# w1. Wstęp teoretyczny

1.1. C#

C# (C Sharp, dosłownie „[cis](https://pl.wikipedia.org/wiki/Cis_(d%C5%BAwi%C4%99k))”, „[c](https://pl.wikipedia.org/wiki/C_(d%C5%BAwi%C4%99k)) z [krzyżykiem](https://pl.wikipedia.org/wiki/Krzy%C5%BCyk)”) – [obiektowy](https://pl.wikipedia.org/wiki/Programowanie_obiektowe) [język programowania](https://pl.wikipedia.org/wiki/J%C4%99zyk_programowania) zaprojektowany w latach 1998-2001 przez zespół pod kierunkiem [Andersa Hejlsberga](https://pl.wikipedia.org/wiki/Anders_Hejlsberg) dla firmy [Microsoft](https://pl.wikipedia.org/wiki/Microsoft).

[Program](https://pl.wikipedia.org/wiki/Program_komputerowy) napisany w tym języku [kompilowany](https://pl.wikipedia.org/wiki/Kompilator) jest do języka [Common Intermediate Language](https://pl.wikipedia.org/wiki/Common_Intermediate_Language) (CIL), specjalnego [kodu pośredniego](https://pl.wikipedia.org/wiki/Kod_bajtowy)wykonywanego w [środowisku uruchomieniowym](https://pl.wikipedia.org/wiki/Biblioteka_uruchomieniowa) takim jak [.NET Framework](https://pl.wikipedia.org/wiki/.NET_Framework), [Mono](https://pl.wikipedia.org/wiki/Mono_(projekt)) lub [DotGNU](https://pl.wikipedia.org/wiki/DotGNU" \o "DotGNU). Wykonanie skompilowanego programu przez [system operacyjny](https://pl.wikipedia.org/wiki/System_operacyjny) bez takiego środowiska nie jest możliwe.

Nazwa.

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)](https://pl.wikipedia.org/wiki/Krzy%C5%BCyk) jest niedostępny na większości klawiatur, niektórych czcionkach i przeglądarkach internetowych, dlatego zalecane jest używanie [symbolu kratki (#)](https://pl.wikipedia.org/wiki/Kratka_(symbol)), który bardzo przypomina symbol muzyczny. Czasami tam, gdzie ma to uzasadnienie np. w materiałach promocyjnych, Microsoft używa nazwy C♯ zamiast C#.

Cechy języka

Język C# ma wiele cech wspólnych z językami programowania [Object Pascal](https://pl.wikipedia.org/wiki/Object_Pascal), [Delphi](https://pl.wikipedia.org/wiki/Embarcadero_Delphi), [C++](https://pl.wikipedia.org/wiki/C%2B%2B) i [Java](https://pl.wikipedia.org/wiki/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](https://pl.wikipedia.org/wiki/Od%C5%9Bmiecanie_pami%C4%99ci): zarządzaniem pamięcią zajmuje się środowisko uruchomieniowe. Oznacza to, że nie ma potrzeby samodzielnego zajmowania się zwalnianiem pamięci po obiektach, które przestają być używane
* [właściwości](https://pl.wikipedia.org/wiki/W%C5%82a%C5%9Bciwo%C5%9B%C4%87_(informatyka)), [indeksery](https://pl.wikipedia.org/wiki/Indekser" \o "Indekser): dodatkowe elementy składowe [klas](https://pl.wikipedia.org/wiki/Klasa_(programowanie_obiektowe))
* [delegaty](https://pl.wikipedia.org/wiki/Delegat_(informatyka)), [zdarzenia](https://pl.wikipedia.org/wiki/Zdarzenie_(informatyka)): odpowiedniki i rozwinięcie [wskaźników](https://pl.wikipedia.org/wiki/Zmienna_wska%C5%BAnikowa) na funkcje z [C++](https://pl.wikipedia.org/wiki/C%2B%2B)
* 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](https://pl.wikipedia.org/wiki/Mapowanie_obiektowo-relacyjne), narzędziach do analizy i weryfikacji kodu czy rozszerzeniach [AOP](https://pl.wikipedia.org/wiki/Programowanie_aspektowe). Mechanizm atrybutów został z C# zapożyczony do języka [Java](https://pl.wikipedia.org/wiki/Java) w wersji 1.5 (adnotacje), jakkolwiek samo Reflection API istniejące od pierwszego wydania języka stanowiło inspirację dla twórców C#
* [typy ogólne](https://pl.wikipedia.org/w/index.php?title=Typ_og%C3%B3lny&action=edit&redlink=1) (generics – dostępne od wersji [.NET](https://pl.wikipedia.org/wiki/.NET_Framework) 2.0): mechanizm zbliżony swoją ogólnością do [szablonów](https://pl.wikipedia.org/wiki/Szablon_(C%2B%2B)) w C++, jednak tu typ ogólny jest przenoszony do [modułu](https://pl.wikipedia.org/wiki/Modu%C5%82_(informatyka)) binarnego i możliwy jest do wykorzystania bez konieczności posiadania [kodu źródłowego](https://pl.wikipedia.org/wiki/Kod_%C5%BAr%C3%B3d%C5%82owy)
* dynamiczne tworzenie kodu: biblioteki [.NET](https://pl.wikipedia.org/wiki/.NET_Framework) 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](https://pl.wikipedia.org/wiki/Common_Intermediate_Language)).
* bogata biblioteka klas [BCL](https://pl.wikipedia.org/w/index.php?title=Base_Class_Library&action=edit&redlink=1), umożliwiająca rozwijanie aplikacji konsolowych, okienkowych (System.Windows.Forms oraz WPF 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](https://pl.wikipedia.org/wiki/Komentarz_(informatyka)):
  1. // to komentarz jednoliniowy, czyli wszystkiego, co znajduje się za nim, kompilator nie bierze pod uwagę.
  2. 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.

