Koncepcja ciągłej integracji • DDD • .NET 4.5 • C++ i Python • Windows Phone

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

programista

4/2012 (04)

Cena 9.90 zł (w tym VAT 23%)

WZNIEŚ SIĘ NA WYŻYNY PROGRAMOWANIA W

Visual Studio 2012

Loose coupling w PHP - nowości w Symfony2

Dowiedz się czym jest i jakie korzyści niesie ze sobą wstrzykiwanie zależności

Co nowego w języku C# - async i await

Blokowanie i zawieszanie się interfejsu użytkownika stało się przeszłością

Bot Gadu-Gadu w chmurze Google

GG BotAPI - stwórz własne rozwiązanie do komunikacji między współpracownikami

ISSN 2084-9400

9 772084 940206 09

Microsoft Visual Studio

bns it #

ALGORYTMY DOGADYWANIA SIĘ

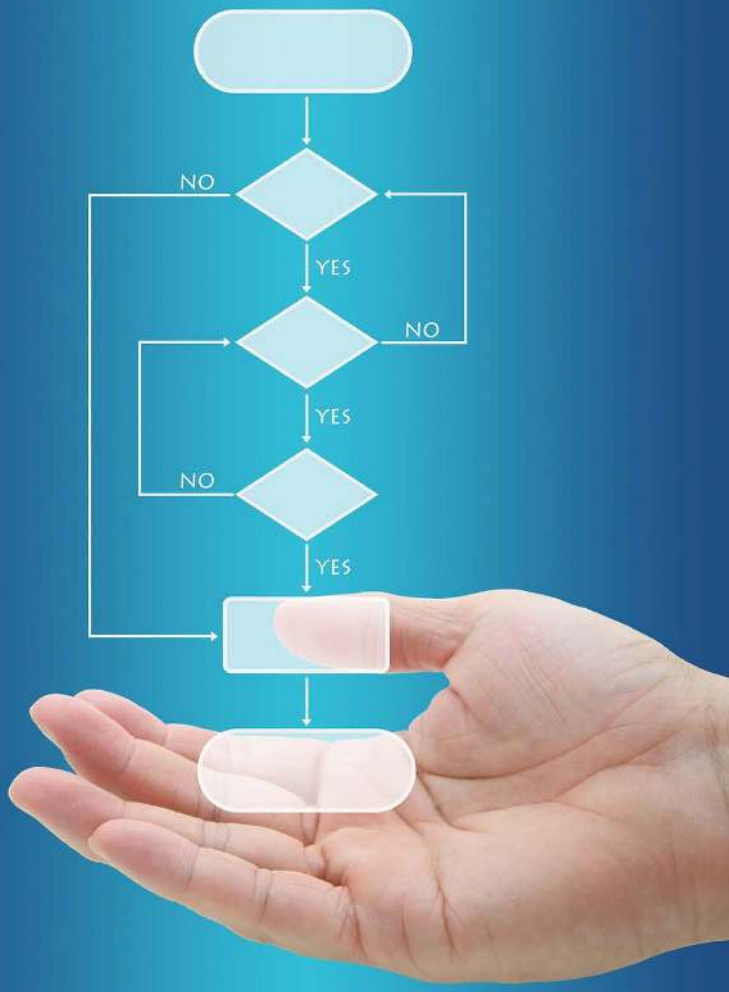
JAK ROZMAWIAĆ Z KLIENTEM, KTÓRY NIE WIE, CZEGO CHCE?

Szkolenie otwarte: 12-13.11.2012 (Łódź)

JAK WYKORZYSTYWAĆ WIZJĘ DO PRECYZOWANIA ZAKRESU PRODUKTU?

CZYM WŁAŚCIWIE JEST „MYŚLENIE BIZNESOWE"?

Naucz się rozmawiać z klientem, który nie wie, czego chce.



JAK SPRAWNIE OKREŚLIĆ PRIORYTETY WYMAGAŃ?

JAK RADZIĆ SOBIE Z TRUDNYMI KOMUNIKATAMI ZE STRONY KLIENTA?

Trenerem prowadzącym szkolenie jest Michał Bartyzel. Od 2004 roku pracuje w branży IT. Przez kilka lat tworzył oprogramowanie dla branży doradczej. Zdobyte doświadczenia spisał w książce pt. Oprogramowanie szyte na miarę. Jak rozmawiać z klientem, który nie wie, czego chce

www.AlgorytmyDogadywaniaSie.pl

SZCZEGÓŁY I ZGŁOSZENIE

programista

Magazyn Programista wydawany jest przez firmę Anna Adamczyk

Wydawca:
Anna Adamczyk
annaadamczyk@programistamag.pl

Redaktor naczelny:
Łukasz Łopuszański
lukaszlopuszanski@programistamag.pl

Redaktor prowadzący:
Rafał Kocisz
rafal.kocisz@gmail.com

Korekta:
Tomasz Łopuszański

Kierownik produkcji:
Krzysztof Kopciowski
bok@keylight.com.pl

DTP:
Krzysztof Kopciowski

Dział reklamy:
reklama@programistamag.pl
tel. +48 663 220 102
tel. +48 604 312 716

Prenumerata:
prenumerata@programistamag.pl

Współpraca:
Michał Bartyzel
Mariusz Sieraczkiewicz
Artur Machura
Marek Sawerwain
Łukasz Mazur
Rafał Kułaga
Sławomir Sobótka
Michał Mac

Adres wydawcy:
Dereniowa 4/47
02-776 Warszawa

Druk:
Estrella Sp.z o.o,
Tel/fax 22 625 08 68
ul. Szaflarska 9,
01-812 Warszawa

Koniec wakacji. Powroty z urlopów. Nowe projekty, nowe wyzwania, nowe problemy, z którymi przyjdzie nam się uporać. To jest właśnie to, co lubimy w zawodzie programisty. Każdy kolejny projekt to nowa przygoda, nowe technologie, możliwość nauczenia się czegoś interesującego.

