



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

Podczas pisania aplikacji programiści są zmuszeni do wykonywania nużących, powtarzalnych czynności, takich jak kompilacja rozwiązań, przeprowadzanie testów jednostkowych, wdrażanie aplikacji, itp. Istnieje potrzeba automatyzacji tych operacji. Z tego powodu powstało wiele narzędzi ułatwiających ten proces.

Celem pracy inżynierskiej jest stworzenie narzędzia spełniającego wymagania rozważane powyżej. Nasza aplikacja – „Cake” – używa części platformy kompilacyjnej .NET, kolokwialnie nazwanej „Roslyn” w celu uruchamiania skryptów napisanych w C# i zawartego w nich kodu. System umożliwia użytkownikowi specyfikację zależności pomiędzy różnymi zadaniami zdefiniowanymi w skrypcie i jest łatwo rozszerzalny. W pracy opisujemy aplikację ze wszystkimi jej modułami oraz porównujemy ją z innymi, podobnymi narzędziami. Praca składa się z czterech rozdziałów.

Pierwszy z nich przedstawia problem oraz dokładniej opisuje cele pracy.

W drugiej części opisane są podobne narzędzia, które były inspiracją przy pisaniu aplikacji oraz podobieństwa i różnice między nimi, a „Cake”.

W trzeciej części pracy zawarta jest architektura systemu i opis narzędzi w nim wykorzystanych. Moduł główny wraz z modułami rozszerzającymi są opisane i przeanalizowane. Metody, które mogą być użyte przy pisaniu skryptu są zaprezentowane wraz z parametrami wejściowymi.

Ostatni rozdział jest podsumowaniem pracy.

ABSTRACT

During development process software developers are forced to repeat dull, repeatable actions, such as compiling solutions, running unit tests, deploying applications, etc. There has been a need for automation of these operations. Due to that many tools have been created to facilitate the process.

The purpose of this engineer thesis is creating tool that fulfils requirements described above. Our application – Cake – uses a part of .NET Compiler Platform to run scripts written in C# and execute code in them. The system enables the user to define dependencies between different jobs defined in scripts and is easily extendable. The dissertation contains a description of the application with all its modules and a comparison of the system to other, similar tools. The thesis includes four chapters.

The first of them makes an introduction to the topic and further describes the goals.

The second part characterizes different tools that had influence on Cake application. Similarities and differences between them and Cake are described.

The third part of the thesis includes architecture of the system and describes the tools used in the application. All the extension modules alongside with the main module of the system are described and analysed. Methods that can be used when writing scripts are presented with input parameters.

The last chapter brings the conclusion of the thesis.

Warszawa, dnia ...............

Oświadczenie

Oświadczam, że moją część pracy inżynierskiej (zgodnie z podziałem zadań opisanym w pkt. ...) pod tytułem; „ ..................................”, której promotorem jest ............... wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

..........................................

Warszawa, dnia ...............

Oświadczenie

Oświadczam, że moją część pracy inżynierskiej (zgodnie z podziałem zadań opisanym w pkt. ...) pod tytułem; „ ..................................”, której promotorem jest ............... wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

..........................................

Spis treści

[1 Wstęp 7](#_Toc497851443)

[2 Opis istniejących narzędzi 9](#_Toc497851444)

[3 Implementacja 12](#_Toc497851445)

[3.1 Architektura systemu 12](#_Toc497851446)

[3.2 Wykorzystane narzędzia 13](#_Toc497851447)

[3.3 Moduł główny 15](#_Toc497851448)

[3.4 Moduły rozszerzające 20](#_Toc497851449)

[3.4.1 Cake.Build 20](#_Toc497851450)

[3.4.2 Cake.Files 21](#_Toc497851451)

[3.4.3 Cake.Zip 23](#_Toc497851452)

[3.4.4 Cake.NUnit 27](#_Toc497851453)

[3.4.5 Cake.xUnit 31](#_Toc497851454)

[3.4.6 Cake.Git 33](#_Toc497851455)

[3.4.7 Cake.Minify 36](#_Toc497851456)

[3.5 Tworzenie własnych modułów rozszerzających 37](#_Toc497851457)

[3.6 Złożone skrypty 37](#_Toc497851458)

[3.7 Uwagi końcowe 40](#_Toc497851459)

[4 Wnioski 41](#_Toc497851460)

[5 Bibliografia 42](#_Toc497851461)

[6 Spis rysunków 43](#_Toc497851462)

# 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 zajmują dużo czasu, przez swoją powtarzalność są nużące, a to z kolei może 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 pojedynczych 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 powstało wiele programów w różnych 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 języku używanym podczas pisania oprogramowania na platformę .NET. Nie istniała wtedy możliwość wykonania pojedynczego pliku napisanego w C#. 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 do wdrożenia w dowolnym środowisku, wykonywania 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.

W 2011 roku Microsoft opublikował projekt **Roslyn** (zostanie on omówiony w dalszej części pracy), którego niewielką częścią było API pozwalające wykonywanie dowolnych (oczywiście poprawnych) fragmentów 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 programista .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.

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 pracy z systemem kontroli wersji Git.

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.

# Opis istniejących narzędzi

Rake

Jednym z popularnych narzędzi automatyzujących proces kompilacji jest **Rake**. Program jest napisany w języku **Ruby**.

Tak samo jak w programie make oraz naszym narzędziu, większość interakcji użytkownika z tym programem sprowadza się do napisania skryptu nazywanego „**Rakefile**”. W przeciwieństwie do make, w którym użytkownik musi poznać składnię pisania plików Makefile, pliki Rakefile zawierają czysty kod napisany w Ruby.

Struktura zadania w Rake jest bardzo podobna do tej z naszego programu. Zadanie składa się z nazwy, zależności oraz anonimowej funkcji stanowiącej „ciało” zadania.

Rake posiada także wbudowane biblioteki dla często używanych zadań, takich jak manipulacja na plikach, czy usuwanie skompilowanych plików (zadanie „clean” często spotykane w różnych środowiskach).

Rake jest częścią standardowej biblioteki Ruby od wersji 1.9.

Fake

**Fake** – F# Make jest kolejnym systemem automatyzującym proces kompilacji. System jest w pełni otwarty (open source). Napisany jest głównie w **F#** (część napisana jest też w C#), a więc tak jak i nasz system opiera się na platformie .NET. Skrypty do Fake pisane są także w F#.

Podobnie jak w naszym narzędziu, programista piszący skrypt do Fake może w pełni korzystać z możliwości platformy .NET.

Kolejnym podobieństwem do naszego systemu jest struktura samego skryptu – począwszy od deklaracji listy bibliotek, z których użytkownik będzie korzystał w skrypcie, poprzez strukturę zadania oraz w końcu wskazanie zadania głównego.

