



na kierunku Informatyka

Narzędzie do zarządzania procesem kompilacji i wdrażania programów dla platformy .NET

Ernest Przestrzelski

Numer albumu 245544

Piotr Szyperski

Numer albumu 234508

promotor

dr Jan Bródka

WARSZAWA 2017

………………………… ………………………

podpis promotora podpis autora

STRESZCZENIE PRACY W JĘZYKU POLSKIM

STRESZCZENIE PRACY W JĘZYKU ANGIELSKIM

OŚWIADCZENIE O AUTORSTWIE

Spis treści

[1 Wstęp 8](#_Toc493951054)

[1.1 Automatyzacja kompilacji oraz innych procesów 8](#_Toc493951055)

[1.2 Cel pracy 8](#_Toc493951056)

[2 Wstęp teoretyczny 8](#_Toc493951057)

[2.1 C# 8](#_Toc493951058)

[2.2 Skrypty, inspiracje oraz podobne narzędzia 10](#_Toc493951059)

[2.2.1 Rake 10](#_Toc493951060)

[2.2.2 Fake 10](#_Toc493951061)

[2.2.3 CS-Script 10](#_Toc493951062)

[2.2.4 ScriptCS 11](#_Toc493951063)

[2.2.5 Cake 11](#_Toc493951064)

[2.2.6 Gulp.js 11](#_Toc493951065)

[2.3 MS Roslyn 11](#_Toc493951066)

[2.4 O naszym projekcie 12](#_Toc493951067)

[2.5 Skrypty i zadania w skryptach 12](#_Toc493951068)

[2.5.1 Implementacja 12](#_Toc493951069)

[2.5.2 Przykłady 13](#_Toc493951070)

[3 Dokumentacja techniczna pracy 16](#_Toc493951071)

[3.1 Uruchamianie programu 16](#_Toc493951072)

[3.1.1 Opcje uruchamiania 16](#_Toc493951073)

[3.1.2 Przykłady 16](#_Toc493951074)

[3.2 Załączanie skryptów i bibliotek, modularność i rozszerzalność 18](#_Toc493951075)

[3.2.1 Zenętrzne biblioteki 18](#_Toc493951076)

[3.2.2 Odnoszenie się do zewnętrznych skryptów 18](#_Toc493951077)

[3.3 Obsługiwanie skryptów 18](#_Toc493951078)

[3.4 Zapisywanie do logów 19](#_Toc493951079)

[3.4.1 Implementacja 19](#_Toc493951080)

[3.4.2 Konfiguracja Nlog 19](#_Toc493951081)

[3.4.3 Zapisywanie do logów bezpośrednio ze skryptu 22](#_Toc493951082)

[3.5 Budowanie projektu i solucji 23](#_Toc493951083)

[3.5.1 Implementacja 23](#_Toc493951084)

[3.5.2 Metody i opisy parametrów 23](#_Toc493951085)

[3.5.3 Przykłady 24](#_Toc493951086)

[3.6 Operacje na plikach i folderach 27](#_Toc493951087)

[3.6.1 Implementacja 27](#_Toc493951088)

[3.6.2 Metody i opis parametrów 27](#_Toc493951089)

[3.6.3 Przykłady 29](#_Toc493951090)

[3.7 Archiwa 32](#_Toc493951091)

[3.7.1 Implementacja 32](#_Toc493951092)

[3.7.2 Metody i opis parametrów 33](#_Toc493951093)

[3.7.3 Przykłady 34](#_Toc493951094)

[3.8 Nunit 37](#_Toc493951095)

[3.8.1 Implementacja 37](#_Toc493951096)

[3.8.2 Metody i opis parametrów 37](#_Toc493951097)

[3.8.3 Przykłady 39](#_Toc493951098)

[3.9 Xunit 42](#_Toc493951099)

[3.9.1 Implementacja 42](#_Toc493951100)

[3.9.2 Metody i opis parametrów 43](#_Toc493951101)

[3.9.3 Przykłady 44](#_Toc493951102)

[3.10 Git 45](#_Toc493951103)

[3.10.1 Implementacja 45](#_Toc493951104)

[3.10.2 Metody i opis parametrów 45](#_Toc493951105)

[3.10.3 Przykłady 47](#_Toc493951106)

[3.11 Minifikacja i złączanie plików 48](#_Toc493951107)

[3.11.1 Implementacja 48](#_Toc493951108)

[3.11.2 Metody i opis parametrów 48](#_Toc493951109)

[3.11.3 Przykłady 49](#_Toc493951110)

[3.12 Złożone skrypty 51](#_Toc493951111)

[4 Wnioski 56](#_Toc493951112)

[5 Bibliografia 57](#_Toc493951113)

[6 Wykaz symboli i skrótów 58](#_Toc493951114)

[7 Spis rysunków 59](#_Toc493951115)

# Wstęp

## Automatyzacja kompilacji oraz innych procesów

Podczas tworzenia oprogramowania, programista często zmuszony jest do wykonywania wielu powtarzalnych czynności, takich jak kompilacja plików źródłowych, uruchamianie testów, minifkacja plików przesyłanych przez sieć, itp. Procesy te zajmowały dużo czasu, przez swoją powtarzalność były nużące, a to z kolei mogło prowadzić do przeoczeń i błędów. Powstały więc programy ułatwiające życie programistom poprzez automatyzację takich procesów.

Wykorzystanie takiego programu najczęściej sprowadza się do napisania pliku reguł, bądź skryptu w określonym dla programu języku.

Jednym z najstarszych i najpopularniejszych narzędzi służących do automatyzacji procesu kompilacji programów jest **make**. Program ten, napisany w języku C, przetwarza specjalny plik reguł *Makefile* i na tej podstawie stwierdza, które pliki źródłowe wymagają kompilacji. Dzięki temu narzędziu programista nie musi kompilować pojedyńczych plików, a w przypadku zmiany pliku źródłowego make kompiluje tylko te pliki, które są od niego zależne.

Program make został stworzony w 1976 roku. Od tego czasu powstała niezliczona liczba programów w róznych językach automatyzujących pewne procesy, które programista wcześniej musiał wykonywać ręcznie. Niektóre z nich zostaną po krótce przedstawione w dalszej części pracy.

## Cel pracy

Kiedy wybieraliśmy temat naszej pracy, istniało kilka popularnych narzędzi automatyzujących przeznaczonych dla programistów platformy **.NET**. Mimo, że programy przeznaczone były do używania z tą konkretną platformą, w żadnym z nich pisanie skryptów nie było możliwe w języku **C#** - bodajże najpopularniejszym językiem używanym podczas pisania oprogramowania na platformę .NET. Nie istniała wtedy możliwość wykonania pojedyńczego pliku napisanego w C#.

W 2011 roku Microsoft opublikował projekt **Roslyn** (zostanie on omówiony w dalszej części pracy), którego niewielką częścią było API pozwalające wykonywanie dowolnych (oczywiście poprawnych) kawałków kodu napisanych w języku C#. Stworzyło to możliwość napisania programu, który korzystając z API Roslyn, dynamicznie kompiluje oraz wykonuje kod C#.

Za cel naszej pracy obraliśmy napisanie programu automatyzującego proces kompilacji oraz procesy związane z wdrażaniem programów przeznaczonych na platformę .NET, którego sposób działania kontrolowany jest przez programistę poprzez pisanie skryptów w C# - czyli języku, w którym najprawdopodobniej progamista .NET czuje się najpewniej.

Oprócz napisania samego programu, jako część naszej pracy napisaliśmy także kilka modułów odpowiedzialnych za wykonywanie niektórych z wcześniej wspomnianych powtarzalnych czynności. Jako cel postawiliśmy sobie także łatwą rozszerzalność możliwości naszego programu poprzez dodawanie nowych, wyspecjalizowanych modułów.

# Wstęp teoretyczny

## C#

C# (C Sharp, dosłownie „cis”, „c z krzyżykiem”) – obiektowy język programowania zaprojektowany w latach 1998-2001 przez zespół pod kierunkiem Andersa Hejlsberga dla firmy Microsoft.

Program napisany w tym języku kompilowany jest do języka Common Intermediate Language (CIL), specjalnego kodu pośredniegowykonywanego w środowisku uruchomieniowym takim jak .NET Framework, Mono lub DotGNU. Wykonanie skompilowanego programu przez system operacyjny bez takiego środowiska nie jest możliwe.

Nazwa języka powstała analogicznie jak nazwa C++. Operator ++ w C (w C++ też) oznacza zwiększenie o jeden, więc C++ to więcej niż C. Autorzy C# wykorzystali podobny pomysł, gdzie symbol krzyżyka/kratki przypomina dwa połączone ze sobą operatory ++. W muzyce dźwięk C♯ jest wyższy niż dźwięk C, może to sugerować, że język jest rozwinięciem C/C++.

Symbol ♯ (krzyżyk) jest niedostępny na większości klawiatur, niektórych czcionkach i przeglądarkach internetowych, dlatego zalecane jest używanie symbolu kratki (#), który bardzo przypomina symbol muzyczny. Czasami tam, gdzie ma to uzasadnienie np. w materiałach promocyjnych, Microsoft używa nazwy C♯ zamiast C#.

Język C# ma wiele cech wspólnych z językami programowania Object Pascal, Delphi, C++ i Java.