Oczywiście wymaga to wielu poświęceń, jednakże satysfakcja z dobrze wykonanej pracy warta jest wysiłku. Kiedyś zastanawialiśmy się, kto bądź co jest największym wrogiem programisty i doszliśmy do wniosku, iż jest to CZAS. Każdy niemalże adept tego zawodu (przynajmniej spośród tych, których znamy) narzeka notorycznie na brak czasu. Jak się dobrze zastanowić, jest to dość naturalne zjawisko: no bo jak tu nie narzekać na brak czasu w zawodzie, który wymaga ciągłego samorozwoju? A terminy w projektach gonią... I tu właśnie pojawiamy się my, z naszą rolą, misją i zadaniem, którym jest nieustanna walka o więcej wolnego czasu dla Ciebie. Jak oni tego chcą dokonać, pytasz pewnie sam siebie? Dostając starannie dobrane, wyselekcjonowane materiały do nauki niewątpliwie zyskujesz na czasie. Liczymy, że i tym razem się na nas nie zawiedziesz...

Jak zapowiadaliśmy ostatnio, tematem przewodnim w aktualnym, czwartym już numerze „Programisty" jest nowa odsłona pakietu Microsoft Visual Studio. Narzędzie to od wielu lat cieszy się niesłabnącą popularnością wśród twórców oprogramowania. Nowa wersja tego narzędzia przynosi wiele zmian. Czy są to zmiany na lepsze, czy na gorsze możesz przekonać się, czytając artykuł Pawła Łukasika „Visual Studio 2012 – rewolucja czy ewolucja?".

Poza tym warto zwrócić uwagę na tekst o tzw. Ciągłej Integracji (ang. Continuous Integration). Jeśli nie słyszałeś nigdy o tym zagadnieniu, to MUSISZ przeczytać ten artykuł. Bardzo interesujący jest również tekst o tworzeniu botów dla platformy Gadu-Gadu.

Jak zwykle w magazynie nie zabraknie tematów związanych z nowoczesnym programowaniem w języku C++; tym razem możesz dowiedzieć się, jak połączyć C++ z językiem Python za pomocą biblioteki boost_python. Poza tym znajdziesz u nas cały szereg innych, ciekawych tekstów m.in. o C#, .NET 4.5, PHP, kolejną część z serii o DDD oraz wiele innych (sprawdź sam!).

Zgodnie z naszą tradycją, kilka słów też o tym, co w kolejnym wydaniu. Numer piąty zapowiada się niezwykle ciekawie, gdyż tematem przewodnim będzie... programowanie gier. Co więcej, sięgając po ten numer, poznasz sztuczki i kruczki zastosowane przy implementacji jednej z najbardziej znanych polskich gier „mobilnych" ostatnich miesięcy (na razie nie zdradzimy jej tytułu; powiemy tylko tyle, że gra polega na ratowaniu kota :)). Poza tym, wychodząc naprzeciw Waszym oczekiwaniom, zdecydowaliśmy się wprowadzić nowy dział: Klub Dobrej Książki.

Więcej szczegółów o tym dziale już wkrótce, a póki co życzymy miłego (i owocnego!) czytania bieżącego numeru.

SPIS TREŚCI

Paweł Łukasik

Visual Studio 2012 – rewolucja czy ewolucja?

Visual Studio 2012 wzbudza kontrowersje od momentu pojawienia się jego najwcześniejszych wersji. Dość radykalne zmiany kolorystyki (a raczej jej pozbawienie) oraz wszechobecne użycie wielkich liter sprawiły, że programiści skupili się na komentowaniu strony wizualnej, pomijając kluczowe zmiany i nowe funkcje. Warto jednak zapoznać się z tym, co oferuje najnowsza wersji IDE.