Definicja zadania jest również bardzo podobna: zadanie definiujemy podając jego nazwę oraz akcję, którą ma wykonać w postaci funkcji anonimowej. Różnica pojawia się podczas definiowania zależności zadań. W skrypcie Fake zależności definiowane są na końcu skryptu, oddzielnie od samych zadań.

Twórcy Fake postawili nacisk na łatwą rozszerzalność narzędzia. Tak samo jak w naszym systemie, użytkownik może napisać własny moduł, z którego funkcjonalności może potem korzystać poprzez dodanie referencji do napisanej biblioteki w skrypcie.

CS-Script

Kolejnym narzędziem, o którym warto wspomnieć jest **CS-Script**. Nie jest to jednak system przeznaczony stricte do automatyzacji procesów kompilacji. CS-Script jest systemem skryptowym (*scripting system*) opartym na **CLR** (Common Language Runtime), w którym do pisania skryptów używa się języka C#.

Dzięki CS-Script możliwe było pisanie skryptów z użyciem C#, co pozwalało na pełne wykorzystanie obiektowości języka, a także silnego typowania – co nie zdarza się zbyt często w świecie języków skryptowych.

CS-Script wymaga zdefiniowanej w skrypcie funkcji Main, punktu wejściowego programu. System pozwala także na kompilacje skryptu do postaci pliku wykonywalnego.

Narzędzie to nie było jednak tworzone z myślą o automatyzacji kompilacji i nie posiada wbudowanych bibliotek pomocnych przy wykonywaniu tych zadań.

ScriptCS

**ScriptCS** jest systemem skryptowym, który podobnie jak nasze narzędzie, wykorzystuje projekt Roslyn do kompilacji skryptów.

Jest narzędziem bardzo podobnym do CS-Script, występują jednak między nimi różnice. Nie wymaga punktu wejściowego w skrypcie, ale nie istnieje możliwość kompilacji skryptu do pliku wykonywalnego. Jest projektem open source, rozwijanym przez społeczność użytkowników.

Cake

**Cake** – C# Make, jest narzędziem automatyzującym proces kompilacji. Ze wszystkich przez nas wymienionych, posiada najwięcej podobieństw z naszym systemem. Skrypty pisane są w C#, a system używa Roslyn do ich kompilacji.

Konstrukcja pojedynczego zadania jest bardzo podobna do tej z naszego systemu. Jedną z ważniejszych różnic jest brak możliwości definiowania różnych ścieżek w grafie zależnych zadań w zależności od powodzenia zadania.

Cake posiada także moduły zawierające wiele przydanych funkcji. Jest także projektem open source.

Gulp.js

Jednym z najpopularniejszych narzędzi służących do automatyzacji procesów budowania części klienckich aplikacji internetowych jest **Gulp.js**. Gulp został zbudowany na bazie **Node.js** i **npm** (*Node Package Manager*), a skrypty pisane są z użyciem języka **JavaScript**. Gulp może być używany na każdej platformie, która ma zainstalowany silnik Node.js.

Podobnie jak w naszym narzędziu, definicja zadania składa się z jego nazwy, listy zależności oraz funkcji będącej ciałem zadania. (Mast, 2016)

Dzięki popularności tego narzędzia powstało mnóstwo wtyczek, co czyni Gulpa bardzo wygodnym narzędziem.

Podsumowanie

Zdecydowana większość z wymienionych narzędzi została stworzona w tym samym celu – aby ułatwić pracę programistów poprzez automatyzację wielu powtarzalnych procesów.

Narzędzia są bardzo podobne co do zasady działania oraz sposobu ich używania. Główne różnice pomiędzy nimi to języki programowania, przy użyciu których pisane są same programy oraz skrypty dla nich. Poza tym niektóre z nich przeznaczone są do używania w konkretnej dziedzinie wytwarzania oprogramowania (np. Gulp.js przeznaczony jest głównie do rozwijania aplikacji internetowych).

Gdy podejmowaliśmy decyzje związane ze sposobem definiowania zadań i ich zależności oraz podstawowych funkcjonalności programu, inspiracje czerpaliśmy głównie z Gulp.js – przede wszystkim ze względu na to, że z wyżej wymienionych jest to narzędzie nam najlepiej znane.

Podczas wyboru pozostałych funkcjonalności naszego narzędzia nie ograniczyliśmy się jednak do działania w obrębie rozwijania aplikacji internetowych (tak jak w Gulp.js). Inspiracje czerpaliśmy niemalże ze wszystkich wymienionych narzędzi. Spośród ich możliwości wybraliśmy te, które naszym zdaniem są najpotrzebniejsze – zostały one zaimplementowane przez nas w poszczególnych modułach rozszerzających, które opisane są w następnym rozdziale.

# Implementacja

## Architektura systemu

W systemie, można wyróżnić 3 główne części:

1. Projekt „Cake” – rdzeń systemu.
2. Projekt „Common” – zawierający funkcjonalności dzielone przez resztę części systemu.
3. Moduły implementujące konkretne funkcjonalności wykorzystywane przez użytkownika w skryptach.

Projekt „Cake”

Projekt „Cake” jest główną częścią naszego systemu.

Zawiera statyczną klasę Program z funkcją Main – punktem wejściowym naszej aplikacji. W tym projekcie znajduje się też klasa ArgumentParser, odpowiedzialna za parsowanie argumentów przekazanych do naszego programu.

W projekcie Cake znajdują się także klasy reprezentujące zadania, jak na przykład Job, czy VoidJob, a także JobManager – klasa statyczna odpowiedzialna za rejestrację zadań ze skryptu, stworzenie grafu ich zależności oraz samo wykonanie zadań.

Jedną z ważniejszych klas tego projektu jest RoslynEngine, klasa udostępnia dostęp do Roslyn API, pozwalając na wykonanie skryptu użytkownika. Jest także odpowiedzialna za wczytywanie referencjonowanych bibliotek i innych skryptów.

Klasy te są dokładniej opisane w dalszej części pracy.

Projekt „Common”

Na projekt Common składa się kilka klas, które wykorzystywane są w różnych częściach systemu.

Do najważniejszych z nich należą Logger – klasa udostępniająca metody pozwalające logować wydarzenia podczas działania programu. Logowanie wykorzystywane jest praktycznie w każdym miejscu naszego systemu. Użytkownik może także korzystać z metod klasy Logger w skrypcie, aby śledzić wykonywanie zadań. Logowanie opisane jest dokładniej w dalszej części pracy.

Kolejną klasą używaną w wielu miejscach w naszym systemie jest PathParser. Klasa ta posiada dwie publiczne metody: GetDirectoriesPaths oraz GetFilePaths. Obydwie metody przyjmują jako argument ścieżkę w systemie plików, która może zawierać znaki wieloznaczne (*ang. wildcards*), a zwracają kolekcję ścieżek do konkretnych (odpowiednio) folderów i plików. Na przykład korzystając z metody GetFilePaths, ścieżka „C:\\*\*\\*proj” zostałaby rozwinięta w listę ścieżek do wszystkich plików z rozszerzeniem proj znajdujących się na dysku C.