1.2 Skrypy - inspiracje oraz podobne narzędzia.

1. Rake - <https://github.com/ruby/rake>

[Program komputerowy](https://pl.wikipedia.org/wiki/Program_komputerowy) automatyzujący proces [kompilacji programów](https://pl.wikipedia.org/wiki/Kompilator) napisanych w języku [Ruby](https://pl.wikipedia.org/wiki/Ruby_(j%C4%99zyk_programowania)" \o "Ruby (język programowania)). Twórcą Rake jest Jim Weirich. Pozwala użytkownikowi zdefiniować zadania i opisać zależności między nimi oraz grupować zadania w przestrzeń nazw. Jest odpowiednikiem programów [make](https://pl.wikipedia.org/wiki/Make" \o "Make) czy [SCons](https://pl.wikipedia.org/w/index.php?title=SCons&action=edit&redlink=1" \o "SCons (strona nie istnieje)) z wieloma istotnymi różnicami. Pliki "Rakefile" są odpowiednikiem plików "Makefile" w [make](https://pl.wikipedia.org/wiki/Make" \o "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.

2. Fake - <https://fake.build/>

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 został zaprojektowany tak, aby był zwięzły, typowany, deklaratywny, rozszerzalny i łatwy w użyciu.

3. CS-Script - <http://www.csscript.net/index.html>

Jest to oparte na CLR narzędzie do zarządzania skryptami, które używa zgodnego ze standardami ECMA 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.

4. ScriptCS - <http://scriptcs.net/>

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#.

5. Cake - <http://cakebuild.net/>

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 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.

//6. Cake - <http://lmarcinek.pl/cake-c-make-niezalezny-build/>

1.3. MS Roslyn

Gdy zaczęliśmy pisać, to było tylko Roslyn.Compilers i Roslyn.Compilers.CSharp, którego używaliśmy. Jednak ten pakiet nugetowy przestał być wspierany i został przemianowany na Microsoft.CodeAnalysis.Scripting – biblioteka ta jest częścią tzw. „.NET Compiler Platform”, lepiej znaną pod kryptonimem „Roslyn”. Jest to zestaw kompilatorów open-source i API do analizy kodu w C# i Visual Basic .NET stworzony przez firmę Microsoft. Projekt ten zawiera między innymi kompilatory wyżej wymienionych języków napisanych właśnie w tych językach. Są one dostępne przez tradycyjne programy z linii wiersza poleceń, ale również jako API dostępne 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.

Więcej informacji oraz kod źródłowy „.NET Compiler Platform” można znaleźć pod adresami:

<https://en.wikipedia.org/wiki/.NET_Compiler_Platform>

<https://github.com/dotnet/roslyn>

1.4. 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, folderach, uruchamianiu testów jednostkowych oraz pracowaniu z systemem kontroli wersji Git. Jak można zauważyć nasz system ma wiele zastosowań.

1.5. Skrypty i zadania w skryptach – ogólny opis na czym to wszystko polega

1.5.1. 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.

Zadania można wykonywać przy użyciu metody SetDefault(string jobName). Wykonanie zadania może nastąpić w skrypcie dopiero po zdefiniowaniu wszystkich zależności oraz samego zadania.

1.5.2. Przykłady. (Zrzuty ekranu trzeba też)

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).

new Job("FirstJob").Does(() =>

{

System.Console.WriteLine("I am the first job.");

});

SetDefault("FirstJob");

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, oddzielonych przecinkami.