* Obiektowość z hierarchią o jednym elemencie nadrzędnym: podobnie jak w Javie, kod programu jest zbiorem klas. W C# podobnie tak jak w Javie/Object Pascalu hierarchia dziedziczenia opiera się na istnieniu jednej klasy object (System.Object), która stanowi element nadrzędny tej hierarchii. W szczególności oznacza to, że również typy proste (int, double, itd.) są strukturami z właściwymi sobie metodami, np. int i = 1; string s = i.ToString();.
* Odśmiecanie pamięci: zarządzaniem pamięcią zajmuje się środowisko uruchomieniowe. Oznacza to, że nie ma potrzeby samodzielnego zajmowania się zwalnianiem pamięci po obiektach, które przestają być używane.
* Właściwości, indeksery: dodatkowe elementy składowe klas.
* Delegaty, zdarzenia: odpowiedniki i rozwinięcie wskaźników na funkcje z C++.
* Refleksje i atrybuty klas: w czasie pracy programu istnieje możliwość analizy struktury kodu z poziomu tego kodu. Umożliwia to tworzenie wysoce uniwersalnych mechanizmów operujących na strukturze kodu nieznanej w czasie kompilacji. Mechanizm ten wykorzystywany jest m.in. w bibliotekach ORM, narzędziach do analizy i weryfikacji kodu czy rozszerzeniach AOP. Mechanizm atrybutów został z C# zapożyczony do języka Java w wersji 1.5 (adnotacje), jakkolwiek samo Reflection API istniejące od pierwszego wydania języka stanowiło inspirację dla twórców C#.
* Typy ogólne (generics – dostępne od wersji .NET 2.0): mechanizm zbliżony swoją ogólnością do szablonów w C++, jednak tu typ ogólny jest przenoszony do modułu binarnego i możliwy jest do wykorzystania bez konieczności posiadania kodu źródłowego.
* Dynamiczne tworzenie kodu: biblioteki .NET umożliwiają dynamiczne tworzenie kodu w czasie działania programu i włączanie go do kodu aktualnie wykonywanego. Możliwe jest zarówno dynamiczne tworzenie kodu wykonywalnego ze źródeł C# jak i tworzenie dynamicznych modułów w języku pośrednim (CIL).
* Bogata biblioteka klas BCL (Base Class Library), umożliwiająca rozwijanie aplikacji konsolowych, okienkowych (System.Windows.Forms oraz WPF (Windows Presentation Foundation) od .NET Framework 3.0), bazodanowych (ADO.NET), sieciowych (System.Net), w architekturze rozproszonej (WebServices) czy dynamicznych aplikacji internetowych (ASP.NET) oraz dynamiczne treści multimedialne (Silverlight).
* W C#, podobnie jak w C stosuje się dwa rodzaje komentarzy:
  + // to komentarz jednoliniowy, czyli wszystkiego, co znajduje się za nim, kompilator nie bierze pod uwagę.
  + Komentarz wielowierszowy zaczyna się od /\* a kończy \*/. Kompilator nie bierze pod uwagę tego, co znajduje się pomiędzy.

Jeśli komentarz zaczyna się od /// lub /\*\*, to jest to komentarz, w którym umieszcza się dokumentację do kodu.

## Skrypty, inspiracje oraz podobne narzędzia

### Rake

Program komputerowy automatyzujący proces kompilacji programów napisanych w języku Ruby. Twórcą Rake jest Jim Weirich. Pozwala użytkownikowi zdefiniować zadania i opisać zależności między nimi oraz grupować zadania w przestrzeń nazw. Jest odpowiednikiem programów make czy SCons z wieloma istotnymi różnicami. Pliki "Rakefile" są odpowiednikiem plików "Makefile" w make, lecz używają składni języka Ruby – anonimowych bloków funkcji. Posiada również bibliotekę najbardziej pospolitych zadań, np. do zarządzania plikami. Rake jest częścią standardowej biblioteki w Ruby od wersji 1.9.

### Fake

FAKE – F# Make – jest międzyplatformowym narzędziem automatyzującym proces kompilacji. Z powodu integracji w F# wszystkie korzyści .NET framework i programowania funkcjonalnego mogą być wykorzystane, w tym ekstensywna biblioteka klas, potężne debuggery i zintegrowane środowiska programistyczne, takie jak Microsoft Visual Studio czy MonoDevelop, które udostępniają podkreślanie składni i automatyczne uzupełnianie kodu. Ten DSL (Domain-specific Language) został zaprojektowany tak, aby był zwięzły, typowany, deklaratywny, rozszerzalny i łatwy w użyciu.

### CS-Script

Jest to oparte na CLR (Common Language Runtime) narzędzie do zarządzania skryptami, które używa zgodnego ze standardami ECMA (European Computer Manufacturers Association) języka C# jako języka programistycznego. Aktualnie, CS-Script mierzy w implementację Microsoftu (.NET 2.0/3.0/3.5/4.0/4.5) z całkowitym wsparciem dla Mono. Jest narzędziem z dostępnym kodem źródłowym za darmo (open-source), na licencji typu MIT. Dzięki statycznemu typowaniu ten język umożliwia nieograniczony dostęp do funkcjonalności .NET/CLR oraz prezentuje brak spadku wydajności w porównaniu ze skompilowanymi binariami. CS-Script to środowisko do zarządzania skryptami, które oferuje uruchamianie pojedynczych skryptów oraz udostępnianie silnika pracującego ze skryptami z aplikacji CLR.

### ScriptCS

ScriptCS sprawia, że możemy w łatwy sposób pisać i uruchamiać kod w C# przy użyciu prostego edytora tekstowego. Pomimo, że Microsoft Visual studio i inne zintegrowane środowiska programistyczne są bardzo potężnymi narzędziami, często mogą spowalniać pracę, zamiast ją przyspieszać. Nie zawsze musisz tworzyć nowy projekt czy solucję, czasami masz ochotę tylko włączyć swój ulubiony edytor tekstowy i zacząć pisać. ScriptCS również używa narzędzia firmy Microsoft „Roslyn” do uruchamiania kodu napisanego w języku C#. Poza tym możemy posłużyć się narzędziem NuGet do zarządzania zależności, a luźna składnia skryptu napisanego w tym języku pozwala na napisanie i uruchomienie aplikacji w obrębie jednej linii kodu. Pakiety skryptów pozwalają na załadowanie środowiska do nowych skryptów, co dodatkowo zmniejsza ilość kodu potrzebnego do korzystania z ulubionych frameworków C#.

### Cake

Cake (C# Make), podobnie jak „Fake”, jest międzyplatformowym narzędziem automatyzującym proces kompilacji. Największą różnicą jest to, że używa DSL C# do kompilowania kodu, kopiowania plików/folderów, uruchamiania testów jednostkowych, kompresowania plików, budowania paczek NuGetowych i nie tylko. Zbudowane jest na kompilatorych Roslyn i Mono, co umożliwia pisanie skryptów w języku C#. Narzędzie jest rzetelne i zachowuje się zawsze w ten sam sposób, niezależnie czy używasz go na własnej maszynie, czy przy użyciu TeamCity, TFS (Team Foundation Server) czy Jenkins. Wspiera najpopularniejsze narzędzia przy procesie kompilacji, takie jak MSBuild, MSTest, xUnit, NUnit, NuGet, itp. Ten system jest dostępny na otwartej licencji (open-source), a kod można znaleźć na GitHub.

### Gulp.js

Gulp.js to zestaw narzędzi JavaScript na otwartej licencji wykorzystywany do automatyzacji procesem budowania w rozwoju aplikacji po stronie klienta. Został zbudowany na bazie Node,js i npm (Node Package Manager), a jego zadaniem jest uruchamianie czynności, które zwykle zajmują dużo czasu i są powtarzalne podczas rozwoju stron internetowych. Przykładami mogą być: minifikacja, konkatenacja, testy jednostkowe, optymalizacja, itp. Gulp.js wykorzystuje podejście skupiające się bardziej na kodzie niż na konfiguracji w celu zdefiniowania zadań i zależy od wielu małych wtyczek, które zajmują się osobnymi czynnościami. Cały ekosystem posiada ponad 300 wtyczek, z których możemy wybierać według naszych potrzeb.

## MS Roslyn

„.NET Compiler Platform”, platforma lepiej znana pod kryptonimem „Roslyn”, zawiera zestaw kompilatorów open-source i API (Application Programming Interface) do analizy kodu stworzone przez firmę Microsoft. Kompilatory C# i Visual Basic .NET napisane są właśnie w tych językach. Mamy do nich dostęp poprzez tradycyjne programy z linii wiersza poleceń, ale również poprzez API - natywnie z poziomu kodu .NET. Roslyn udostępnia moduły do analizy leksykalnej (syntaktycznej) kodu, semantycznej analizy, dynamicznego kompilowania do CIL i do emisji kodu.

W naszej pracy wykorzystujemy bardzo niewielką część projektu „Roslyn”, a właściwie wąską funkcjonalność jednego z API do obsługi skryptów C#, który to udostępnia możliwość instancjonowania silnika C# i uruchamiania kodu ze skryptu napisanego w tym języku. Oprócz tego wykorzystujemy kilka klas z biblioteki Microsoft.CodeAnalysis, w szczególności Microsoft.CodeAnalysis.MSBuild, do budowania projektów i solucji w naszym komponencie Cake.Build.