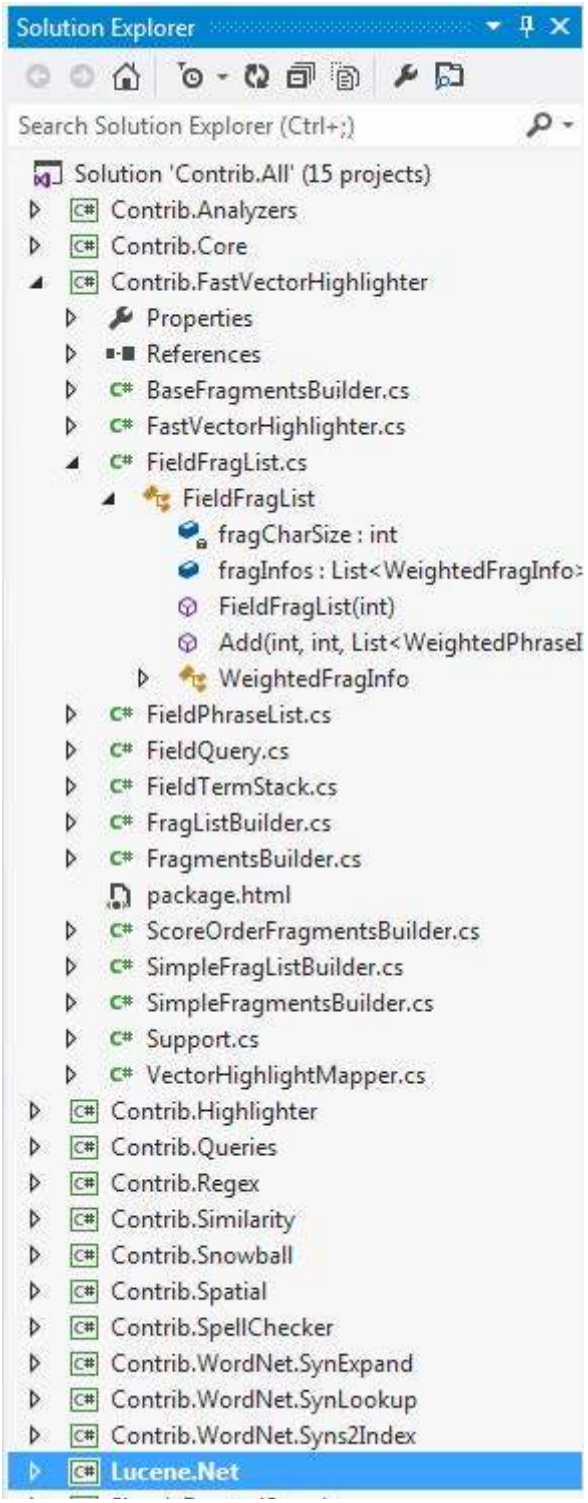
Visual Studio to w zasadzie jedyne środowisko pracy dla osób programujących na platformie Windows z wykorzystaniem stosu firmy Microsoft. Zmiany wprowadzane w kolejnych wersjach IDE są ważne z punktu widzenia dość znacznej liczby programistów zarówno w Polsce, jak i na świecie. Zobaczmy, jakie nowe możliwości dostajemy w stosunku do wersji 2010. W artykule przyjrzymy się wersji Professional, która jest najuboższą w funkcje wersją płatną edytora. Nie będziemy omawiać nowych typów projektów wspieranych przez wersję 2012 czy też nowinek językowych, a skupimy się jedynie na samym edytorze.

SOLUTION EXPLORER

Solution Explorer został dość mocno rozbudowany. Choć dla osób korzystających już wcześniej z pakietu Productivity Power Tools zmiany nie będą rewolucyjne, warto się z nimi zapoznać. Pierwsze, co rzuca się w oczy, to możliwość zajrzenia do pliku bez wyświetlania go w edytorze. Od razu z poziomu okna możemy zobaczyć zawarte w pliku klasy, metody, pola czy właściwości. Dzięki temu oszczędzimy czas na otwieraniu plików i szybciej zapoznamy się z jego zawartością. Wprowadzona zmiana w sposobie wyświetlania plików jest kluczowa, gdyż wpływa także na inne aspekty pracy z tym oknem narzędziowym. Operacje, działające na plikach solucji, uwzględniają te dodatkowe poziomy. I tak np. znajdujący się u góry Search po wpisaniu tekstu zwróci nie tylko pliki o nazwie pasującej do wzorca, ale również te, które zawierają pasujące do niego metody czy właściwości. Gdyby z jakiegoś powodu zwracana była zbyt duża lista pasujących elementów, zawsze możemy ją zawęzić. Służy do tego nowe polecenie menu kontekstowego, a mianowicie *Scope to This*. Jego użycie spowoduje, że Solution Explorer ograniczy listę wyświetlanych elementów i wszelkie operacje będzie wykonywał na tym zbiorze. Widok możemy zawęzić do projektu, pliku czy też pojedynczej klasy. Aby wrócić do poprzednich widoków, możemy skorzystać z opcji *Back*, która wraz z kolejnym poleceniem *Forward* działa jak Undo/Redo, umożliwiając nam poruszanie się po filtrowanych widokach Solution Explorera. Aby zresetować widok do domyślnego, można użyć opcji *Home*, która pokaże nam pełny widok projektów.

Dodatkowo na zawężonej liście elementów mamy możliwość przejścia do kolejnych widoków. Menu kontekstowe na pliku, klasie czy właściwości daje nam dodatkowe możliwości. Są to:

- Calls
- Is Called by
- Is Used by



Rysunek 1. Nowy Solution Explorer daje więcej możliwości do działania

Opcja Calls pokaże nam wszystkie wywołania, z których korzysta analizowana metoda. Dla przykładu, gdy będziemy analizować metodę **ToString** z Listingu 1, lista zawierać będzie wywołania StringBuilder'a oraz metody **Append** i **ToString**.

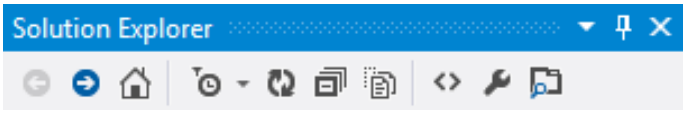
Listing 1. Analizowana metoda ToString()

```
public override string ToString()
{
   StringBuilder sb = new StringBuilder();
   sb.Append(text).Append('(');
   foreach (Toffs to in termsOffsets)
      sb.Append(to.ToString());
   sb.Append(')');
   return sb.ToString();
}
```

Ta sama metoda z uruchomionym poleceniem **Is Called** by pokaże nam miejsca w naszej aplikacji, gdzie dany kod jest używany. Ta opcja ma zastosowanie oczywiście tylko do metod. Ostatnia z opcji – **Is Used By** – pokazuje miejsca, gdzie występuje dany element.