Ostatnią ważną klasą w projekcie Common jest Processor, z publiczną metodą RunProcess, dzięki której możliwe jest uruchomienie ze skryptu dowolnego programu tak samo, jakbyśmy mogli to zrobić korzystając z wiersza poleceń.

Moduły funkcjonalności

Trzecią z opisywanych części systemu są moduły, w których implementowane są konkretne funkcjonalności udostępniane użytkownikowi piszącemu skrypt, takie jak operacje na plikach, korzystanie z funkcji systemy kontroli wersji, czy uruchamianie testów. Moduły te są wymienione oraz opisane w rozdziale „Moduły rozszerzające”.

## Wykorzystane narzędzia

MS Roslyn

„.NET Compiler Platform”, platforma lepiej znana pod kryptonimem „Roslyn” (Arh, 2016), 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. Jednym ze sposobów dostępu do nich jest dostęp poprzez API - natywnie z poziomu kodu .NET. Roslyn udostępnia moduły do analizy leksykalnej, syntaktycznej i semantycznej kodu, 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 uruchamiania kodu ze skryptu napisanego w tym języku poprzez statyczne metody klasy CSharpScript. 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.

NLog

Aby poinformować użytkownika co dzieje się w naszej aplikacji wykorzystujemy platformę NLog.

Dzięki temu narzędziu 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.

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 1

<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="${date:format=HH\:mm\:ss}>>> ${logger} | ${message}" />

<target name="script" xsi:type="ColoredConsole"

layout="${date:format=HH\:mm\:ss}>>> ${logger} | ${message}"

useDefaultRowHighLightingRules="false">

<highlight-row backgroundColor="Cyan" foregroundColor="Black"

condition="true" />

</target>

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

</targets>

<rules>

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

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

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

</rules>

</nlog>

W konfiguracji możemy ustalić takie zachowania jak archiwizacja plików po przekroczeniu danego rozmiaru, format wiadomości zapisywanych do logów, przepływ/warunki logowania, itp.

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

Przykład 2

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

{

Log(LogLevel.Info, “message to log”);

});

JobManager.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, np.

Przykład 3

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

{

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

});

JobManager.SetDefault("LogSomething");

Więcej informacji na ten temat można znaleźć w dokumentacji NLog na platformie GitHub.

DotNetZip

Jednym z modułów rozszerzających w naszej aplikacji jest moduł Cake.Zip pozwalający użytkownikowi wykonywać operację związane z kompresją i archiwizacją plików. Podczas implementacji modułu zdecydowaliśmy się na użycie jednej z najpopularniejszych bibliotek do manipulacji plikami zip na platformę .NET – DotNetZip.

DotNetZip to niewielka biblioteka open source, która w prosty sposób pozwala na wykonywanie podstawowych operacji związanych z archiwizacją plików, takich jak:

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

Testy jednostkowe

Do obsługi wykonywania testów jednostkowych korzystamy z dwóch zewnętrznych narzędzi: NUnit oraz xUnit. Oba te zestawy operują w .NET i posiadają otwartą licencję. Dzięki nim możemy uruchamiać testy jednostkowe poprzez konsolę, Visual Studio czy adapter testowy lub inne zewnętrzne narzędzia. Testy mogą być uruchamiane równolegle i kategoryzowane w grupy.

LibGit2Sharp

LibGit2Sharp to niewielkie narzędzie, które pozwala na implementację podstawowych operacji z systemu kontroli wersji Git w .NET i Mono. Biblioteki tej używamy również do uwierzytelniania w naszym module Git.

Microsoft Ajax Minifier

Microsoft Ajax Minifier umożliwia poprawienie wydajności aplikacji sieciowych poprzez zmniejszenie rozmiaru plików .css i .js. Pozwala również opcjonalnie wygenerować mapy źródeł dla skryptów JavaScript. Narzędzie to domyślnie będzie próbowało zredukować kod jak najbardziej: usuwa komentarze, białe znaki, niepotrzebne średniki i nawiasy klamrowe. Zamienia również nazwy lokalnych zmiennych i funkcji na mniejsze oraz usuwa niepotrzebny kod.

## Moduł główny

Moduł główny naszej aplikacji obsługuje parametry wejściowe, uruchamia skrypty .csx oraz obsługuje zadania i zależności między nimi.

Program

Uruchamianie naszej aplikacji obsługiwane jest w klasie głównej Program, która posiada tylko jedną metodę:

void Main(string[] args)

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

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

Przy uruchamianiu programu możliwe jest wykorzystanie jednego lub więcej poniższych parametrów:

* /script (/s) - lokalna ścieżka do skryptu .csx napisanego w języku 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, Debug, Info, Warn, Error, Fatal)
* /help (/h) – pomoc

Parametry obsługujemy przy pomocy dwóch klas pomocnicznych:

* Argument
* ArgumentParser

Pierwsza klasa służy do przedstawienia parametru, druga zaś przetwarza wszystkie parametry wpisane przez użytkownika. Po ich odczytaniu próbujemy uruchomić skrypt przy użyciu klasy RoslynEngine.

RoslynEngine

Ta klasa jest rdzeniem naszej aplikacji. Wykorzystujemy w niej klasy pomocnicze CSharpScript oraz ScriptOptions z przestrzeni nazw Microsoft.CodeAnalysis (Scripting oraz CSharp.Scripting). Implementując ją wykorzystaliśmy wzorzec projektowy singleton. (Rossel, 2016)

Task ExecuteFile(string filePath)

Metoda uruchamia plik .csx, którego ścieżkę podajemy w parametrze. W opcjach do uruchomienia ładujemy domyślne referencje oraz przestrzenie nazw potrzebne do wykonania skryptu. (Microsoft, 2017)

Skrypty, które obsługuje nasz program, składają się zwykle z różnorakich zadań (reprezentowanych przez niżej opisane klasy), polegających na definiowaniu operacji zarządzających wdrażaniem aplikacji oraz zależności między tymi zadaniami. 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 zadań zależnych i musi posiadać unikatową w obrębie skryptu nazwę.

Do obsługi zadań stworzyliśmy następujące obiekty pomocnicze:

JobStatus

Jest to typ wyliczeniowy, który opisuje status zadania.