Więcej informacji oraz kod źródłowy „.NET Compiler Platform” można znaleźć na stronach platformy GitHub i Wikipedia.

## O naszym projekcie

Nazwa „Cake” powstała po złączeniu C# + make i została stworzona na wzór rake (ruby make) i fake (f# make). Chcieliśmy zgrabnie ująć esencję naszego projektu oraz do czego będzie on służył. Utwierdziliśmy się w tej nazwie, gdy po długotrwałej pracy nad naszym rozwiązaniem zobaczyliśmy inny projekt zatytułowany identycznie.

Gdy zaczynaliśmy pisać naszą aplikację, zauważyliśmy, że na rynku brakuje darmowego, prostego w obsłudze narzędzia, które pozwoli na konfigurację kolejnych kroków budowania projektów, przygotowywania paczek z nowymi wersjami aplikacji gotowych pod „deploy” na dowolne środowisko, sprawdzania testów jednostkowych oraz podobnych procesów. Nawet gdy mogliśmy znaleźć jakieś rozwiązanie, spełniające niektóre wyżej wymienione wymogi, to nie potrafiliśmy natrafić na narzędzie, które byłoby łatwo rozszerzalne o nowe moduły.

Nasz program ma służyć jako ułatwienie w codziennej pracy z procesami przy zarządzaniu kompilacją i wdrażaniem programów dla platformy .NET. Aplikacja „Cake” będzie właśnie pomagała w budowaniu projektów i solucji napisanych przy użyciu środowiska Microsoft Visual Studio, operacjach na plikach i folderach, uruchamianiu testów jednostkowych oraz pracowaniu z systemem kontroli wersji Git. Jak można zauważyć nasz system ma wiele zastosowań.

## Skrypty i zadania w skryptach

### Implementacja

Skrypty, które obsługuje nasz program, składają się zwykle z różnorakich zadań (reprezentowanych przez klasę Job), polegających właśnie na definiowaniu operacji zarządzających wdrażaniem aplikacji oraz zależności między tymi zadaniami.

Zadania (w naszej aplikacji reprezentowane przez klasę Job) są zwykle główną częścią skryptu. Ważnym założeniem jest to, że zadania muszą być idempotentne – to jest kilkukrotne wykonanie zadania przyniesie taki sam skutek, jak wykonanie jednokrotne.

Zadanie może posiadać akcję, zero lub więcej zależności (zadań potrzebnych do wykonania się) i musi posiadać unikatową w obrębie skryptu nazwę.

### Przykłady

Poniżej znajduje się skrypt z jednym zadaniem nazwanym “FirstJob”. Jego akcja zdefiniowana jest jako argument metody Does(Action jobAction). Zadanie jest wywoływane metodą SetDefault(string jobName).

Przykład 1



Rysunek 1

**

Rysunek 2

Na powyższym zrzucie ekranu można zauważyć szczegółowe dane wypisane podczas działania programu Cake ze skryptem z pierwszego przykładu.

W przypadku zdefiniowania zależności na zadaniu poprzez metodę DependsOn(string otherJobs), program zagwarantuje, że zadania-zależności zostaną wykonane w pierwszej kolejności. Parametr otherJobs jest listą zadań, od których nasze zadanie jest zależne.

Poniżej znajduje się skrypt z dwoma zadaniami. Zadanie “Deploy” jest zależne od zadania “Test”.

Przykład 2



Rysunek 3

**

Rysunek 4

Uruchomienie zadania “Deploy” spowoduje, że program najpierw wykona zadanie “Test”, a dopiero potem przejdzie do „Deploy”.

Przykład 3



Rysunek 5

**

Rysunek 6

Na powyższym zrzucie ekranu możemy zauważyć rezultat skryptu z trzeciego przykładu, natomiast szczegółowość logów została ograniczona do minimalnego poziomu „Info”. Kolejność wywołań zadań w powyższym skrypcie jest następująca: „ThirdJob”, „SecondJob”, „FirstJob”. Każde z zadań będzie wykonane tylko raz, gdyż są one idempotentne i kolejne wywołania „ThirdJob” byłyby bezcelowe i czasochłonne.

# Dokumentacja techniczna pracy

## Uruchamianie programu

### Opcje uruchamiania

Nasza aplikacja powinna być uruchamiana z wiersza poleceń w następujący sposób:

Przykład 4

X:\PathTo\Cake>Cake.exe /s “Z:\Path\Script.csx” /av Info /sv Trace /r YourJobToRun

Jak widać w powyższym przykładzie, przy uruchamianiu programu możliwe jest wykorzystanie jednego lub więcej poniższych parametrów:

* /script (/s) - lokalna ścieżka do skryptu C#
* /runjob (inaczej /r) - zadanie do uruchomienia zdefiniowane w skrypcie c# - niezbędne, jeśli w skrypcie brakuje metody JobManager.SetDefault
* /scriptverbosity (/sv) - poziom zapisywania do logów w podanym skrypcie (możliwy jeden z sześciu: Trace, Debug, Info, Warn, Error, Fatal)
* /appverbosity (/av) – poziom zapisywania do logów w aplikacji (możliwy jeden z sześciu: Trace, Deug, Info, Warn, Error, Fatal)
* /help (/h) – pomoc

Parametry te są przetwarzane w głównej klasie naszej aplikacji - Program.cs. Główna metoda w tej klasie po sczytaniu parametrów próbuje uruchomić skrypt przy użyciu klasy RoslynEngine.cs, opisanej poniżej (rozdział 2.3.).

### Przykłady

Na poniższym zrzucie ekranu program został uruchomiony z konfiguracją logów na poziomie minimalnym „Trace”, ale w programie podaliśmy minimalny poziom „Error” – dzięki temu nie widzimy wiadomości zawartych w skrypcie poniżej tego poziomu (logi z programu dalej znajdują się na konsoli, gdyż sekcja konfiguracyjna NLog nie została zmieniona).

Przykład 5

**

Rysunek 7

Przykład 6

**

Rysunek 8

Powyższy zrzut ilustruje uruchamianie tylko niektórych zadań ze skryptu podanego w przykładzie 3. Można zauważyć, że zostały zarejestrowane trzy zadania, ale uruchomione tylko dwa, gdyż wyszczególniliśmy w argumencie, od którego zadania mamy zacząć skrypt.

## Załączanie skryptów i bibliotek, modularność i rozszerzalność

### Zenętrzne biblioteki

Oprócz bibliotek załączonych w programie CAKE można dodawać referencje do dowolnego pliku .dll ze skompilowanym kodem C# i wykorzystywać klasy i metody w nim zawarte bezpośrednio w skrypcie. Jest to prosty sposób na rozszerzalność naszego systemu o dodatkowe moduły, których funkcjonalności nie przewidzieliśmy w naszej aplikacji. W ten sam sposób powinniśmy odnosić się do bibliotek naszego systemu, aby skorzystać z danego modułu. Tym samym zapewniamy modularność naszemu rozwiązaniu.

Aby odnieść się do danej biblioteki należy to zrobić na samym początku skryptu w następujący sposób:

Przykład 7



Rysunek 9

Pierwsza linijka skryptu mówi nam o potencjalnych bibliotekach do załadowania. Ścieżka może być względna lub bezwzględna. Przestrzenie nazw znajdujące się w dołączanym pliku będą również zaimportowane.

### Odnoszenie się do zewnętrznych skryptów

W podobny sposób możemy dodawać referencje do innych skryptów, które chcemy załadować, zanim wykonamy nasz kod C#. Jest to przydatna funkcjonalność dla ludzi lubiących trzymać małe skrypty obsługujące pojedynczą funkcjonalność oraz do separacji czynności, co również wpływa na modularność rozwiązania.

Przykład 8



Rysunek 10

## Obsługiwanie skryptów

Nasza aplikacja obsługuje skrypty z pomocą dwóch klas - RoslynEngine oraz JobManager. Ta pierwsza posiada tylko jedną publiczną metodę – ExecuteFile, który wczytuje odpowiednie referencje (biblioteki i inne skrypty) oraz uruchamia nasz docelowy skrypt c#. Menedżer zadań obsługuje rejestrację zadań, ustawianie zależności między zadaniami oraz egzekwowanie kodu.

## Zapisywanie do logów

### Implementacja

Aby poinformować użytkownika co dzieje się w naszej aplikacji wykorzystujemy platformę NLog. Jest to narzędzie przeznaczone do użytku z .NET, Xamarin, Silverlight i Windows Phone. NLog udostępnia bogate możliwości zarządzania zapisywaniem informacji do logów. Bez względu na wielkość i złożoność aplikacji ta platforma produkuje i utrzymuje wysokiej jakości logi z jakiegokolwiek języka .NET (C#, VB.NET, itp.), które możemy rozszerzyć o kontekstowe informacje (data i czas, poziom, wątek, proces czy środowisko danego zdarzenia) oraz sformatować zgodnie z upodobaniem, a następnie wysłać do jednego lub więcej miejsc.