W przypadku klas mamy dodatkowo możliwość zobaczenia ich hierarchii. Dostępne są dwie nowe opcje - Base Types oraz Derived types. Pierwsza opcja pokaże typy bazowe, a druga te, które z naszej klasy dziedziczą.

Górna belka narzędziowa również została rozbudowana o dodatkowe ikony.

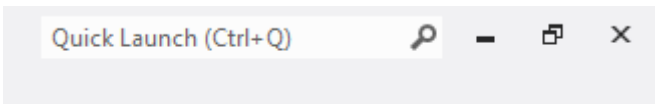


Rysunek 2. Górna belka narzędziowa

Choć opcje w belce narzędziowej Solution Explorera zmieniają się w zależności od kontekstu, to w najszerszej opcji w belce narzędziowej Solution Explorera mamy:

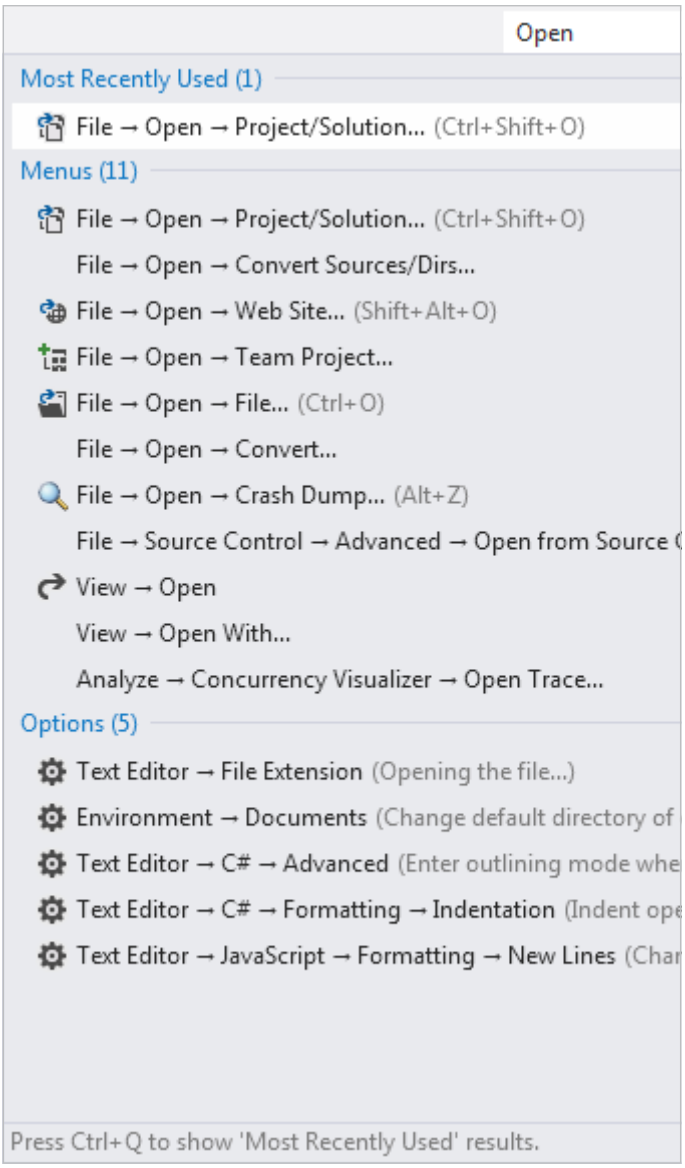
- Back oraz Forward – do przechodzenia po historii widoków
- Home – przywracający domyślny widok plików w solucji
- Filters – opcja filtrów. Pozwala zawęzić listę wyświetlanych plików zgodnie z kryterium
- Refresh – odświeża i aktualizuje widok
- Collapse All – zwija całą solucję do widoku projektów
- Show All Files – pokazuje wszystkie pliki – także te niewłączone do projektu
- View Code – wyświetla zawartość pliku
- Properties – wyświetla kartę właściwości danego elementu
- Preview Selected Items – po wybraniu tej opcji, Visual Studio automatycznie otworzy plik po jego zaznaczeniu na liście

QUICK LAUNCH



Rysunek 3. Pole QuickLauch to teraz rozbudowane centrum operacyjne

Quick Launch to znane z poprzedniej wersji pole *Find* na sterydach. To niepozorne pole tekstowe jest zawsze dostępne, a przejść do niego możemy, wciskając CTRL + Q. Tak jak wspomniałem wcześniej, jest to ewolucja pola *Find*, gdyż jego możliwości zostały dość mocno rozbudowane, a prezentacja wyników znacznie ulepszona. Za jego pomocą dostaniemy się do praktycznie każdego zakątka Visual Studio. Po wpisaniu ciągu znaków zostaną przeszukane pozycje w menu opcji konfiguracyjnych, lista ostatnio oraz aktualnie otwartych dokumentów. Jeśli liczba prezentowanych wyników dla danego słowa kluczowego jest zbyt duża, mamy możliwość zastosowania filtrów ograniczających wyniki do konkretnych kategorii. I tak wprowadzając przed szukanym słowem **@mru** (*Most recently used*), ograniczymy listę wyników od ostatnio użytych opcji. Dodanie **@menu** zadba o to, aby wyniki tyczyły się tylko pozycji, które znajdziemy w menu Visual Studio. Aby przeszukać nazwy i ścieżki otwartych dokumentów, wystarczy dodać **@doc** na początek naszej frazy wyszukiwania. Należy jednak pamiętać, że opcja ta nie przeszukuje zawartości dokumentów. Szybsze wyszukiwanie opcji załatwimy przedrostkiem **@opt**