Zadanie może posiadać następujące statusy:

* NotVisited – gdy przy obsłudze skryptu nie doszliśmy jeszcze do tego zadania
* Pending – gdy doszliśmy do zadania w skrypcie, ale obsługujemy jego zależności
* Done – gdy wykonaliśmy dane zadanie i wszystkie jego zależności
* Failed – gdy zadanie zostało wykonane niepomyślnie

JobResult

Klasa opisująca rezultat zadania, posiada trzy właściwości:

* bool Success – flaga mówiąca o tym, czy zadanie się powiodło
* dynamic ResultObject – właściwość przetrzymująca rzeczywisty rezultat z zadania, może być dowolnego typu (Mayo, 2015)
* Exception Exception – właściwość przetrzymująca potencjalny wyjątek napotkany podczas wykonywania zadania

Do definiowania zadań przygotowaliśmy dwie klasy – Job oraz VoidJob, które dziedziczą po abstrakcyjnej klasie CakeJob.

CakeJob

* internal string Name – nazwa zadania
* internal List<string> Dependencies – lista zadań zależnych
* internal string ExceptionJob – nazwa zadania, które powinno wykonać się gdy nastąpi wyjątek
* internal JobStatus Status – status zadania
* internal JobResult Result – rezultat zadania
* public CakeJob DependsOn(params string[] dependenciesToAdd) – metoda pozwalająca na zdefiniowanie zależności danego zadania przy pomocy nazw innych zadań
* public CakeJob DependsOn(params CakeJob[] dependenciesToAdd) – metoda pozwalająca na zdefiniowanie zależności danego zadania przy pomocy obiektów z innymi zadaniami
* public CakeJob OnException(string jobName) – metoda przypisująca wartość do właściwości ExceptionJob
* public CakeJob Does() – Metoda pozwalająca na zdefiniowanie akcji wykonywanej podczas danego zadania. Parametr wejściowy może mieć postać wyrażenia lambda lub metody z któregokolwiek modułu lub własnej biblioteki. W naszej aplikacji są dwa warianty:
  + Does(Action actionToDo) – metoda dla zadania typu VoidJob
  + Does(Func<dynamic> actionToDo) – metoda dla zadania typu Job

Różnica między klasami VoidJob i Job jest niewielka – zadanie zdefiniowane przy pomocy tej pierwszej zwraca rezultat, który posiada wyłącznie flagę przypisaną do właściwości Success – ta logika jest rozwiązywana wewnętrznie. Zadanie typu Job zwraca również rezultat typu dynamic, a więc może być to obiekt dowolnego typu. W obu przypadkach możemy w skrypcie przypisać zadanie do zmiennej, a następnie z właściwości Result odczytać wynik naszego zadania oraz zdecydować o przebiegu skryptu na podstawie flagi Success lub rezultatu ResultObject. Możemy na przykład przerwać działanie naszego skryptu, jeśli zadanie nie zostało pomyślnie wykonane. Ta funkcjonalność jest przydatna między innymi do zadań związanych z testami jednostkowymi, czy budowaniem projektów.

JobManager

Menedżer zadań obsługuje rejestrację zadań, ustawianie zależności między zadaniami oraz uruchamianie kodu.

private bool CycleDetection()

Metoda służy do sprawdzania czy po zbudowaniu grafu z zależności pomiędzy zadaniami powstanie cykl. W celu uniknięcia zapętlenia się skryptu zwracamy wyjątek w takiej sytuacji. Używamy w tej metodzie przeszukiwania w głąb (algorytm Depth-First-Search) oraz rekurencji. Graf budujemy na podstawie wewnętrznego słownika ze wszystkimi zarejestrowanymi zadaniami w skrypcie.

internal void RegisterJob(CakeJob job)

Metoda służy do rejestracji zadania w słowniku. Zadanie musi posiadać unikatową nazwę w obrębie skryptu.

public void SetDefault(CakeJob job)

Metoda służy do ustawienia punktu startowego w skrypcie – zadania, od którego działanie skryptu powinno się rozpocząć. Jako argument powinniśmy podać obiekt typu CakeJob. Jeśli nie podamy w argumentach naszej aplikacji od jakiego zadania program powinien zacząć działanie, musimy zdefiniować to przy użyciu metody SetDefault(string jobName).

public void SetDefault(string name)

Druga metoda służąca do ustawienia punktu startowego w skrypcie. Różni się od powyższej metody parametrem – jest to nazwa zadania, a nie sam obiekt.

private JobResult PerformJobWithDependencies(string jobName)

Najważniejsza metoda w menedżerze zadań. Odpowiada za obsługę wykonywania zadań i ich zależności – jest wywoływana rekurencyjnie, zaczynając od zadania podanego w argumencie lub w metodzie SetDefault.

Przykłady

Poniżej znajduje się skrypt z jednym zadaniem nazwanym “FirstJob”. Jego akcja zdefiniowana jest jako argument metody Does(Action actionToDo). Aby uniknąć konieczności podawania zadania – punktu wejściowego korzystamy z metody SetDefault z klasy JobManager.

Przykład 4

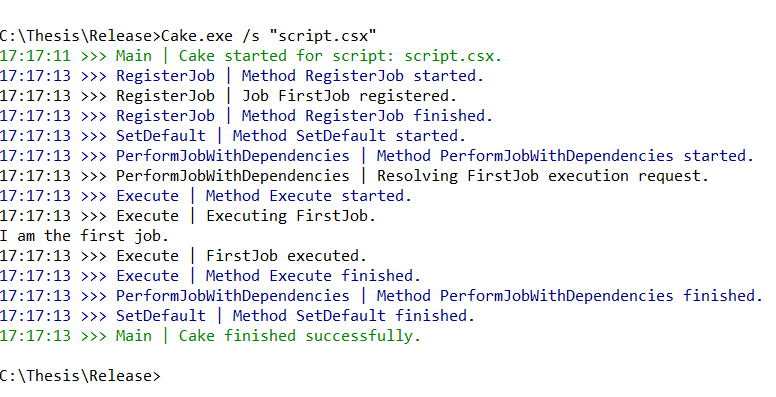
new VoidJob("FirstJob").Does(() =>

{

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

});

JobManager.SetDefault("FirstJob");



Rysunek 1

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

Przykład 5

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

{

Log(LogLevel.Info,

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

});

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

{

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

});

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

{

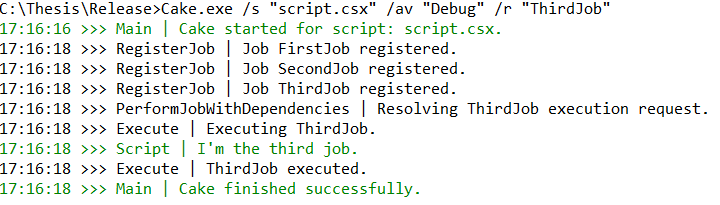
Log(LogLevel.Info, "I’m the third job.");

});

JobManager.SetDefault("FirstJob");

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

Poniższy zrzut ilustruje uruchamianie tylko niektórych zadań ze skryptu podanego w przykładzie 5. Można zauważyć, że zostały zarejestrowane trzy zadania, ale uruchomione tylko zadanie trzecie, gdyż wyszczególniliśmy w argumencie, od którego zadania mamy zacząć skrypt.