Dzięki narzędziu NLog mamy w naszej aplikacji 6 poziomów wiadomości: trace, debug, info, warn, error i fatal. Szczegółowość tą wyszczególnia się w pierwszym argumencie metody Log(LogLevel level, string message). Jak można łatwo się domyślić, poziom trace jest wykorzystywany do notowania najbardziej szczegółowych informacji, które są mało ważne z punktu widzenia użytkownika, zaś poziom fatal informuje nas o drastycznych błędach w aplikacji.

### Konfiguracja Nlog

Decydowanie o poziomie wpisywania do logów i wypisywania danych na konsolę polega na edycji pliku Cake.exe.config, a mianowicie sekcji <nlog> w tym pliku. Znajduje się on w folderze z plikiem wykonywalnym aplikacji.

Oto przykładowa sekcja z konfiguracją narzędzia:

Przykład 9



Rysunek 11

Konfiguracja NLog składa się z 3 podstawowych sekcji: targets, rules i variable.

#### </targets>

Jest sekcją wymaganą. To właśnie w niej konfigurujemy miejsce zapisywania logów (czy informacja ma być zapisana w pliku, w bazie danych, wyświetlona na konsoli, czy może ma być ona wysłana przez sieć). Istnieją dwa typy celów - pierwszy otrzymuje i obsługuje wiadomości, drugi buforuje lub przekierowuje wiadomości do innego celu.

Istnieje wiele możliwości zdefiniowania celu:

Typ pierwszy:

* Chainsaw – Wysyła logi do zdalnej instancji aplikacji Chainsaw od log4j.
* ColoredConsole – Wypisuje logi na konsolę z możliwością dostosowania kolorów.
* Console – Wypisuje logi na konsolę.
* Database – Wpisuje logi do bazy danych przy użyciu ADO.NET provider.
* Debug – Udawany cel, przydatny przy testowaniu.
* Debugger – Wypisuje wiadomości do podpiętego debuggera.
* EventLog – Wypisuje wiadomości do logu ze zdarzeniami.
* File – Zapisuje wiadomości do jednego lub więcej plików.
* LogReceiverService – Wysyła wiadomości do serwisu NLog Receiver (przy użyciu WCF (Windows Communication Foundation) lub Web Services).
* Mail – Wysyła wiadomości przez email z użyciem protokołu SMTP (Simple Mail Transfer Protocol).
* Memory – Zapisuje wiadomośći do ArrayList w pamięci, w celu późniejszego dostępu w kodzie.
* MethodCall – Wywołuje statyczną metodę dla każdej wiadomości i przekazuje do niej kontekstowe parametry.
* Network – Wysyła logi przez sieć.
* NLogViewer – Wysyła wiadomości do zdalnej instancji NLog Viewer.
* Null – Ignoruje logi. Używane przeważnie do debuggowania oraz benchmarkingu.
* OutputDebugString – Produkuje wiadomości poprzez OutputDebugString() Win32 API.
* PerfCounter – Zwiększa licznik wydajności przy każdym zapisie.
* Trace – Wysyła wiadomości przez System.Diagnostics.Trace.
* WebService – Wywołuje web service na każdej wiadomości.
* Typ drugi (wrappery):
* AsyncWrapper – Zapewnia asynchroniczne, buforowane egzekwowanie zapisu do logów.
* AutoFlushWrapper – Powoduje czyszczenie strumienia po każdym zapisie na celu.
* BufferingWrapper – Buforuje zdarzenia i wysyła w paczkach do celu. Przydatne razem z celem „Mail”.
* FallbackGroup – Zapewnia obsługę awaryjności.
* FilteringWrapper - Filtruje wiadomości na podstawie warunków.
* ImpersonatingWrapper – Podszywa się pod innego użytkownika podczas zapisu.
* LimitingWrapper – Ogranicza liczbę zdarzeń wysyłanych do celu.
* PostFilteringWrapper – Filtruje zbuforowane logi na podstawie warunków narzuconych na grupę zdarzeń.
* RandomizeGroup – Wysyła wiadomości do losowo wybranego celu.
* RepeatingWrapper – Powtarza każde zdarzenie wybraną liczbę razy.
* RetryingWrapper – Ponawia probe zapisu w razie błędu.
* RoundRobinGroup – Rozsyła wiadomości do celów na sposób “każdy z każdym – parami”.
* SplitGroup – Zapisuje zdarzenia do wszystkich celów.

Każda z sekcji posiada własne atrybuty, które pomagają nam w jej konfiguracji.



Powyższy przykład archiwizuje pliki, gdy ich wielkość przekroczy wartość pola „archiveAboveSize” wyrażonego w bajtach. Z kolei ustawienie pola „archiveNumbering” na wartość „Rolling” powoduje zapis najnowszego pliku archiwum pod nazwą „log.00000.txt” i przesunięcie już istniejących o wartość 1.

Sekcja <target> oferuje jednak o wiele więcej. To w niej możemy dodać atrybut „layout”, w którym definiujemy w jakim formacie chcemy logować informacje. Domyślny format, który niektórym z nas może być wystarczający wygląda następująco:



Wykaz wszystkich możliwych zmiennych:

* ${activityid}
* ${all-event-properties}
* ${appdomain}
* ${assembly-version}
* ${basedir}
* ${callsite}
* ${callsite-linenumber}
* ${counter}
* ${date}
* ${document-uri}
* ${environment}
* ${event-properties}
* ${exception}
* ${file-contents}
* ${gc}
* ${gdc}
* ${guid}
* ${identity}
* ${install-context}
* ${level}
* ${literal}
* ${log4jxmlevent}
* ${logger}
* ${longdate}
* ${machinename}
* ${mdc}
* ${message}
* ${ndc}
* ${ndlc}
* ${newline}
* ${nlogdir}
* ${performancecounter}
* ${processid}
* ${processinfo}
* ${processname}
* ${processtime}
* ${qpc}
* ${registry}
* ${shortdate}
* ${sl-appinfo}
* ${specialfolder}
* ${stacktrace}
* ${tempdir}
* ${threadid}
* ${threadname}
* ${ticks}
* ${time}
* ${var}
* ${windows-identity}
* Wrappery:
  + ${cached}
  + ${filesystem-normalize}
  + ${json-encode}
  + ${lowercase}
  + ${onexception}
  + ${pad}
  + ${replace}
  + ${replace-newlines}
  + ${rot13}
  + ${trim-whitespace}
  + ${uppercase}
  + ${url-encode}
  + ${when}
  + ${whenEmpty}
  + ${WrapLine}
  + ${xml-encode}

#### <rules/>

Jest drugą i ostatnią wymaganą sekcją konfiguracji NLog’a, w której ustawiamy przepływ/warunki logowania. Elementy w tej sekcji noszą nazwę <logger/> i akceptują poniższe atrybuty:

* name – źródło (nazwa) logger’a (np. „Program. NLogProject”, „\*”)
* minlevel – minimalny poziom logu, dla którego pasuje ta reguła
* maxlevel – maksymalny poziom logu, dla którego pasuje ta reguła
* level  – pojedynczy poziom logu, dla którego pasuje reguła
* levels – lista poziomów logu oddzielona przecinkiem, dla których pasuje reguła
* writeTo – lista celów oddzielona przecinkami, które będą użyte podczas reguły
* final – flaga indykująca, że żadna inna reguła nie będzie wykonana, gdy obecna reguła pasuje

Jeżeli chcemy zapisywać wszystkie możliwe wiadomości do naszego pliku, nasza reguła wyglądałaby tak:



Jeżeli jednak chcemy zapisać wiadomości o poziomie Warn, Error i Fatal ze wszystkich klas z namespace NLogProject wystarczy dodać taki wpis:



#### <variable/>

Sekcja ta służy do definiowania zmiennych używanych w dalszej definicji NLog’a.



### Zapisywanie do logów bezpośrednio ze skryptu

W naszej aplikacji istnieje możliwość zapisywania wiadomości bezpośrednio ze skryptu, tak jak w poniższym skrypcie:

Przykład 10



Rysunek 12

Takie zadanie zapisze wiadomość do pliku, którego ścieżka i nazwa zdefiniowane są w pliku konfiguracyjnym Cake.exe.config. Domyślnie cel nazwany jest „Script”, lecz można zdefiniować własną nazwę do zapisywania wiadomości i różne zasady w sekcji <nlog> pliku konfiguracyjnego, zgodnie z wcześniej opisanymi zasadami, np.

Przykład 11



Rysunek 13

## Budowanie projektu i solucji

### Implementacja

Do zadań związanych z budowaniem projektów i solucji korzystamy z biblioteki Microsoft.CodeAnalysis. W tej części naszego projektu wykorzystujemy klasę MSBuildWorkspace z przestrzeni nazw Microsoft.CodeAnalysis.MSBuild, aby stworzyć przestrzeń roboczą do wczytania solucji czy projektu MSBuild. Następnie używamy na każdym projekcie z tej przestrzeni metody GetCompilationAsync, aby sprawdzić rezultat kompilacji oraz go emitować do odpowiednich plików - .dll (Dynamic-Link Library), .pdb (Program database), czy .xml (Extensible Markup Language). Wszystkie te działania obudowane są w zapisywanie do logów, sprawdzanie argumentów oraz obsługę wyjątków.

### Metody i opisy parametrów

#### bool BuildProject(string projectFile, string outputPath = null, string configuration = "Debug", string platform = "Any CPU")

Metoda pozwala na zbudowanie jednego lub więcej projektów. Zwraca flagę indykującą czy proces kompilacji powiódł się dla wszystkich projektów.