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

using System;

new Job("Test").Does(() =>

{

Console.WriteLine("Testing stuff...");

});

new Job("Deploy").DependsOn("Test").Does(() =>

{

Console.WriteLine("Heavy deploy action");

});

SetDefault("Deploy");

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

new Job("FirstJob").DependsOn("ThirdJob", "SecondJob").Does(() =>

{

Log(LogLevel.Info,

"I'm the first Job. I depend on the third and the second Job.");

});

new Job("SecondJob").DependsOn("ThirdJob").Does(() =>

{

Log(LogLevel.Info, "I'm the second Job. I depend on the third Job.");

});

new Job("ThirdJob").Does(() =>

{

Log(LogLevel.Info, "I'm the third Job.");

});

SetDefault("FirstJob");

Kolejność wywołań zadań w powyższym skrypcie będzie następująca: „ThirdJob”, „SecondJob”, „FirstJob”. Każde z zadań będzie wykonane tylko raz, gdyż są one idempotentne i kolejne wywołania „ThirdJob” byłyby bezcelowe i czasochłonne.

2. Dokumentacja techniczna – o pracy

2.1 Uruchamianie programu

2.1.1. Opcje uruchamiania

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

X:\Path\To\Cake>Cake.exe /s Z:\Path\To\Your\Script.csx /v Info /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
* /verbosity (/v) - poziom zapisywania do logów (możliwy jeden z pięciu: Debug, Info, Warn, Error, Fatal)
* /help (/h) – pomoc

2.1.2. Przykłady

2.2. Referencjonowanie skryptów i bibliotek, modularność i rozszerzalność.

2.2.1. Zewnętrzne biblioteki.

Oprócz bibliotek załączonych w programie CAKE można dodawać referencje do dowolnego pliku .dll ze skomiplowanym 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:

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.

// cake using "X:/Path/To/Your/dll/Build.dll";

new Job("BuildMySolution").Does(() =>

{

BuildProject(@"X:\Path\To\Your\Solution.sln");

});

SetDefault("BuildMySolution");

2.2.2. 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.

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

// cake load "X:/Path/To/Your/script/script.csx";

new Job("GetDataFromLoadedScript").Does(() =>

{

System.Console.WriteLine(SomethingFromTheLoadedScript);

});

SetDefault("GetDataFromLoadedScript");

2.3. JobManager, RoslynEngine (opisanie naszego projektu Cake w solucji)

2.4. Zapisywanie do logów.

2.4.1. 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 5 poziomów wiadomości: 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 debug 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.

2.4.2. 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:

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<targets>

<target name="coloredConsole" xsi:type="ColoredConsole" layout="${level:uppercase=true} | ${logger} | ${message}" />

<target name="script" xsi:type="ColoredConsole" layout="${level:uppercase=true} | ${logger} | ${message}" useDefaultRowHighLightingRules="false">

<highlight-row foregroundColor="Cyan" condition="true" />

</target>

<target name="file" xsi:type="File" fileName="log.txt" />

</targets>

<rules>

<logger name="Script" writeTo="script" final="true" minlevel="Debug" />

<logger name="\*" writeTo="coloredConsole" minlevel="Debug" />

<logger name="\*" writeTo="file" minlevel="Debug" />

</rules>

</nlog>

Konfiguracja NLoga 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).

Istnieje wiele możliwości definiowania źródła. Lista dostępnych znajduje się pod adresem:

<https://github.com/nlog/nlog/wiki/Targets>

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

<targets async="true">

<target name="f" xsi:type="File" fileName="${basedir}/logs/log.txt"

archiveFileName="${basedir}/logs/archives/log.{#####}.txt"

archiveAboveSize="10240"

archiveNumbering="Rolling"

keepFileOpen="false"/>

</targets>

 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:

<target name="f" xsi:type="File" fileName="logfile.txt"

layout="${longdate}|${level:uppercase=true}|${logger}|${message}"/>

 Natomiast wykaz wszystkich możliwych zmienny dla ciekawych, można znaleźć pod adresem: <https://github.com/nlog/nlog/wiki/Layout-renderers>

<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 celówoddzielona 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:

<logger name="\*" minlevel="Debug" writeTo="f" />

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

<logger name="NLogProject.\*" minlevel="Warn" maxlevel="Fatal" writeTo="f" />

<variable/>

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

<variable name="logProcess" value="${windows-identity}|${processid}"/>

<targets>

<target name="f" xsi:type="File" fileName="logfile.txt" layout="${longdate}|${level }|${logger}|${message}|${logProcess}"/>

</targets>

2.4.3. Zapisywanie do logów bezpośrednio ze skryptu

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

new Job("LogSomething").Does(() =>

{

Log(LogLevel.Info, “message to log”);

});

SetDefault("LogSomething");

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.

new Job("LogSomething").Does(() =>

{

Log(LogLevel.Warn, “warning to log”, "CustomLoggerName");

});

SetDefault ("LogSomething");

2.5. Budowanie projektu i solucji

2.5.1. Implementacja

Do zadań związanych z budowaniem projektów i solucji korzystamy z narzędzia firmy Microsoft MSBuild, a właściwie klasy BuildManager z przestrzeni nazw „Microsoft.Build.Execution”. Obudowaliśmy metodę budującą projekt/solucję w sprawdzanie warunków oraz zapisywanie do logów.