Rysunek 4. Wyniki wyszukiwania w okienku Quick Launch

Dodatkowo dostaliśmy możliwość konfiguracji działania tego elementu. Jeśli jest on nam niepotrzebny, możemy go wyłączyć. Możemy także wyłączyć poszczególne źródła danych. Jeśli nie korzystamy z wyników np. Ostatnio użytych elementów (**@mru**), możemy je w ustawieniach wyłączyć i nie pojawią się one nam na liście wyników. Przydatne, gdy dostajemy sporo wyników, a w rzeczywistości nie potrzebujemy widzieć niektórych z nich. Ostatnia z możliwości konfiguracyjnych to możliwość

prezentacji wyników poprzedniego wyszukiwania. Użyteczne, jeśli często korzystamy z tej samej funkcji – wtedy od razu po przejściu do pola *Quick Find* mamy wyniki i możemy z danej opcji użyć bez potrzeby wprowadzania szukanej frazy.

ERROR LIST

O tym, że Visual Studio zmieniło się znacząco od poprzedniej wersji, niech świadczy fakt, że nawet takie drobne okno jak *Error* list także doczekało się jakiejś nowości. Dodano do niego możliwość filtrowania i przeszukiwania listy błędów. Z lewej strony okna dodano ikonę lejka, za pomocą której możemy przefiltrować listę błędów pod kątem przynależności do jednej z grup. Do wyboru mamy otwarte dokumenty (*Open Documents*), aktualnie wybrany projekt (*Current Project*) oraz aktualnie wyświetlany dokument (*Current Document*). Gdyby to nadal powodowało niemożliwość znalezienia poszukiwanego przez nas błędu, mamy dodatkowe oręże. Po przeciwległej stronie okna znajduje się pole tekstowe umożliwiające wprowadzenie ciągu znaków. Dzięki temu możemy zawęzić listę błędów tylko do tych, które dany ciąg zawierają. Przeszukanie odbywa się po każdej z kolumn, tak więc możemy szukać nie tylko po opisie błędu, ale także po nazwie pliku, wierszu, kolumnie (w której wystąpił dany błąd) czy projekcie.

USPRAWNIONA PRACA Z ZAKŁADKAMI

Praca z zakładkami to kolejny ukłon w stronę programistów posiadających dwa lub więcej monitorów. Visual Studio 2010 wprowadził pracę na zakładkach, ale była ona daleka od ideału. Dwuklik na zakładce często nieintencjonalny powodował jej wypięcie z głównego okna IDE i trzeba było ją potem wpinać do niego ponownie. Najnowsza wersja eliminuje ten problem i zakładkę można wypiąć, przeciągając ją poza główne okno. Dzięki temu mniej będzie przypadkowego działania i mniej frustracji. W nowej wersji zakładki można także grupować ze sobą, tak więc niekoniecznie muszą być porozrzucane na drugim monitorze, ale mogą być zebrane logicznie w kilka dodatkowych okien. Poprawiono także samo wsparcie dla wielu monitorów. Poprzednio, gdy główne okno Visual Studio było aktywne, aktywne stawały się także wszelkie okna, które zostały z niego wyciągnięte. Skutecznie uniemożliwiało to model pracy, w którym na jednym ekranie mamy Visual Studio, a na drugim kilka wyciągniętych z IDE okien i np. przeglądarkę, w której obserwujemy efekty naszej pracy. Za każdym razem, gdy Visual Studio stawało się aktywne, przeglądarka była przykrywana przez pozostałe okienka VS. W najnowszej wersji zostało to naprawione i możemy bez przeszkód pracować w takim modelu na dwóch czy więcej monitorach.

KOLORYSTYKA

Omawiając zmiany w Visual Studio, nie sposób przejść obojętnie nad wspomnianymi już zmianami kolorystyki. W porównaniu do poprzedniej wersji ubyło kolorów. W Developer Preview nie było ich praktycznie wcale. Dopiero zdecydowane reakcje społeczności dały skutek i zarówno Beta, jak i finalny produkt trochę kolorów posiada. Wprowadzono także schematy kolorystyczne. Prócz standardowego jasnego wprowadzono ciemny motyw kolorystyczny. Przełączać się między nimi można za pomocą *Tools → Options*, a następnie wybierając *Environment* i zmieniając *Color theme* na *Dark*. Poniżej porównanie jasnego i ciemnego motywu.