Parametry:

* projectFile – ścieżka z nazwą pliku projektu. Może zawierać znaki wieloznaczne (wildcard characters).
* configuration – konfiguracja buildu. Możliwe wartości: „Debug” i „Release”. Wartość domyślna to „Debug”.
* platform – platforma buildu. Możliwe wartości: „x86”, „x64”, „Any CPU”. Wartość domyślna to „Any CPU”.
* outputPath – ścieżka do katalogu wyjściowego z rezultatem kompilacji. Jeśli pozostanie pusta, pliki zostaną zapisane do „./bin/configuration” (configuration – wartość z parametru).

#### bool BuildSolution(string solutionFile, string outputPath = null, string configuration = "Debug", string platform = "Any CPU")

Metoda pozwala na zbudowanie jednej lub więcej solucji. Zwraca flagę indykującą czy proces kompilacji powiódł się dla wszystkich solucji.

Parametry:

* solutionFile – ścieżka z nazwą pliku solucji. Może zawierać znaki wieloznaczne (wildcard characters).
* configuration – konfiguracja buildu. Możliwe wartości: „Debug” i „Release”. Wartość domyślna to „Debug”.
* platform – platforma buildu. Możliwe wartości: „x86”, „x64”, „Any CPU”. Wartość domyślna to „Any CPU”.
* outputPath – ścieżka do katalogu wyjściowego z rezultatem kompilacji. Jeśli pozostanie pusta, pliki zostaną zapisane do „./bin/configuration” (configuration – wartość z parametru)

### Przykłady

Przykład 12



Rysunek 14

**

Rysunek 15

Powyższy skrypt skompiluje projekt z lokalizacji podanej w parametrze oraz wypisze błędy kompilacji, jeśli jakieś nastąpią. Zrzut ekranu pokazuje przykładowy wynik kompilacji z błędami.

Przykład 13



Rysunek 16

**

Rysunek 17

Powyższy przykład poprawnie skompiluje projekt z lokalizacji podanej w lokalizacji. Zrzut ekranu pokazuje przykładowy wynik kompilacji z błędami.

Przykład 14

****

Rysunek 18

**

Rysunek 19

## Operacje na plikach i folderach

### Implementacja

Do operacji na plikach i folderach wykorzystujemy metody z przestrzeni nazw „System.IO”, a do wyszukiwania wzorców „System.Text.RegularExpressions”. Podstawowe operacje obudowaliśmy w sprawdzanie warunków wejściowych oraz odpowiednie zapisywanie do logów.

### Metody i opis parametrów

#### bool CopyDirectory(string sourceDir, string destinationDir, bool copySubDirs = true, bool overwrite = false, bool cleanDestinationDirectory = false)

Metoda kopiuje katalog z jednego miejsca w drugie. Zwraca flagę indykującą czy kopiowanie się powiodło.

Parametry

* sourceDir – ścieżka do katalogu źródłowego.
* destinationDir – ścieżka do katalogu docelowego.
* copySubDirs – flaga indykująca czy kopiować podkatalogi. Wartość domyślna to „true”.
* overwrite – flaga indykująca czy pliki w katalogu docelowym powinny zostać nadpisane. Wartość domyślna to „false”.
* cleanDestinationDirectory – Flaga indykująca czy pliki znajdujące się w katalogu docelowym przed operacją powinny zostać usunięte. Wartość domyślna to „false”.

#### • bool CopyFile(string sourceName, string destName, bool overwrite = true)

Metoda kopiuje plik z jednego miejsca w drugie. Zwraca flagę indykującą czy kopiowanie się powiodło.

Parametry

* sourceName – ścieżka i nazwa pliku źródłowego
* destName – ścieżka docelowa
* overwrite – flaga indykująca czy nadpisać plik w miejscu docelowym. Wartość domyślna to „true”.

#### bool DeleteFile(string filePath)

Metoda usuwa plik z podanego miejsca. Zwraca flagę indykującą czy usuwanie się powiodło.

Parametry

* filePath – ścieżka i nazwa pliku do usunięcia.

#### String[] GetFilesWithPattern(string parentDirectoryPath, string filePattern, bool subdirectories = false)

Metoda zwraca pełne ścieżki do wszystkich plików pasujących do wzorca z podanego katalogu.

Parametry:

* parentDirectoryPath – ścieżka do katalogu, w którym chcemy odnaleźć pliki.
* filePattern – wzorzec dopasowania do plików.
* subdirectories – flaga indykująca czy przeszukiwać również podkatalogi. Wartość domyślna to „false”.

#### bool DeleteFilesWithPattern(string parentDirectoryPath, string filePattern)

Metoda usuwa pliki pasujące do wzorca z podanego katalogu. Zwraca flagę indykującą czy usuwanie się powiodło.

Parametry:

* parentDirectoryPath – ścieżka do katalogu, z którego chcemy usunąć pliki.
* filePattern – wzorzec dopasowania do plików.

#### bool DeleteDirectoriesWithPattern(string parentDirectoryPath, string directoryPattern, bool subdirectories = false)

Metoda usuwa podkatalogi pasujące do wzorca z podanego katalogu. Zwraca flagę indykującą czy usuwanie się powiodło.

Parametry:

* parentDirectoryPath – ścieżka do katalogu, z którego chcemy usunąć podkatalogi.
* directoryPattern – wzorzec dopasowania do katalogów.
* subdirectories – flaga indykująca czy usuwać podkatalogi rekurencyjnie.

#### bool DeleteDirectory(string directoryPath)

Metoda usuwa katalog z podanego miejsca. Zwraca flagę indykującą czy usuwanie się powiodło.

Parametry:

* directoryPath – ścieżka do katalogu, który chcemy usunąć.

#### bool CleanDirectory(string directoryPath)

Metoda czyści podany katalog z plików i folderów. Zwraca flagę indykującą czy cała zawartość została usunięta.

Parametry

* directoryPath – ścieżka do katalogu, który chcemy wyczyścić.

#### bool ReplaceText(string filePath, string regex, string newText)

Metoda zamienia w pliku tekst pasujący do wzorca, na inny tekst. Zwraca flagę indykującą czy plik został poprawnie nadpisany.

Parametry:

* filename – nazwa pliku, po której chcemy przeszukiwać katalogi.
* directories – ścieżki do katalogów, które chcemy przeszukać, oddzielone przecinkami.

### Przykłady

Przykład 15

****

Rysunek 20

**

Rysunek 21

Powyższy przykład ilustruje wykorzystania metody CopyDirectory do skopiowania katalogu z jednego miejsca do drugiego. Obie ścieżki, zarówno źródłowa, jak i docelowa, podane są w parametrach metody.

Przykład 16



Rysunek 22

**

Rysunek 23

Skrypt z przykładu 16. Wyszuka wszystkie pliki z katalogu w parametrze, których nazwa kończy się na „cript.csx”, a początek nazwy może być dowolny oraz wypisze rezultat na konsolę.

Przykład 17



Rysunek 24

**

Rysunek 25

Rezultat skryptu z przykładu 17. widać na powyższym zrzucie ekranu. Wszystkie wystąpienia „Log” zostały zamienione wyrażeniem „newLog”.

## Archiwa

### Implementacja

Udostępniamy funkcjonalność związaną z kompresją plików i folderów, a także innymi operacjami na archiwach w module Cake.Zip z pomocą biblioteki DotNetZip (IonicZip.dll).

Jest to mała, łatwa w użyciu biblioteka klas do manipulowania plikami .zip. Daje możliwość aplikacjom .NET napisanym w VB.NET, C# lub innym języku .NET łatwego tworzenia, czytania i aktualizowania plików .zip. Dostępna jest też kompresja i dekompresja GZIP, ZLIB, Deflate, czy biblioteka do obsługi BZip2, narzędzie z graficznym interfejsem użytkownika ZIP oraz kilka narzędzi wiersza poleceń.

DotNetZip działa na komputerach z systemem Windows oraz frameworkiem .NET, ale też urządzeniach z Windows Mobile, które używają .NET Compact Framework. Biblioteka może być używana również ze środowisk COM, takich jak PHP, klasyczne ASP czy VBScript. Wspiera poniższe scenariusze:

* Tworzenie archiwum, dodawanie plików i katalogów do archiwum.
* Wypisanie plików z archiwum, wypakowanie plików z archiwum.
* Modyfikowanie istniejącego archiwum – zmiana nazwy wpisu, usuwanie wpisu z Archiwum czy dodawanie nowych wpisów do archiwum.
* Tworzenie plików .zip lub zaczytywanie ze strumienia danych, zapisywanie czy wypakowywanie do strumienia danych.
* Dynamiczne tworzenie plików .zip z aplikacji ASP.NET czy Silverlight.

### Metody i opis parametrów

#### bool ZipFiles(string zipPathAndName, params string[] filePaths)

Metoda dodaje pliki do archiwum w formacie „.zip”. Zwraca flagę indykującą czy spakowanie plików się powiodło.

Parametry:

* zipPathAndName – ścieżka i nazwa archiwum docelowego
* filePaths – ścieżki do plików, które chcemy dodać do archiwum.

