższego poziomu dla platformy Microsoft .NET odczytywalny przez człowieka. Jest to odpowiednik asemblera jako języka pośredniego dla typowych języków wysokiego poziomu (tu: Common Language Infrastructure (CLI) wyrażający kod w C#, Visual Basic .NET, Managed C++ lub dowolnym języku z wielu (40+) języków kompilowanych do CIL). CIL jest tłumaczony bezpośrednio na kod bajtowy.

CIL przypomina obiektowy asembler w całości oparty na stosie. Jego wykonanie następuje za pomocą maszyny wirtualnej.

Początkowo CIL nosił nazwę Microsoft Intermediate Language (MSIL), ale uległa ona zmianie wskutek standaryzacji C# i CIL. Czasem jednak można jeszcze spotkać zastosowanie poprzedniej nazwy, szczególnie wśród starszych użytkowników .NET.

```
.assembly HelloWorld
.class auto ansi HelloWorldApp
{
     .method public hidebysig static void Main() cil managed
     {
          .entrypoint
          .maxstack 1
          ldstr "Hello world."
          call void [mscorlib]System.Console::WriteLine(string)
          ret
     }
}
```

Zadania refleksji

- uzyskanie dostępu do metadanych (pełna nazwa typu, nazwy składowych i atrybuty powiązane z daną jednostką) dotyczących typów z podzespołu,
- dynamicznie wywoływanie składowych typu w czasie wykonywania programu za pomocą metadanych, a nie na podstawie wiązania zdefiniowanego na etapie kompilacji.

"Normalnie":

- gdy kod jest kompilowany do postaci języka maszynowego, wszystkie metadane (np. nazwy typów i metod) są usuwane.

W C#:

- przy kompilacji do języka CIL zachowywana jest większość metadanych dotyczących kodu.

Dostęp - przestrzeń nazw `System.Type`.

System.Type

Przykładowe informacje do uzyskania:

- nazwa typu: `Type.Name`
- "publiczność" typu: `Type.IsPublic`
- typ bazowy: `Type.BaseType`
- obsługiwane interfejsy: `Type.GetInterfaces()`
- podzespół, w którym jest zdefiniowany typ: `Type.Assembly`
- właściwości, typy, metody - składowe typu: `Type.GetProperties()`, `Type.GetMethods()`, `Type.GetFields()`, ...
- atrybuty typu `Type.GetCustomAttributes()`

Metoda `GetType()`

Metoda `GetType()` pochodzi z klasy `Object`, dlatego znajduje się ona w każdym typie. Wywołanie `GetType()` zwraca obiekt typu `System.Type` reprezentujący pierwotny obiekt.

Kod do analizy: Refleksja1, Refleksja2.

Minusy używania tej metody: potrzebujemy instancji, nie da się wykorzystać dla klas statystycznych.

Operator `typeof()`

Na etapie kompilacji jest on wiązany z konkretnym obiektem typu Type i jako parametr bezpośrednio przyjmuje nazwę typu.

Kod do analizy - Refleksja3, Refleksja4.

Klasa generyczna

Klasy generyczne pozwalają na tworzenie obiektów tej samej klasy dla różnych typów danych. Dopiero przy tworzeniu obiektu klasy generycznej podajemy typ danych na którym będzie operował.

Przykład: generyczny stos, kolejka.

Kod do analizy: KlasaGeneryczna, Refleksja5.

`nameof()`

`nameof()` zwraca w postaci stringa nazwę argumentu, który przekazujemy do tej metody.

Powyższe słowo kluczowe nie działa przy typach prostych i przy wyrażeniu `this`.

Kod do analizy - Refleksja6.

Atrybuty

Atrybut to znacznik, który służy do przekazywania informacji do środowiska wykonawczego o zachowaniu różnych elementów, takich jak: klasy, metody, struktury, typy wyliczeniowe czy poszczególne podzespoły naszego programu. Informacje takie mogą zostać zdeklarowane przy użyciu atrybutów. Atrybuty znajdują się pomiędzy nawiasami kwadratowymi. Atrybuty służą do dodawania metadanych takich jak instrukcje dla kompilatora, komentarze, opisy metod oraz klas w naszym programie.

Dwa rodzaje:

- atrybuty predefiniowane,
- atrybuty niestandardowe.

Składnia:

```
[Atrybut(parametry_wskazujace, wlasciowosci = value, ...)]
element
```

Predefiniowane atrybuty:

- `AtributeUsage`
- `Conditional`
- `Obsolete`

Atrybut `AttributeUsage` opisuje sposób użycia innego atrybutu klasy. Określa on typ elementów, które mogą ten atrybut zastosować.

```
[AttributeUsage(
    validon,
    AllowMultiple = allowMultiple,
    Inherited =inherited)]
```

- `validon` określa elementy aplikacji do której można przypisać atrybuty

Kod do analizy - Atrybuty1, Atrybuty2.

`Conditional`

Atrybut ten oznacza metodę warunkową, której wykonanie będzie zależało od zdefiniowanego identyfikatora.