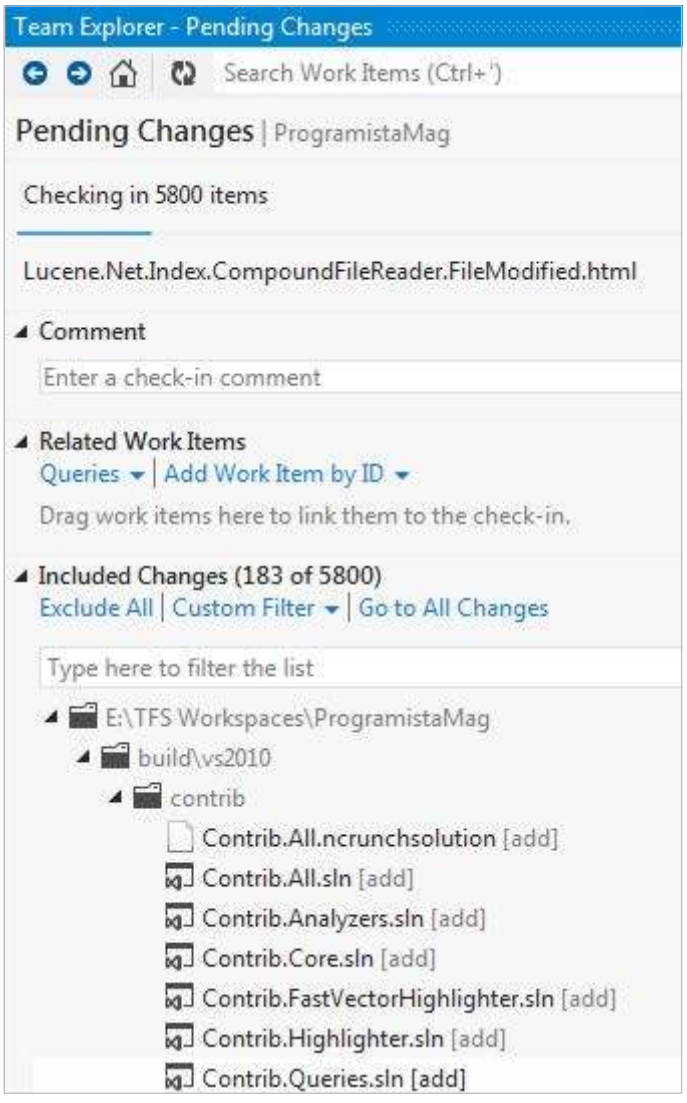
Oczywiście nadal istnieje możliwość dostosowania kolorystyki do naszych indywidualnych potrzeb. Wystarczy przejść do pozycji Fonts & Colors w opcjach (Options).

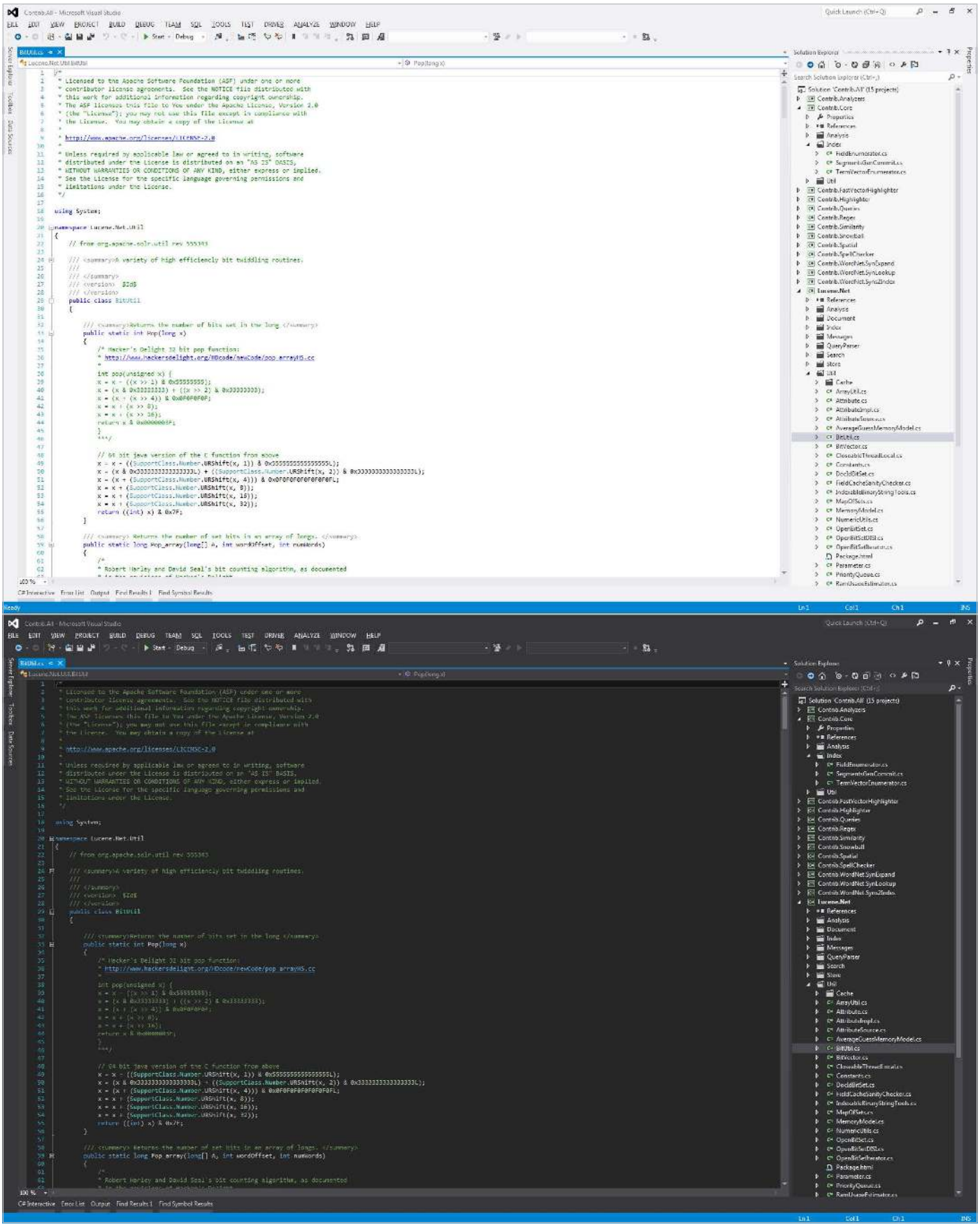
CZY TO JUŻ WSZYSTKO?