#### bool ZipFilesWithOptions(string zipPathAndName, string password = null, string compression = null, bool aes256Encryption = false, bool useZip64 = false, params string[] filePaths)

Metoda ta również dodaje pliki do archiwum w formacie „.zip”, jednak udostępnia więcej parametrów do dokładniejszego zdefiniowania potrzeb. Zwraca flagę indykującą czy spakowanie plików się powiodło.

Parametry:

* zipPathAndName – ścieżka i nazwa archiwum docelowego
* password – hasło do archiwum
* compression – poziom kompresji. Możliwe wartości: none, best, fastest.
* aes256Encryption – flaga indykująca czy używać szyfrowania Aes256 plików w archiwum.
* useZip64 – flaga indykująca czy użyć Zip64 przy zapisywaniu archiwum (do dużych plików).
* filePaths – ścieżki do plików, które chcemy dodać do archiwum.

#### bool ExtractFiles(string zipPathAndName, string destination, string password = null, bool overwrite = false)

Metoda wypakowuje pliki z archiwum. Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* zipPathAndName - ścieżka i nazwa archiwum do rozpakowania.
* destination – miejsce docelowe do zapisania rozpakowanej zawartości.
* password – hasło do archiwum.
* overwrite – flaga indykująca czy nadpisać zawartość w miejscu docelowym

#### bool DeleteEntriesFromArchive(string zipPathAndName, params string[] entriesToDelete)

Metoda usuwa wpisy w archiwum. Zwraca flagę indykującą czy operacja się powiodła.

Parametry

* zipPathAndName – ścieżka i nazwa archiwum do zmodyfikowania.
* entriesToDelete – nazwy wpisów do usunięcia w archiwum.

#### bool UpdateEntriesInArchive(string zipPathAndName, params string[] entriesToUpdate)

Metoda aktualizuje wpisy w archiwum.

Parametry:

* zipPathAndName – ścieżka i nazwa archiwum do zmodyfikowania.
* entriesToUpdate – nazwy wpisów do zaktualizowania w archiwum.

#### bool RenameEntryInArchive(string zipPathAndName, string oldName, string newName)

Metoda zamienia nazwę danego wpisu w archiwum. Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* zipPathAndName – ścieżka i nazwa archiwum, w którym znajduje się plik do zmodyfikowania.
* oldName – nazwa pliku do zmodyfikowania.
* newName – nowa nazwa pliku.

### Przykłady

Poniższy skrypt wypisze pliki z katalogu, który chcemy spakować, następnie doda ten katalog do nowego archiwum, wypakuje archiwum w nowej lokalizacji, a na koniec wypisze wypakowane pliki w celu porównania, czy operacje kompresji i wypakowywania przebiegły pomyślnie.

Przykład 18



Rysunek 26

**

Rysunek 27

Przykład 19

****

Rysunek 28

**

Rysunek 29

Powyższy przykład ilustruje stworzenie archiwum z hasłem i kompresją oraz próbę rozpakowania przy podaniu nieprawidłowego hasła do archiwum.

Przykład 20



Rysunek 30

**

Rysunek 31

W przykładzie 20. podaliśmy prawidłowe hasło do rozpakowania archiwum i operacja przebiegła pomyślnie.

## Nunit

### Implementacja

Do uruchamiania testów jednostkowych wykorzystujemy program nunit3-console.exe, zawarty we frameworku NUnit w wersji trzeciej. Jest to prosta aplikacja, która potrafi uruchomić testy jednostkowe napisane w NUnit 3.0 lub wyższych wersjach biblioteki oraz niższych, gdy jest zainstalowany odpowiedni sterownik. Program automatycznie zapisuje wyniki w formacie .xml, pozwalając na wytwarzanie odpowiednich raportów czy analizę wyników testów.

### Metody i opis parametrów

W naszym projekcie udostępniamy dwie metody, pozwalające uruchomić testy jednostkowe napisane w NUnit z podanych ścieżek do bibliotek z testami. Metody zwracają flagę indykującą czy proces z testami powiódł się, czy nie.

#### bool RunTests(string assemblyPaths, string conditions = null, string config = null)

Parametry:

* assemblyPaths – ścieżki do plików .dll, .csproj lub .nunit z testami jednostkowymi napisanymi w NUnit
* conditions – warunki do nałożenia na testy jednostkowe – możliwe są na przykład nazwy testów, klas, metod, kategorii testów czy właściwości zawartych w atrybutach oraz stosowania operatorów do porównania tych wartości (poniżej więcej informacji).
* config – nazwa konfiguracji projektu do załadowania, np. ‘Debug’

#### bool RunTestsWithOptions(string assemblyPaths, string conditions = null, string config = null, string workingDirectoryPath = null, string outputPath = null, string errorPath = null, bool? stopOnError = null, bool? skipNonAssemblies = null, bool? noResult = null, string verbosity = null, string timeout = null, bool? shadowcopy = null, string processIsolation = null, string numberOfAgents = null, string domainIsolation = null, string frameworkVersion = null, bool? runIn32Bit = null)

Parametry:

* assemblyPaths – ścieżki do plików .dll, .csproj lub .nunit z testami jednostkowymi napisanymi w NUnit
* conditions – warunki do nałożenia na testy jednostkowe – możliwe są na przykład nazwy testów, klas, metod, kategorii testów czy właściwości zawartych w atrybutach oraz stosowania operatorów do porównania tych wartości (poniżej więcej informacji).
* config – nazwa konfiguracji projektu do załadowania, np. ‘Debug’
* workingDirectoryPath – ścieżka do katalogu z plikami wyjściowymi (rezultatami testów)
* outputPath – ścieżka do pliku zawierającego tekst wyjściowy z testów
* errorPath – ścieżka do pliku zawierającego błędy z testów
* stopOnError – flaga indykująca czy zatrzymać testy przy napotkaniu jakiegokolwiek błędu
* skipNonAssemblies – flaga indykująca czy pomijać wszystkie pliki podane w parametrze ‘assemblyPaths’ niebędące bibliotekami z testami jednostkowymi, bez jakichkolwiek błędów
* noResult – flaga indykująca czy nie zapisywać pliku wynikowego testów
* verbosity – wewnętrzny poziom informacji. Możliwe wartości: Off, Error, Warning, Info, Verbose (Debug)
* timeout – timeout dla każdego przypadku testowego wyrażony w milisekundach
* shadowcopy – flaga indykująca czy .NET powinien utworzyć kopie załadowanych bibliotek.
* processIsolation – izolacja procesu dla bibliotek testowych. Możliwe wartości: Single, Separate, Multiple. Wartość domyślna to ‘Separate’ dla jednej biblioteki i ‘Multiple’ dla wielu bibliotek. Domyślnie procesy są uruchamiane równolegle.
* numberOfAgents – liczba agentów, które mogą być uruchomione jednocześnie, zakładając, że testy nie są uruchomione w jednym procesie. Domyślnie wszystkie testy są uruchamiane jednocześnie. To ustawienie pozwala na kontrolowanie uruchamiania testów jednostkowych równolegle.
* domainIsolation – izolacja domeny dla testów. Możliwe wartości: None, Single, Multiple. Wartość domyślna to ‘Single’ dla jednej biblioteki i ‘Multiple’ dla wielu bibliotek. Domyślnie procesy są uruchamiane równolegle.
* frameworkVersion – wersja/typ frameworku do użycia w testach, np. mono, net-4.5, v4.0, 2.0, mono-4.0.
* runIn32Bit – flaga indykująca czy testy mają być uruchomione w procesie 32-bitowym na systemach 64-bitowych.

Parametr conditions – filtry można zdefiniować w bardzo szczegółowy sposób. Możliwe wartości atrybutów (w warunkach po lewej stronie):

* test – pełna nazwa testu jednostkowego, np. Moja.Przestrzen.Nazw.KlasaTestowa.Test(5)
* name – nazwa testu jednostkowe, np. Test(5)
* class – pełna nazwa klasy zawierającej test, np. Moja.Przestrzen.Nazw.KlasaTestowa
* method – nazwa metody, np. Test
* cat – kategoria przypisana do testu, np. TestyKontrolera

Możliwe wartości operatorów:

* ‘==’ - równość
* ‘!=’ - nierówność
* ‘=~’ - dopasowanie do wyrażenia regularnego (Regex.IsMatch z .NET)
* ‘!~’ - niedopasowanie do wyrażenia regularnego (Regex.IsMatch z .NET)

Aby łączyć warunki, można zastosować operatory ‘&&’ (inne formy: ‘&’, ‘and’), ‘||’ (inne formy: ‘|’, ‘or’) oraz zaprzeczenie ‘!’.

Bardziej szczegółowe informacje można odnaleźć w dokumentacji NUnit (GitHub).

### Przykłady

Przykład 21



Rysunek 32

**

Rysunek 33

Powyżej widać rezultat przykładowych testów jednostkowych napisanych w NUnit i uruchomionych za pomocą naszej aplikacji Cake.exe.