Rysunek 2

Załączanie skryptów i bibliotek

Aby załączyć inne skrypty lub biblioteki możemy posłużyć się standardem C# REPL (read-eval-print-loop). Przy użyciu dwóch prostych dyrektyw możemy łatwo użyć innego pliku .csx lub .dll (Price, 2017):

* #r „X:\Path\To\Your\Library\library.dll” – pozwala na załadowanie innej biblioteki (wykorzystanie na przykład w modułach rozszerzających).
* #load „X:\Path\To\Your\Script\script.csx” – pozwala na załadowanie innego skryptu (wykorzystanie na przykład w podziale kodu na wiele plików, klas – modularyzacja)

Dodawanie referencji do bibliotek czy skryptów, które chcemy załadować powinno mieć miejsce na początku skryptu. Używając tych dyrektyw z poziomu Visual Studio mamy dostęp do automatycznego uzupełniania (Intellisense), co może ułatwić pisanie bardziej złożonych skryptów – możemy w ten sposób trzymać małe skrypty obsługujące pojedynczą funkcjonalność oraz otrzymać separację czynności, co również wpływa na modularność rozwiązania.

## Moduły rozszerzające

### Cake.Build

Do zadań związanych z budowaniem projektów i solucji korzystamy z biblioteki Microsoft.CodeAnalysis. W tej części naszego projektu wykorzystujemy klasę MSBuildWorkspace (Harrison, 2017) z przestrzeni nazw Microsoft.CodeAnalysis.MSBuild, aby stworzyć przestrzeń roboczą do wczytania solucji czy projektu MSBuild. Następnie używamy GetCompilationAsync, aby sprawdzić rezultat kompilacji oraz go zapisać do odpowiednich plików - .dll (Dynamic-Link Library), .exe (Executable), .pdb (Program database) czy .xml (Extensible Markup Language). Wszystkie te działania obudowane są w zapisywanie do logów, sprawdzanie argumentów oraz obsługę wyjątków.

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

Metoda pozwala na zbudowanie pojedynczego projektu. Zwraca informację czy proces kompilacji powiódł się dla wszystkich projektów.

Parametry:

* projectFile – ścieżka z nazwą pliku projektu.
* 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 solucji. Zwraca informację czy proces kompilacji powiódł się dla wszystkich solucji.

Parametry:

* solutionFile – ścieżka z nazwą pliku solucji.
* 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)

### Cake.Files

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.

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

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

Parametry

* sourceDir – ścieżka do katalogu źródłowego.
* destinationDir – ścieżka do katalogu docelowego.
* copySubDirs – flaga mówiąca czy kopiować podkatalogi. Wartość domyślna to „true”.
* overwrite – flaga mówiąca czy pliki w katalogu docelowym powinny zostać nadpisane. Wartość domyślna to „false”.
* cleanDestinationDirectory – Flaga mówiąca 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 informację czy kopiowanie się powiodło.

Parametry

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

bool DeleteFile(string filePath)

Metoda usuwa plik z podanego miejsca. Zwraca informację 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 mówiąca czy przeszukiwać również podkatalogi. Wartość domyślna to „false”.

bool DeleteFilesWithPattern(string parentDirectoryPath, string filePattern)

Metoda usuwa pliki pasujące do wzorca z podanego katalogu. Zwraca informację czy usuwanie wszystkich plików 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 informację czy usuwanie wszystkich podkatalogó się powiodło.

Parametry:

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

bool DeleteDirectory(string directoryPath)

Metoda usuwa katalog z podanego miejsca. Zwraca informację 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 informację 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 informację czy plik został odnaleziony a jego zawartość poprawnie zamieniona według podanego wyrażenia regularnego.

Parametry:

* filePath – ścieżka do pliku, którego zawartość chcemy zmodyfikować.
* regex – wyrażenie regularne dopasowujące tekst do podmienienia.
* newText – tekst, którym chcemy zamienić wszystkie wystąpienia wzorca pasujące do wyrażenia regularnego z parametru regex.

### Cake.Zip

W naszym systemie udostępniamy funkcjonalność związaną z kompresją plików i folderów, a także innymi operacjami na z pomocą pakietu DotNetZip (IonicZip.dll).

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

Metoda dodaje pliki do archiwum w formacie „.zip”. Zwraca informację 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 informację 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 mówiąca czy używać szyfrowania Aes256 plików w archiwum.
* useZip64 – flaga mówiąca 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 informację 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 mówiąca czy nadpisać zawartość w miejscu docelowym

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

Metoda usuwa pozycje w archiwum. Zwraca informację czy operacja się powiodła.

Parametry

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

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

Metoda aktualizuje pozycje w archiwum.

Parametry:

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

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

Metoda zamienia nazwę danego elementu w archiwum. Zwraca informację 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.

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 6

#r "../../../Zip/bin/Debug/Zip.dll"

new Job("ZipOutput").Does(() => {

return Zip.Methods.ZipFilesWithOptions(@"D:\Dane\Ernest\Praca\ZippedOutput.zip", "password", "fastest",

false, false, @"D:\Dane\Ernest\Praca\TestOutput\");

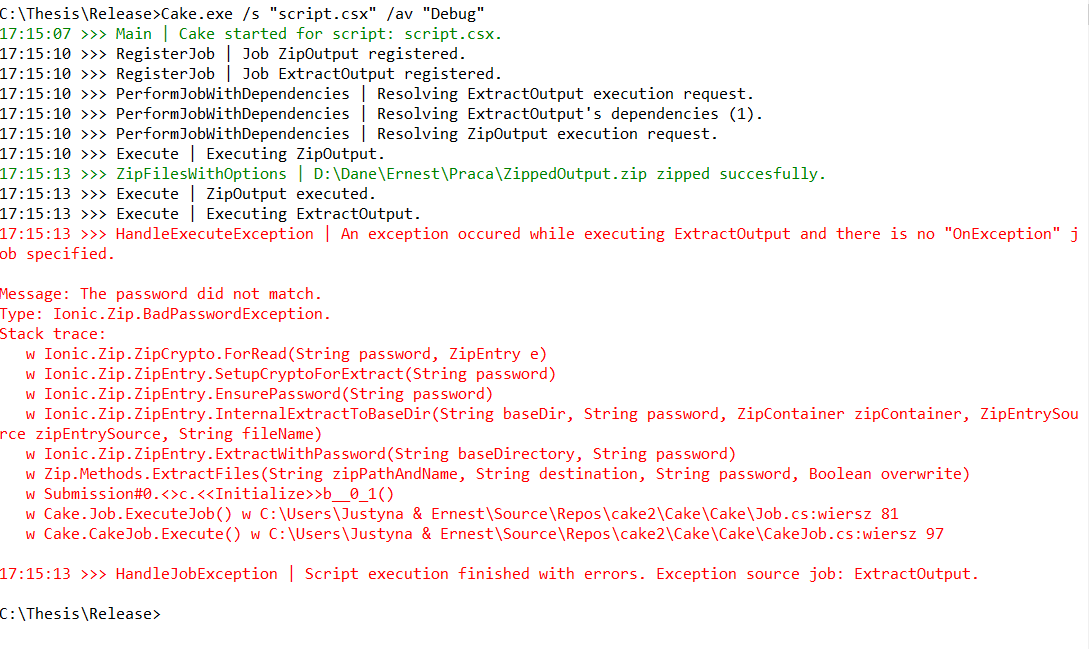
});

new Job("ExtractOutput").DependsOn("ZipOutput").Does(() => {

return Zip.Methods.ExtractFiles(@"D:\Dane\Ernest\Praca\ZippedOutput.zip", @"D:\Dane\Ernest\Praca\TestOutput\", "wrongPassword");