Wspomniane wyżej nowości to nie wszystko, co oferuje nowe Visual Studio 2012. Choć tu nie mam twardych liczb na potwierdzenie tezy, to najnowsza wersja IDE sprawia wrażenie szybszej w reagowaniu na akcje użytkownika, a wczytywanie projektów i solucji działa niemal błyskawicznie. Dużym plusem jest brak konieczności konwertowania plików .sln. Pliki utworzone w wersji 2010 można otworzyć w najnowszej wersji i na odwrót. Dzięki temu, jeśli z jakiegoś powodu musimy korzystać z obu wersji, nie będzie z tym problemów. Oczywiście może się zdarzyć tak, że niektóre projekty będą wymagały migracji, ale zgodność plików w obu wersjach zminimalizuje te sytuacje drastycznie.

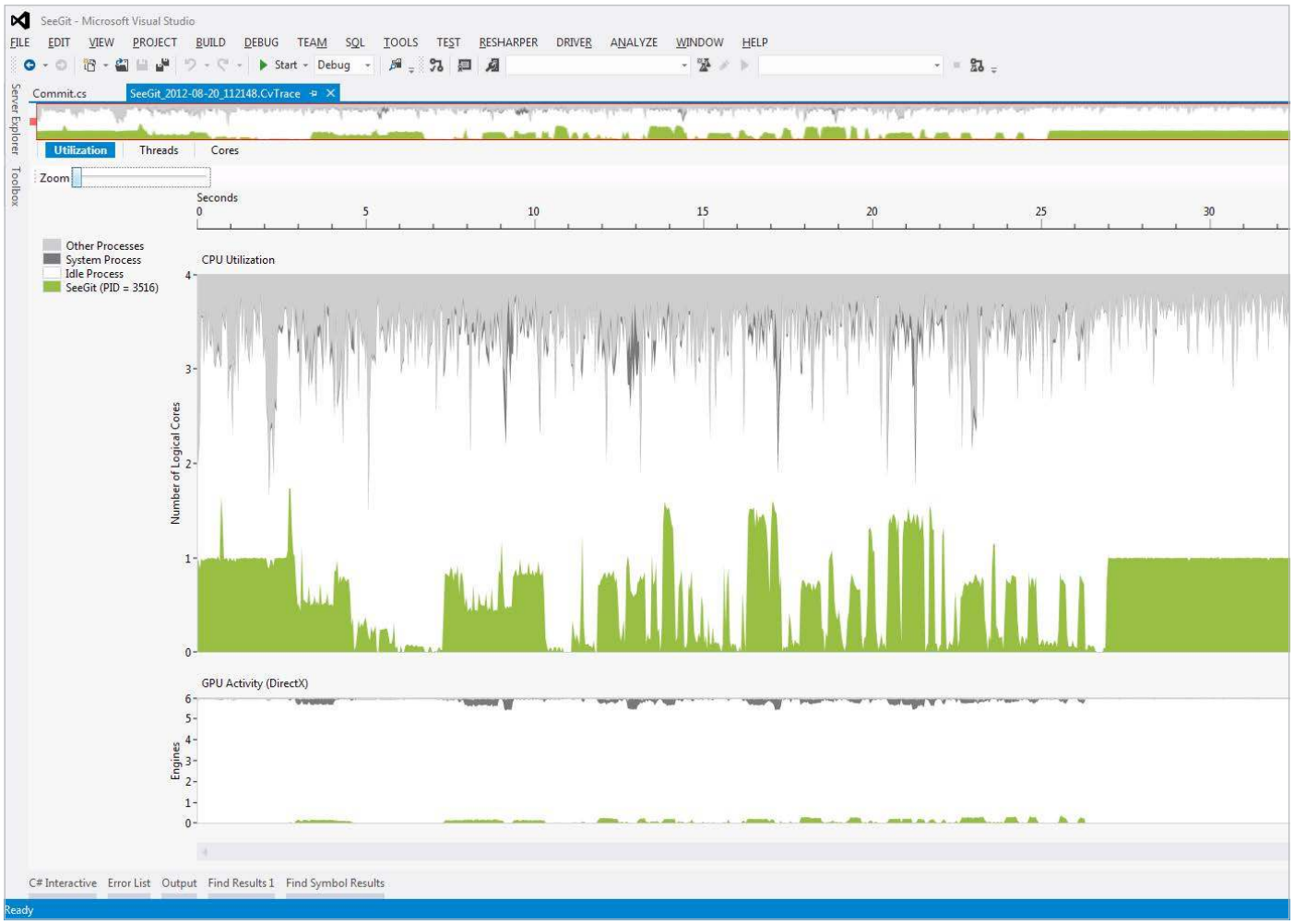
Team Foundation Server w najnowszej odsłonie zyskał kilka nowych funkcji, ale także została odświeżona jego kliencka część, dostępna w Visual Studio. Graficznie zmienione zostały praktycznie wszystkie okna. Team Explorer, Pending Changes oraz Source Control Explorer zostały przystosowane do nowego wyglądu. Więcej akcji dzieje się bez wyskakujących okienek dialogowych, a tym samym nie blokuje głównego okna edytora.