Poniżej znajduje się przykład skryptu z większą liczbą opcji podaną przy uruchamianiu testów oraz zrzut ekranu z wynikiem procesu:

Przykład 22



Rysunek 34

**

Rysunek 35

Przykład 23



Rysunek 36

Powyższy skrypt uruchomi bibliotekę z testami jednostkowymi, które nie powinny się zakończyć sukcesem. Poniżej znajduje się zrzut ekranu z rezultatem:

**

Rysunek 37

## Xunit

### Implementacja

Do uruchamiania testów jednostkowych wykorzystujemy program xunit.console.exe, zawarty we frameworku xUnit w wersji drugiej. Jest to prosta aplikacja, która potrafi uruchomić testy jednostkowe napisane w xUnit 1.9.2 oraz 2.0 i wyższych wersjach.

Program automatycznie zapisuje wyniki w formacie .xml, pozwalając na wytwarzanie odpowiednich raportów czy analizę wyników testów. ???

### Metody i opis parametrów

Podobnie jak w bibliotece obsługującej testy jednostkowe napisane z użyciem NUnit, przygotowaliśmy dwie metody – jedną podstawową i jedną z rozbudowanymi opcjami - pozwalające uruchomić w skrypcie testy xUnit. Zwracają one flagę indykującą czy proces z uruchomionymi testami jednostkowymi powiódł się czy nie:

#### bool RunTests(string assemblyPaths, string traits = null)

Parametry:

* assemblyPaths – ścieżki do plików .dll z testami jednostkowymi napisanymi w xUnit
* traits – warunki do narzucenia na atrybuty metod z testami jednostkowymi z bibliotek podanych w poprzednim parametrze. Atrybut „trait” na danej metodzie testowej jest w postaci jednego wpisu do słownika typu (string name, string value), a warunek w parametrze powinien być w formie „name=value”. Przy wielu parametrach typu „trait” są one łączone operatorem „OR”. Ten parametr nakłada filtr i pozostawia tylko te metody, które posiadają jeden ze zdefiniowanych atrybutów w parametrze.
* notraits – warunki do narzucenia na atrybuty metod z testami jednostkowymi z bibliotek podanych w poprzednim parametrze. Atrybut „trait” na danej metodzie testowej jest w postaci jednego wpisu do słownika typu (string name, string value), a warunek w parametrze powinien być w formie „name=value”. Przy wielu parametrach typu „notrait” są one łączone operatorem „AND”. Ten parametr nakłada filtr i pozostawia tylko te metody, które NIE posiadają żadnego ze zdefiniowanych atrybutów w parametrze.

#### bool RunTestsWithOptions(string assemblyPaths, string traits = null, string notraits = null, string methodname = null, string classname = null, string parallel = null, int? maxthreads = null, bool? noshadow = null, bool? quiet = null, bool? serialize = null, string outputTypeAndName = null)

Parametry:

* assemblyPaths – ścieżki do plików .dll z testami jednostkowymi napisanymi w xUnit
* traits – warunki do narzucenia na atrybuty metod z testami jednostkowymi z bibliotek podanych w poprzednim parametrze. Atrybut „trait” na danej metodzie testowej jest w postaci jednego wpisu do słownika typu (string name, string value), a warunek w parametrze powinien być w formie „name=value”. Przy wielu parametrach typu „trait” są one łączone operatorem „OR”. Ten parametr nakłada filtr i pozostawia tylko te metody, które posiadają jeden ze zdefiniowanych atrybutów w parametrze.
* notraits – warunki do narzucenia na atrybuty metod z testami jednostkowymi z bibliotek podanych w poprzednim parametrze. Atrybut „trait” na danej metodzie testowej jest w postaci jednego wpisu do słownika typu (string name, string value), a warunek w parametrze powinien być w formie „name=value”. Przy wielu parametrach typu „notrait” są one łączone operatorem „AND”. Ten parametr nakłada filtr i pozostawia tylko te metody, które NIE posiadają żadnego ze zdefiniowanych atrybutów w parametrze.
* methodname – nazwa metody do testowania podana w całości (przestrzeń\_nazw.nazwa\_klasy.\_nazwa\_metody).
* classname – nazwa klasy do testowania podana w całości (przestrzeń\_nazw.nazwa\_klasy).
* parallel – parameter służący do ustawiania równoległości. Możliwe wartości: none – wyłącza równoległość, collections – zrównolegla testy tylko w obrębie kolekcji, assemblies – zrównolegla testy tylko w obrębie bibliotek, all – zrównolegla testy w obrębie kolekcji i bibliotek.
* maxthreads – Maksymalna liczba wątków przy zrównolegleniu testów z kolekcji. Możliwe wartości: 0 – nielimitowana liczba wątków, >0 – ustaw limit puli wątków zadania na wartość z parametru.
* noshadow – flaga indykująca czy .NET powinien utworzyć kopie załadowanych bibliotek.
* quiet – flaga indykująca czy pokazywać wiadomości z postępem testów
* serialize – flaga indykująca czy serializować wszystkie przypadki testów – do celów diagnostycznych.
* outputTypeAndName – opcja do ustawienia typu i nazwy pliku z rezultatami. Możliwe wartości: xml nazwa\_pliku – plik xml w stylu xUnit.net v2, xmlv1 nazwa\_pliku – xml w stylu xUnit.net v1, nunit nazwa\_pliku – xml w stylu NUnit, html nazwa\_pliku – plik html.

### Przykłady

Przykład 24



Rysunek 38

**

Rysunek 39

Skrypt z przykładu 24. uruchomi testy jednostkowe napisane z pomocą xUnit, a powyższy zrzut ekranu przedstawia wyniki tych testów.

## Git

### Implementacja

Narzędzie Git zintegrowaliśmy z .NET przy pomocy biblioteki LibGit2Sharp – jest to niewielkie narzędzie, które pozwala na implementację podstawowych operacji z systemu kontroli wersji Git do świata .NET i Mono. Biblioteki tej używamy również do uwierzytelniania w naszym module Git.

### Metody i opis parametrów

#### bool CommitAllChanges(string message)

Metoda ta aktualizuje lokalne repozytorium Git o wszystkie zmiany zawarte w projekcie/solucji. Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* message – wiadomość dodana do operacji commit.

#### bool CommitStagedChanges(string message)

Metoda ta aktualizuje lokalne repozytorium Git o zmiany, o których repozytorium zostało powiadomione za pomocą komendy „stage”. Pozostałe zmiany zostaną pominięte przy operacji commit. Metoda zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* message – wiadomość dodana do operacji commit.

#### bool DiffStaged()

Metoda wypisuje wszystkie zmiany pomiędzy lokalnym repozytorium a aktualną gałęzią z kodem. Zwraca flagę indykującą czy operacja się powiodła.

#### bool DiffWorkingDir()

Metoda wypisuje wszystkie zmiany pomiędzy katalogiem roboczym a aktualną gałęzią z kodem. Zwraca flagę indykującą czy operacja się powiodła.

#### bool DiffAll()

Metoda wypisuje wszystkie zmiany pomiędzy katalogiem roboczym i lokalnym repozytorium a atkualną gałęzią z kodem. Zwraca flagę indykującą czy operacja się powiodła.

#### bool Fetch()

Metoda pobiera wszystkie zmiany (zrobione przy pomocy operacji commit) z gałęzi docelowej do lokalnego repozytorium, bez próby łączenia kodu. Zwraca flagę indykującą czy operacja się powiodła.

#### bool Pull()

Metoda pobiera wszystkie zmiany (zrobione przy pomocy operacji commit) z gałęzi docelowej do lokalnego repozytorium oraz automatycznie łączy zmiany w kodzie. Zwraca flagę indykującą czy operacja się powiodła.

#### bool ResetHard(string commitShaHash = null)

Metoda wycofuje zmiany z aktualnej gałęzi usuwając je również z lokalnego repozytorium. Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* commitShaHash – Sha operacji commit, do której chcemy wycofać zmiany.

#### bool ResetMixed(string commitShaHash = null)

Metoda wycofuje zmiany z aktualnej gałęzi pozostawiając je w roboczej wersji projektu/solucji (lokalne repozytorium jest niepowiadomione o zmianach – potrzebna jest operacja „stage”). Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* - commitShaHash – Sha operacji commit, do której chcemy wycofać zmiany.

#### bool ResetSoft(string commitShaHash = null)

Metoda wycofuje zmiany z aktualnej gałęzi pozostawiając je w lokalnym repozytorium. Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* commitShaHash – Sha operacji commit, do której chcemy wycofać zmiany.

#### bool Stage(string path)

Metoda informuje lokalne repozytorium o zmianach w pliku. Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* path – ścieżka do pliku ze zmianami.

#### bool Stage(IEnumerable<string> paths = null)

Metoda informuje lokalne repozytorium o zmianach w plikach. Zwraca flagę indykującą czy operacja się powiodła.

### Przykłady

Przykład 25

****

Rysunek 40

Wykonanie powyższego skryptu rozpocznie się od zadania „GitInit”, w którym dostarczone są informacje niezbędne do pracy z systemem kontroli wersji, tj. ścieżka do repozytorium oraz dane potrzebne do zalogowania się w zewnętrznym serwisie, na którym hostowane jest repozytorium.

Następnie, użytkownikowi zostaną przedstawione zmiany w kopii lokalnej repozytorium w stosunku do ostatniego commita. Użytkownik musi wyrazić zgodę na stworzenie nowego commita poprzez wpisanie „y”.

Kolejnym krokiem jest pobranie zmian z serwera (które może zakończyć działanie skryptu w przypadku wykrycia konfliktów) oraz wypchnięcie uprzednio stworzonego commita w razie braku konfliktów.