});

JobManager.SetDefault("ExtractOutput");



Rysunek 3

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

Przykład 7

#r "./../../Zip/bin/Debug/Zip.dll"

new Job("ZipOutput").Does(() => {

return Zip.Methods.ZipFilesWithOptions(@"D:\Dane\Ernest\Praca\ZippedOutput.zip", "password",

"fastest", false, false, @"D:\Dane\Ernest\Praca\TestOutput\");

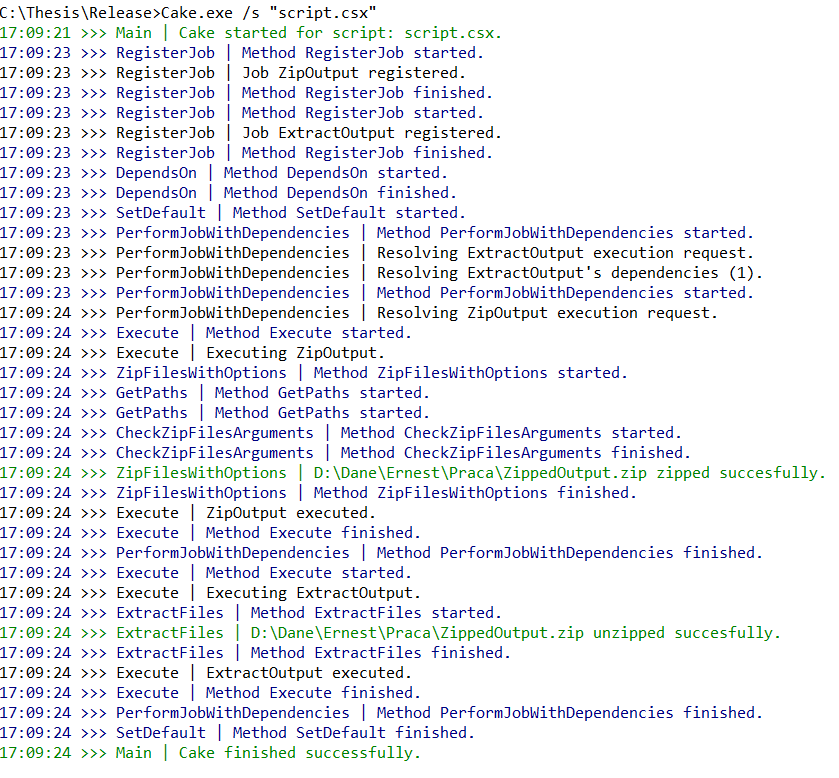
});

new Job("ExtractOutput").DependsOn("ZipOutput").Does(() => {

return Zip.Methods.ExtractFiles(@"D:\Dane\Ernest\Praca\ZippedOutput.zip", @"D:\Dane\Ernest\Praca\TestOutput\", "password");

});

JobManager.SetDefault("ExtractOutput");



Rysunek 4

W przykładzie 7. podaliśmy prawidłowe hasło do rozpakowania archiwum i operacja przebiegła pomyślnie.

### Cake.NUnit

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. (Maddock, 2017)

W naszym projekcie udostępniamy dwie metody, pozwalające uruchomić testy jednostkowe napisane w NUnit z podanych ścieżek do bibliotek z testami. Metody zwracają informację czy wszystkie testy jednostkowe zakończyły się sukcesem.

bool RunTests(string conditions = null, string config = null, params string[] assemblyPaths)

Parametry:

* 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’.
* assemblyPaths – ścieżki do plików .dll, proj lub .nunit z testami jednostkowymi napisanymi w NUnit.

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, proj 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 – podanie tego parametru wiąże się z przekierowaniem wyjścia z konsoli do pliku.
* errorPath – ścieżka do pliku zawierającego błędy z testów.
* stopOnError – flaga mówiąca czy zatrzymać testy przy napotkaniu jakiegokolwiek błędu.
* skipNonAssemblies – flaga mówiąca czy pomijać wszystkie pliki podane w parametrze assemblyPaths niebędące bibliotekami z testami jednostkowymi, bez jakichkolwiek błędów.
* noResult – flaga mówiąca 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 mówiąca 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 mówiąca 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).

Przykład 8

#r "../../../NUnit/bin/Debug/NUnit.dll"