Powoduje warunkową kompilację wywołań metod w zależności od podania wartości takich jak: Debug czy Trace. Dzięki takiej kompilacji warunkowej możemy wyświetlać dodatkowe informacje w trakcie debugowania naszego kodu.

Uwaga: potrzeba przestrzeń nazw `System.Diagnostics`.

Kod do analizy - Atrybuty3.

Obsolete

Atrybut ten oznacza część programu, która nie powinna być dłużej używana. Prosty przykład dotyczy powstania nowej metody przy konieczności pozostawienia starej w naszym kodzie. Tą drugą możemy oznaczyć jako przestarzałą (Obsolete) przez wyświetlenie komunikatu informującego o tym, że nowa metoda powinna być używana.

```
[Obsolete(
   informacja
)]
[Obsolete(
   informacja,
   czy_traktowac_uzycie_jako_blad
)]
```

Kod do analizy - Atrybuty4.

Asercja

Asercja (ang. assertion) – predykat (forma zdaniowa w danym języku, która zwraca prawdę lub fałsz), umieszczony w pewnym miejscu w kodzie. Asercja wskazuje, że programista zakłada, że predykat ów jest w danym miejscu prawdziwy. W przypadku gdy predykat jest fałszywy (czyli niespełnione są warunki postawione przez programistę) asercja powoduje przerwanie wykonania programu. Asercja ma szczególne zastosowanie w trakcie testowania tworzonego oprogramowania, np. dla sprawdzenia luk lub jego odporności na błędy. Zaletą stosowania asercji jest możliwość sprawdzenia, w którym fragmencie kodu źródłowego programu nastąpił błąd.

Testowanie oprogramowania

Testowanie oprogramowania

Testowanie oprogramowania – proces związany z wytwarzaniem oprogramowania. Jest to jeden z procesów zapewnienia jakości oprogramowania. Testowanie ma na celu weryfikację oraz walidację oprogramowania. Weryfikacja oprogramowania pozwala skontrolować, czy wytwarzane oprogramowanie jest zgodne ze specyfikacją. Walidacja sprawdza, czy oprogramowanie jest zgodne z oczekiwaniami użytkownika. Testowanie oprogramowania może być wdrożone w dowolnym momencie wytwarzania oprogramowania (w zależności od stosowanej metody). W podejściu kaskadowym zgodnym z modelem V wysiłek zespołu testerskiego zaczyna się wraz z definicją wymagań i jest kontynuowany po zaimplementowaniu zdefiniowanych wymagań. Nowsze metody wytwarzania oprogramowania (np. Agile) rozkładają wysiłek testerski równomiernie na poszczególne iteracje i skupiają się na testach jednostkowych oraz automatyzacji weryfikacji wykonywanych przez członków zespołu.

Testy można podzielić na kilka sposobów:

- ze względu na weryfikowane obiekty (przykładowo testy klas, komponentów, podsystemów, systemu lub zintegrowanych systemów)
- na białoskrzynkowe (strukturalne), weryfikujące kod źródłowy oraz czarnoskrzynkowe testujące warstwę interfejsu
- bazujące na wymaganiach (testy weryfikujące zgodność implementacji z wymaganiami, np. testy funkcjonalne, testy graficznego interfejsu użytkownika), testy niefunkcjonalne – por. klasyfikacja wymagań (i testów) FURPS+ zdefiniowana w ramach Rational Unified Process (RUP) czy testy weryfikacji (testy sprawdzające zgodność implementacji z założeniami np. programisty)
- ze względu na metodę weryfikacji z wyróżnieniem testów statycznych, bez uruchomienia aplikacji i testów dynamicznych wymagającej pracę na uruchomionym oprogramowaniu.

Standardy w testowaniu

- IEEE 829- 2008 Standard for Software and System Test Documentation
- ISO / IEC / IEEE 29119 Software Testing Standard
- IEEE 730-2014 Standard for Software Quality Assurance Processes
- ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models

Testy jednostkowe

Test jednostkowy (ang. unit test) – metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu – np. metod lub obiektów w programowaniu obiektowym lub procedur w programowaniu proceduralnym. Testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik (np. zwrócone wartości, stan obiektu, zgłoszone wyjątki) z oczekiwanymi wynikami – tak pozytywnymi, jak i negatywnymi (niepowodzenie działania kodu w określonych sytuacjach również może podlegać testowaniu).

Zaletą testów jednostkowych jest możliwość wykonywania na bieżąco w pełni zautomatyzowanych testów na modyfikowanych elementach programu, co umożliwia często wychwycenie błędu natychmiast po jego pojawieniu się i szybką jego lokalizację zanim dojdzie do wprowadzenia błędnego fragmentu do programu. Testy jednostkowe są również formą specyfikacji. Z powyższych powodów są szczególnie popularne w programowaniu ekstremalnym.