Poniższy rysunek przedstawia wynik wykonania skryptu.



Rysunek 41

## Minifikacja i złączanie plików

### Implementacja

W naszym projekcie istnieje możliwość minifikacji oraz łączenia plików. Te operacje są możliwe dzięki wykorzystaniu pakietu „Microsoft Ajax Minifier” z galerii NuGet (przestrzeń nazw Microsoft.Ajax.Utilities).

### Metody i opis parametrów

W poniższych metodach wzorzec dopasowania może zawierać znaki specjalne:

* ‘?’ – dowolny pojedynczy znak
* ‘\*’ – dowolne zero lub więcej znaków
* ‘\*\*’ – zero lub więcej katalogów rekursywnych
* [...] – zestaw znaków, równoważny z grupami znaków w klasie Regex
* {group1,group2,...} – którakolwiek z grup wzorców (grupa może składać się z grup i wzorców)

#### bool MinifyJs(string pattern, string excludePattern = null, string destination = null, bool ignoreCase = true)

Metoda poddaje pliki .js (skrypty w języku JavaScript) procesowi minifikacji. Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* pattern – wzorzec dopasowania plików do minifkacji.
* excludePattern – wzorzec niedopasowania plików do minifikacji.
* destination – ścieżka i nazwa pliku z wynikiem minifikacji.
* ignoreCase – flaga indykująca czy ignorować wielkość liter podczas dopasowywania wzorca.

#### bool MinifyCss(string pattern, string excludePattern = null, string destination = null, bool ignoreCase = true)

Metoda poddaje pliki .css (Cascading Style Sheet) procesowi minifikacji. Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* pattern – wzorzec dopasowania plików do minifkacji.
* excludePattern – wzorzec niedopasowania plików do minifikacji.
* destination – ścieżka i nazwa pliku z wynikiem minifikacji.
* ignoreCase – flaga indykująca czy ignorować wielkość liter podczas dopasowywania wzorca.

#### bool BundleFiles(string pattern, char? separator = null, string destination = null, string excludePattern = null, bool ignoreCase = true)

Metoda łączy pliki w jeden wynikowy. Zwraca flagę indykującą czy operacja się powiodła.

Parametry:

* pattern – wzorzec dopasowania plików do złączenia.
* separator – znak oddzielający złączone pliki. (Na przykład ‘;’ przy plikach .js lub ‘\n’ przy skryptach html)
* excludePattern – wzorzec niedopasowania plików do złączenia.
* destination – ścieżka i nazwa pliku z wynikiem łączenia.
* ignoreCase – flaga indykująca czy ignorować wielkość liter podczas dopasowywania wzorca.

### Przykłady

Przykład 26

****

Rysunek 42

**

Rysunek 43

Powyższy skrypt podda dwa pliki, będące skryptami napisanymi w języku JavaScript, procesowi minifikacji, po czym złączy pliki wynikowe w jeden końcowy.

Przykład 27



Rysunek 44

**

Rysunek 45

Skrypt z przykładu 26. Złączy wszystkie pliki .html z lokalizacji źródłowej i zapisze plik wynikowy „bundled.html” do podanego w parametrze miejsca.

## Złożone skrypty

Poniżej znajdują się skrypty, które składają się z kombinacji wielu zadań i mają bardziej praktyczne zastosowanie.

Poniższy skrypt:

* zbuduje solucję ze wszystkimi projektami;
* zminifikuje wszystkie pliki .js i złączy je w jeden wynikowy;
* złączy pliki .html w jeden wynikowy;
* uruchomi testy jednostkowe;
* spakuje katalog z rezultatem procesu budowania i łączenia plików do archiwum;
* skopiuje archiwum do katalogu, z którego chcemy przygotować następną wersję aplikacji i przenieść ją na serwer testowy/produkcyjny.

Przykład 28

****

Rysunek 46

Kolejny skrypt:

* Uruchomi testy jednostkowe (kończy wykonanie skryptu, gdy nie wszystkie testy się powiodą).
* Zainicjalizuje zmienne konieczne do pracy z zewnętrznym repozytorium.
* Zaprezentuje użytkownikowi zmiany w lokalnym repozytorium oraz zapyta, czy użytkownik chce te zmiany wprowadzić do zewnętrznego repozytorium (w przypadku odmowy – wykonanie się kończy).
* Poprosi użytkownika o opisanie zmian i stworzy commita w lokalnym repozytorium.
* Pobierze zmiany ze zdalnego repozytorium (wykonanie zakończy się, jeśli pojawią się konflikty).
* Wypchnie utworzonego commita na serwer.
* Zminifikuje pliki z kodem JavaScript oraz stylami CSS.
* Połączy wszystkie zminifikowane pliki do odpowiednio: custom.min.js oraz custom.min.css.

W skrypcie zdefiniowana jest także pomocnicza klasa PathManager, tylko po to by uniknąć wielu powtórzeń ścieżki do repozytorium.

Przykład 29

****

Rysunek 47

Przykładowy wynik działania programu zakończonego niepowodzeniem z powodu testów jednostkowych:



Rysunek 48

Przykładowy wynik działania programu zakończonego sukcesem:

****

Rysunek 49

# Wnioski

Gdy zaczęliśmy pisać naszą pracę MS Roslyn było nowym pomysłem od Microsoft, jeszcze niedopracowanym oraz rozwiniętym tylko w części. Najważniejszymi komponentami tego narzędzia były biblioteki Roslyn.Compilers i Roslyn.Compilers.CSharp, których używaliśmy do uruchamiania skryptów .csx. Jednak pakiet nugetowy dostarczający te biblioteki przestał być wspierany i powstał nowy zestaw narzędzi - Microsoft.CodeAnalysis. Wykorzystaliśmy go w rdzeniu naszej pracy - egzekwowaniu kodu napisanego w C# oraz w budowaniu projektów i solucji MSBuild.

Przerwaliśmy pisanie pracy z rozmaitych powodów i wróciliśmy do projektu po 2-letniej przerwie. Po powrocie odkryliśmy istniejący projekt cakebuild.net, który jest bardzo rozbudowany i zapewnia większe możliwości. Chcieliśmy jednak dokończyć nasz projekt, gdyż jest on prostszy w użyciu i łatwiejszy do zrozumienia.

Gdy próbowaliśmy wykorzystać MSTest.exe do uruchamiania testów jednostkowych, napotkaliśmy się na problem – domyślnie program, gdy uruchamiany ze zwykłej konsoli systemowej, nie rozpoznawał bibliotek .dll jako poprawnych plików z testami jednostkowymi. Niezbędny do działania okazał się wpis do rejestru HKEY\_LOCAL\_MACHINE, a dokładniej:

„HKEY\_LOCAL\_MACHINE\SOFTWARE\Wow6432Node\Microsoft\VisualStudio\Your.Visual.Studio.Version\EnterpriseTools\QualityTools\TestTypes\”. W tym miejscu należało umieścić wpis z rozszerzeniem .dll jako prawidłowy typ testu, aby MSTest.exe mógł rozpoznać biblioteki z testami, w przeciwnym wypadku pojawiał się błąd: „File extension specified '.dll' is not a valid test extension”. Dlatego też zrezygnowaliśmy z tego rozwiązania i użyliśmy nunit3-console.exe i xunit.console.exe do uruchamiania testów jednostkowych napisanych przy użyciu pakietów NUnit i xUnit.

Nasza aplikacja ma szerokie zastosowanie w życiu codziennym. Skrypty podobne do tych przedstawionych powyżej w złożonych przykładach wystarczy zdefiniować raz, co nie jest czasochłonne. Następnie można je wykorzystywać regularnie w procesach związanych z kompilacją i wdrażaniem programów dla platformy .NET. Dzięki modularyzacji i ograniczonym skomplikowaniu naszego systemu, zarówno firmy, jak i pojedyncze osoby mogą szybko zaznajomić się z możliwościami i działaniem programu oraz dostosować go do swoich potrzeb.

# Bibliografia

# Wykaz symboli i skrótów

# Spis rysunków

Rysunek 1 12

Rysunek 2 12

Rysunek 3 13

Rysunek 4 13

Rysunek 5 14

Rysunek 6 14

Rysunek 7 16

Rysunek 8 16

Rysunek 9 17

Rysunek 10 17

Rysunek 11 18

Rysunek 12 22

Rysunek 13 22

Rysunek 14 23

Rysunek 15 24

Rysunek 16 24

Rysunek 17 25

Rysunek 18 25

Rysunek 19 26

Rysunek 20 28

Rysunek 21 29

Rysunek 22 29

Rysunek 23 30

Rysunek 24 30

Rysunek 25 31

Rysunek 26 33

Rysunek 27 34

Rysunek 28 34

Rysunek 29 35

Rysunek 30 35

Rysunek 31 36

Rysunek 32 38

Rysunek 33 39

Rysunek 34 40

Rysunek 35 40

Rysunek 36 41

Rysunek 37 41

Rysunek 38 43

Rysunek 39 43

Rysunek 40 46

Rysunek 41 47

Rysunek 42 48

Rysunek 43 49

Rysunek 44 49

Rysunek 45 49

Rysunek 46 50

Rysunek 47 54

Rysunek 48 54

Rysunek 49 55