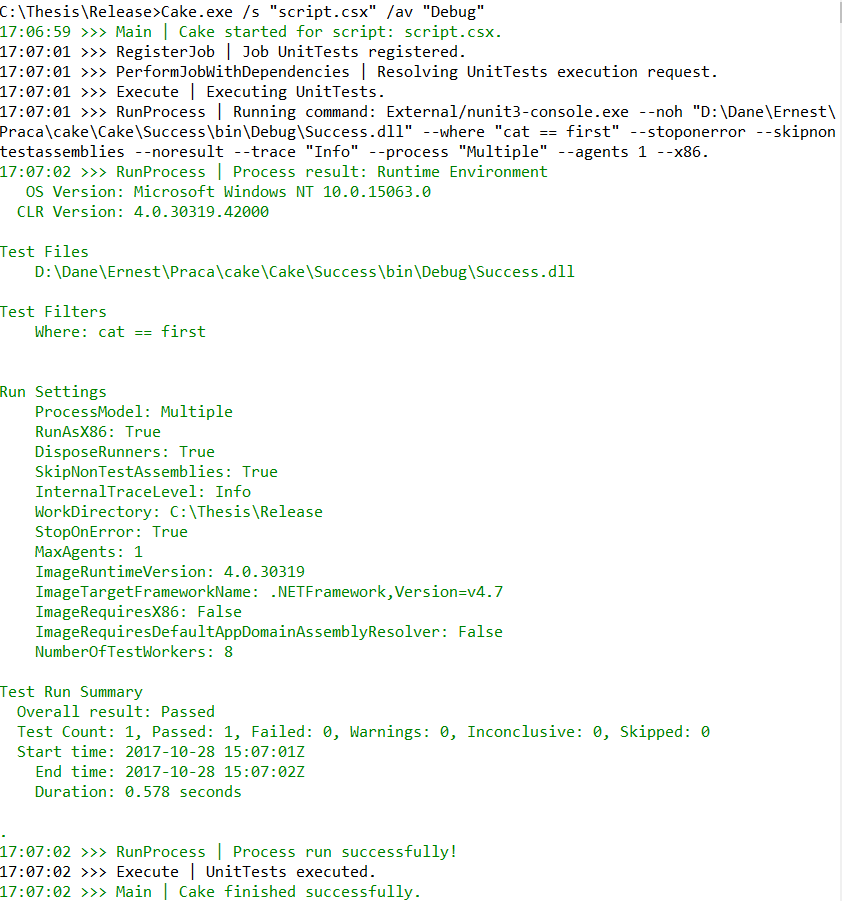
new Job("UnitTests").Does(() => {

return NUnit.Methods.RunTestsWithOptions(@"D:\Dane\Ernest\Praca\cake\Cake\Success\bin\Debug\Success.dll", "cat == first",

null, null, null, null, true, true, true, "Info", null, null, "Multiple", "1", null, null, true);

});

JobManager.SetDefault("UnitTests");



Rysunek 5

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

### Cake.xUnit

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.

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 informację czy wszystkie testy jednostkowe zakończyły się pomyślnie:

bool RunTests (string traits = null, string notraits = null, params string[] assemblyPaths)

Parametry:

* 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.
* assemblyPaths – ścieżki do plików .dll z testami jednostkowymi napisanymi w xUnit.

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 mówiąca czy .NET powinien utworzyć kopie załadowanych bibliotek.
* quiet – flaga mówiąca czy pokazywać wiadomości z postępem testów
* serialize – flaga mówiąca 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.

Przykład 9

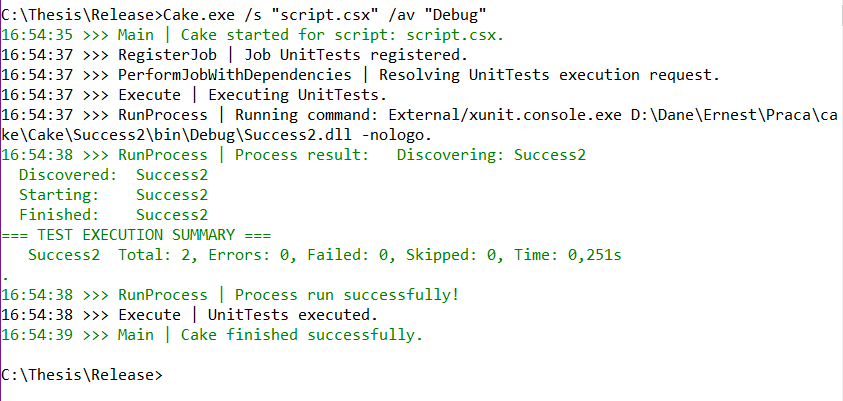
#r "../../../XUnit/bin/Debug/XUnit.dll"

new Job("UnitTests").Does(() => {

return XUnit.Methods.RunTests(null, null, @"D:\Dane\Ernest\Praca\cake\Cake\Success2\bin\Debug\Success2.dll");

});

JobManager.SetDefault("UnitTests");



Rysunek 6

Skrypt z przykładu 9. uruchomi testy jednostkowe napisane z pomocą xUnit, a powyższy zrzut ekranu przedstawia wyniki tych testów.

### Cake.Git

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

bool CommitAllChanges(string message)

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

bool DiffWorkingDir()

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

bool DiffAll()

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

bool ResetHard(string commitShaHash = null)

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

Przykład 10

#r "../../../Git/bin/Debug/Git.dll"

using System;

new VoidJob("GitInit").Does(() =>

{

Methods.RepositoryPath = @"C:\Repository";

Methods.UserIdentity = Identity.FromJsonFile(@"C:\creds.json");

});

new Job("Commit").DependsOn("GitInit").Does(() =>

{

string input;

Logger.Log(LogLevel.Info, "These changes will be committed:");

Methods.DiffWorkingDir();

Logger.Log(LogLevel.Warn, "Commit these changes? (y/n)");

do input = Console.ReadLine();

while (input?.ToLower() != "y" && input?.ToLower() != "n");

return input.ToLower() == "y" && Methods.CommitAllChanges();

});

new Job("Pull").DependsOn("Commit").Does(Methods.Pull);

new Job("Push").DependsOn("Pull").Does(Methods.Push);

JobManager.SetDefault("Push");

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 zrzut ekranu przedstawia wynik wykonania skryptu.

Rysunek 7

### Cake.Minify

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).

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 informację 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 mówiąca 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 informację 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 mówiąca 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 informację 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 mówiąca czy ignorować wielkość liter podczas dopasowywania wzorca.

## Tworzenie własnych modułów rozszerzających

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. Dzięki temu uzyskujemy modularność naszego rozwiązania.

Gdy napiszemy moduł rozszerzający nasz program, skompilowany plik .dll może znajdować się w dowolnym miejscu na dysku. Wystarczy użyć dyrektywy wspomnianej w podrozdziale 3.3.3:

#r „X:\Path\To\Your\Library\library.dll”.

Przestrzenie nazw użyte w dołączanym pliku będą również zaimportowane.

## 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 11

#r "../../../Build/bin/Debug/Build.dll"

#r "../../../NUnit/bin/Debug/NUnit.dll"

#r "../../../Zip/bin/Debug/Zip.dll"

#r "../../../Files/bin/Debug/Files.dll"

#r "../../../Minify/bin/Debug/Minify.dll"

new Job("BuildSolution").Does(() => {

return Build.Methods.BuildSolution(@"D:\Dane\Ernest\Praca\cake\Cake\Cake.sln", @"D:\Dane\Ernest\Praca\TestOutput\", "Release");

});

new Job("MinifyJs").DependsOn("BuildSolution").Does(() => {

return Minify.Methods.MinifyJs(@"D:\Dane\Ernest\Praca\cake\Cake\scripts\js\*", null, @"D:\Dane\Ernest\Praca\TestOutput\");

});

new Job("BundleJs").DependsOn("MinifyJs").Does(() => {

return Minify.Methods.BundleFiles(@"D:\Dane\Ernest\Praca\TestOutput\\*min.js", @"D:\Dane\Ernest\Praca\TestOutput\bundled.min.js", ';');

});

new Job("BundleHtml").DependsOn("BundleJs").Does(() => {

return Minify.Methods.BundleFiles(@"D:\Dane\Ernest\Praca\cake\Cake\scripts\\*html", @"D:\Dane\Ernest\Praca\TestOutput\bundled.html", '\n');