Podział:

- analiza ścieżek
- użycie klas równoważności
- testowanie wartości brzegowych
- testowanie składniowe

Analiza ścieżek

Podejście, w którym określamy punkt początkowy oraz punkt końcowy dla przeprowadzenia testów i badamy przebieg możliwych ścieżek pomiędzy nimi. Rozpatrywane możliwe ścieżki od punktu początkowego do punktu końcowego dzielimy na dwa podejścia:

- każda możliwa ścieżka w każdej funkcji została przetestowana
- ścieżki są niemożliwe do sprawdzenia z powodu istnienia pętli

Aby zapobiec niemożności sprawdzenia kodu z powodu pętli stosujemy dwie grupy testów:

Boundary test – działania w pętli nie są wykonywane lub działania w każdej pętli są wykonywane raz dodatkowo wszystkie ścieżki wewnątrz pętli są raz wykonane.

Interior test – działania we wnętrzu pętli uważa się za przetestowane, jeśli zostały wykonane wszystkie ścieżki, które są możliwe przy dwukrotnym powtórzeniu pętli.

Boundary Test: zwracane wartości to zero lub jedno przejście

Użycie klas równoważności

Klasa równoważności jest to zbiór danych o podobnym sposobie przetwarzania, używanych do przeprowadzenia testu. Wykonanie testu z użyciem kilku elementów zbioru, powoduje uznanie całej klasy za poprawną i zwalnia nas od testowania wszystkich elementów w np. 1000-elementowym zbiorze.

Klasy równoważności dzielą się na:

- klasy poprawności – są to przypadki, dla których przewidujemy poprawne wykonanie programu,
- klasy niepoprawności – są to przypadki, dla których przewidujemy błędne wykonanie programu.

Przykłady:

- rejestracja osoby w wieku od 0 do 120 lat: przypadki testowe $= \{15, 18, 30, 60, 5\}$
- długość wiadomości od 1 do 50 znaków: przypadki testowe $= \{1, 2, 5, 8, 30, 45\}$

Testowanie wartości brzegowych

Rozwinięciem testów z użyciem klas równoważności jest testowanie wartości brzegowych. Wartość brzegowa to wartość znajdująca się wewnątrz, pomiędzy lub tuż przy granicy danej klasy równoważności.

Przykłady:

rejestracja osoby w przedziale wiekowym 0 – 120,

testowane wartości brzegowe: {-1, 0, 1, 119, 120, 121}

długość wiadomości od 1 do 50 znaków:

testowane wartości brzegowe: {0, 1, 2, 49, 50, 51}

napięcie od 0 do 100 V:

testowane wartości brzegowe: {-1, 0, 1, 99, 100, 101}

Testowanie składniowe

Podstawowym zadaniem testowania składniowego jest sprawdzenie poprawności wprowadzanych danych do systemu.

Błędy zależne od systemu i środowiska:

- wymuszone wartości pól (bazy danych)
- autokorekty (MS Office)

W testowaniu składniowym należy pamiętać o zasadzie “garbage in – garbage out”, która zadziała gdy:

- brak mechanizmu walidacji danych na wejściu
- brak testów na tolerancje systemu na błędne dane

Test-driven development (TDD)

TDD – technika tworzenia oprogramowania, zaliczana do metodyk zwinnych. Pierwotnie była częścią programowania ekstremalnego (ang. extreme programming), lecz obecnie stanowi samodzielną technikę. Polega na wielokrotnym powtarzaniu kilku kroków:

- Najpierw programista pisze automatyczny test sprawdzający dodawaną funkcjonalność. Test w tym momencie nie powinien się udać.
- Później następuje implementacja funkcjonalności. W tym momencie wcześniej napisany test powinien się udać.
- W ostatnim kroku programista dokonuje refaktoryzacji napisanego kodu, żeby spełniał on oczekiwane standardy.

Technika została stworzona przez Kenta Becka. Można jej też używać do poprawiania istniejącego kodu.

Bibliografia

- Wikipedia, Wikibooks na licencji CC.
- Joseph Albahari, Ben Albahari, C# 7.0 w pigułce, wyd. Helion, 2018.
- Krzysztof, Sopyła, Wykłady z ZPO, http://wmii.uwm.edu.pl/~ksopyla/, dostęp online 19.04.2019.
- https://www.geeksforgeeks.org/typeof-operator-keyword-in-c-sharp/, dostęp online 20.04.2019.
- http://kurs.aspnetmvc.pl/Csharp/Klasy_generyczne, dostęp online 20.04.2019.
- https://www.geeksforgeeks.org/c-sharp-generics-introduction/, dostęp online 20.04.2019.
- https://blogprogramisty.net/c-6-0-nowosci-slowo-kluczowe-nameof/, dostęp online 20.04.2019.

- https://www.plukasiewicz.net/Csharp_dla_zaawansowanych/Atrybuty, dostęp online 20.04.2019.
- Mark Michaelis, Eric Lippert, C# 6.0. Kompletny przewodnik dla praktyków. Wydanie V, wyd. Helion, 2016.