2.5.2. Metody i opis parametrów

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

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

Parametry:

- projectFile – ścieżka z nazwą pliku projektu/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)

2.5.3 Przykłady pojedynczego projektu i całej solucji ze zrzutami ekranu

2.6. Operacje na plikach i folderach

2.6.1. 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.

2.6.2. 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:

- filePath – ścieżka do pliku, w którym chcemy wykonać zamianę tekstu.

- regex – wzorzec do podmiany.

- newText – tekst, na który chcemy zamienić podany wzorzec.

* string[] LookForFileInDirectories(string filename, string directories)

Metoda szuka pliku w katalogach. Zwraca pełne ścieżki do wszystkich odnalezionych plików o podanej nazwie lub nazwę szuaknego pliku, gdy plik nie został odnaleziony w żadnej   
z lokalizacji.

Parametry:

- filename – nazwa pliku, po której chcemy przeszukiwać katalogi.

- directories – ścieżki do katalogów, które chcemy przeszukać, oddzielone przecinkami.

2.6.3. Przykłady

2.7. Archiwa

2.7.1. 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.

2.7.2. 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.

2.7.3. Przykłady.

Poniższy skrypt zbuduje solucję Solution.sln (wytłumaczone w rozdziale powyżej), a potem spakuje pliki wyjściowe do archiwum o nazwie “Build Output” w folderze z aplikacją.

// cake using "X:/Path/To/Your/dll/Build.dll";

// cake using "X:/Path/To/Your/dll/Zip.dll";

new Job("BuildMySolution").Does(() =>

{

BuildProject(@"X:\Path\To\Your\Solution.sln", @"Z:\Output\Path\");

});

new Job("ZipTheOutput").DependsOn("BuildMySolution").Does(() =>

{

ZipFiles("BuildOutput", @"Z:\Output\Path\");

});

SetDefault("ZipTheOutput");

2.8. Nunit

2.8.1. 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.

2.8.2. 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, np. ‘==’, ‘!=’, ‘=~’, ‘||’, itp.

- 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, bool? disposeRunners = 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, np. ‘==’, ‘!=’, ‘=~’, ‘||’, itp.

- 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.

- disposeRunners – flaga indykująca czy pozbywać się każdego ‘runnera’, gdy skończy on swoje testy jednostkowe.

Dokładniejsze informacje na temat możliwych parametrów oraz w jaki sposób narzucać warunki na testowanie danych bibliotek można znaleźć pod adresami:

<https://github.com/nunit/docs/wiki/Console-Command-Line>

<https://github.com/nunit/docs/wiki/Test-Selection-Language>

2.8.3. Przykłady.

2.9. XUnit

2.9.1. 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. ???

2.9.2. 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

2.10. Git

3. Wnioski – co nam wyszło, co nie wyszło

Nie udało nam się dodać naszej aplikacji jako dodatek do Microsoft Visual Studio („extension”). Poza tym, gdy próbowaliśmy wykorzystać MSTest.exe do uruchamiania testów jednostkowych, napotkaliśmy się na problem – domyślnie program nie rozpoznawał bibliotek .dll jako poprawnych plików z testami jednostkowymi, potrzebny był wpis do rejestru HKEY\_LOCAL\_MACHINE, gdy MSTest.exe uruchamiany był z konsoli, dlatego też zrezygnowaliśmy z tego rozwiązania i użyliśmy nunit3-console.exe.

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.

4. Podsumowanie

# Uruchamianie CAKE

Cake powinien być uruchamiany z wiersza poleceń w następujący sposób:

X:\Path\To\Cake>Cake.exe /s Z:\Path\To\Your\Script.csx /v Info /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#
* /verbosity (/v) - poziom loggowania (możliwy jeden z pięciu: Debug, Info, Warn, Error, Fatal)
* /help (/h) - pomoc
* /runjob (inaczej /r) - zadanie do uruchomienia zdefiniowane w skrypcie c#, niezbędne jeśli w skrypcie brakuje metody SetDefault