});

new Job("RunUnitTests").DependsOn("BundleJs").Does(() => {

return NUnit.Methods.RunTests(null, null, @"D:\Dane\Ernest\Praca\cake\Cake\Success\bin\Debug\Success.dll");

});

new Job("ZipOutput").DependsOn("RunUnitTests").Does(() => {

return Zip.Methods.ZipFiles(@"D:\Dane\Ernest\Praca\ZippedOutput.zip", @"D:\Dane\Ernest\Praca\TestOutput\");

});

new Job("CopyZip").DependsOn("ZipOutput").Does(() => {

return Files.Methods.CopyFile(@"D:\Dane\Ernest\Praca\ZippedOutput.zip", @"D:\Dane\Ernest\Praca\Deploy\ZippedOutput.zip");

});

JobManager.SetDefault("CopyZip");

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 12

#r "../../../Minify/bin/Debug/Minify.dll"

#r "../../../NUnit/bin/Debug/NUnit.dll"

#r "../../../Git/bin/Debug/Git.dll"

new VoidJob("GitInit").Does(() =>

{

Git.Methods.RepositoryPath = PathManager.RepositoryPath;

Git.Methods.UserIdentity = Identity.FromJsonFile(@"C:\creds.json");

});

new Job("RunTests").Does(() => NUnit.Methods.RunTests(

conditions: null, config: null,

assemblyPaths: PathManager.RelativeToRepo("Tests.dll")));

new Job("Commit").DependsOn("RunTests", "GitInit").Does(() =>

{

string input;

Logger.Log(LogLevel.Info, "These changes will be committed:");

Git.Methods.DiffWorkingDir();

Logger.Log(LogLevel.Warn, "Commit these changes? (y/n)");

do input = System.Console.ReadLine();

while (input?.ToLower() != "y" && input?.ToLower() != "n");

return input.ToLower() == "y" && Git.Methods.CommitAllChanges();

});

new Job("Pull").DependsOn("Commit").Does(Git.Methods.Pull);

new Job("Push").DependsOn("Pull").Does(Git.Methods.Push);

new Job("MinifyCss").DependsOn("Push")

.Does(() => Minify.Methods.MinifyCss(

PathManager.RelativeToRepo(@"\*\*\\*s"),

PathManager.RelativeToRepo(@"\*\*\\*.mins")

));

new Job("MinifyJs").DependsOn("Push")

.Does(() => Minify.Methods.MinifyJs(

PathManager.RelativeToRepo(@"\*\*\\*.js"),

PathManager.RelativeToRepo(@"\*\*\\*.min.js")

));

new Job("BundleCss").DependsOn("MinifyCss")

.Does(() => Minify.Methods.BundleFiles(

PathManager.RelativeToRepo(@"\*\*\\*.mins"),

PathManager.RelativeToRepo("custom.mins"),

excludePattern: PathManager.RelativeToRepo("custom.mins")

));

new Job("BundleJs").DependsOn("MinifyJs")

.Does(() => Minify.Methods.BundleFiles(

PathManager.RelativeToRepo(@"\*\*\\*.min.js"),

PathManager.RelativeToRepo("custom.min.js"),

excludePattern: PathManager.RelativeToRepo("custom.min.js")

));

new VoidJob("Publish").DependsOn("BundleJs", "BundleCss");

JobManager.SetDefault("Publish");

static class PathManager

{

public const string RepositoryPath = @"C:\Repository";

public static string RelativeToRepo(string path) => System.IO.Path.Combine(RepostoryPath, path);

}

## Uwagi końcowe

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.Compilersharp, 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.

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.

# Wnioski

Nasz system spełnia swoją podstawową funkcję – pozwala na automatyzację procesów, a tym samym na oszczędność czasu oraz zmniejsza podatność na błędy, które mogłyby powstać podczas wielokrotnego ręcznego wykonywania powtarzalnych czynności.

Skrypty podobne do tych przedstawionych powyżej wystarczy zdefiniować raz, co w stosunku do ręcznego wykonywania tych czynności nie jest czasochłonne. Można je później wykorzystywać regularnie w procesach związanych z kompilacją i wdrażaniem programów dla platformy .NET. Dzięki modularyzacji i ograniczonemu skomplikowaniu naszego systemu, użytkownicy mogą szybko zapoznać się z możliwościami i działaniem programu oraz dostosować go do swoich potrzeb.

W naszym systemie udało się nam zaimplementować wszystkie założone funkcjonalności. Niektóre z modułów ograniczone są do narzędzi jednego dostawcy – np. posiadamy obsługę funkcji systemu kontroli wersji, ale tylko dla repozytoriów GIT. Co więcej, w przypadku testów jednostkowych – obsługujemy NUnit oraz xUnit, ale nie obsługujemy MSTest. Pomimo tych braków, nasz system może być w łatwy sposób rozszerzony przez użytkowników o obsługę potrzebnego narzędzia.

Podczas tworzenia systemu poznaliśmy wiele narzędzi ułatwiających pracę programistów stworzonych do użytku w różnych dziedzinach oraz językach programowania, w szczególności tych przeznaczonych do wdrażania aplikacji .NET.

# Bibliografia

1.

Arh, D., 2016. *DotNetCurry.* [Online]   
Available at: http://www.dotnetcurry.com/csharp/1258/dotnet-platform-compiler-roslyn-overview  
[Data uzyskania dostępu: 15 10 2017].

2. Harrison, N., 2017. W: *Code Generation with Roslyn.* Lexington: Apress, pp. 42-48.

3. Maddock, C., 2017. *GitHub.* [Online]   
Available at: https://github.com/nunit/docs/wiki/Console-Command-Line  
[Data uzyskania dostępu: 15 10 2017].

4. Mast, K. v. d., 2016. W: *Gulp Succinctly.* Morrisville: Syncfusion, pp. 23-27.

5.Mayo, J., 2015. W: *C# Succinctly.* Morrisville: Syncfusion, pp. 97-98.

6. Microsoft, 2017. *GitHub.* [Online]   
Available at: https://github.com/dotnet/roslyn/wiki/Scripting-API-Samples  
[Data uzyskania dostępu: 15 10 2017].

7. Price, M. J., 2017. W: *C# 7 and .NET Core: Modern Cross-Platform Development - Second Edition.* Birmingham: Packt Publishing, pp. 89-90.

8. Rossel, S., 2016. W: *Object-Oriented Programming in C# Succinctly.* Morrisville: Syncfusion, pp. 42-44.

# Spis rysunków

[Rysunek 1 20](#_Toc495501756)

[Rysunek 2 21](#_Toc495501757)

[Rysunek 3 26](#_Toc495501758)

[Rysunek 4 27](#_Toc495501759)

[Rysunek 5 30](#_Toc495501760)

[Rysunek 6 32](#_Toc495501761)

[Rysunek 7 35](#_Toc495501762)