Rysunek 5. Okno Pending Changes podczas pracy





Rysunek 6. Dostępne schematy kolorystyczne w Visual Studio 2012



Rysunek 7. Concurrency Visualizer

Zmieniony został także sposób przeprowadzania review naszego kodu. W najnowszej VS wersji mamy możliwość wysłania prośby o analizę kodu do innego członka zespołu, który nanosi swoje uwagi bezpośrednio w Visual Studio i przekazuje nam je z powrotem. My możemy się z nimi zapoznać i przyjąć bądź je odrzucić. Niestety jest to opcja dostępna powyżej wersji Professional. Ze zmian w odwrotnym kierunku warto wymienić udostępnienie narzędzia Concurrency Visualizer już w najtańszej wersji. Dzięki niemu będziemy mogli uruchomić, zbadać i wykryć ewentualne problemy naszych wielowątkowych aplikacji. Dodatkowo w stosunku do poprzedniej wersji został wzbogacony on o nowe znaczniki umożliwiające orientację, jaki fragment kodu aktualnie jest wykonywany, wsparcie dla operacji wykonywanych na GPU oraz dodatkowe narzędzie uruchamiane z linii poleceń do przeprowadzania analiz na maszynach produkcyjnych bez Visual Studio.

PODSUMOWANIE

Mam nadzieję, że w artykule udało się przedstawić kluczowe zmiany w najnowszym Visual Studio. Nie wyczerpują one jednak wszystkich zmian w edycji 2012. Wiele drobnych zmian, np. dodanie możliwości przeszukiwania w wielu miejscach – *Search Everywhere*, usprawnień doszlifowuje świetny produkt, a zatem jest to solidny następca dobrej wersji 2010. Pomijając kolorystykę, nie wprowadza rewolucyjnych zmian, a raczej łata błędy oraz dodaje rzeczy, które ułatwiają pracę programisty (zmiany z Productivity Power Tools). Wszystko to sprawia, że praca z najnowszą wersję edytora powinna być przyjemnością.

W sieci

- What's New in Visual Studio 2012 RC: http://msdn.microsoft.com/en-us/library/bb386063(v=vs.110).aspx

Paweł Łukasik pawel@octal.pl, http://pawlos.blogspot.com

Programista pasjonat. Z komputerami związany od najmłodszych lat za sprawą popularnej na tamte czasy maszyny Timex 2048. Aktualnie zajmuje się wszelkimi sprawami związanymi z komputerami i programowaniem, prowadząc własną firmę. Współzałożyciel Wrocławskiej Grupy .NET oraz założyciel strony http://dotnetomaniak.pl

Robert Nowak

Wydajne i elastyczne programy

Łączenie C++ i Pythona przy pomocy boost_python

Aplikacje wydajne najwygodniej tworzy się w językach kompilowanych do kodu maszynowego, możemy wtedy wykorzystać wszystkie możliwości, które daje sprzęt. Rozwiązania elastyczne tworzymy, wykorzystując interpreter – nie ma potrzeby translacji do kodu maszynowego. W artykule omówiono przykład stosowania obu podejść jednocześnie dla języków C++ i Python. Komunikację pomiędzy modułami tej samej aplikacji, utworzonymi w różnych tych językach, upraszcza biblioteka **boost_python**.

Języki kompilowane translują kod źródłowy programu komputerowego do kodu binarnego lub kodu pośredniego i zapisują wynik translacji. Jeżeli wynikiem kompilacji jest kod binarny, algorytmy mogą być zaimplementowane wydajnie – procesor nie robi niczego innego, tylko wykonuje nasz kod. Podejście takie ma pewną wadę: kod binarny jest nieprzenośny, nie można go wykonać na procesorze z inną architekturą niż procesor, dla którego wykonano kompilację. Ze względu na wykorzystywanie różnych udogodnień systemu operacyjnego, nasz program w wersji binarnej może nie działać poprawnie na komputerze z innym (niż przeznaczono) systemem operacyjnym. Jeżeli chcemy zapewnić przenośność takiego programu, to tylko na poziomie kodu źródłowego, dla każdej platformy potrzebna będzie niezależna kompilacja.

Języki interpretowane wykonują kod źródłowy, translacja odbywa się ,,na bieżąco''. Nie ma podziału na kod źródłowy i kod wynikowy (np. binarny), procesor wykonując nasz kod, jednocześnie go transluje. Programy nie mogą więc być tak wydajne, jak te poprzednie. Ponieważ mamy do czynienia tylko z kodem źródłowym, programy są przenośne, działają podobnie, niezależnie od systemu operacyjnego, architektury komputera czy architektury procesora.

Kod binarny jest bardzo trudny do czytania przez człowieka, głównie z powodu swojej objętości (miliony instrukcji …), dlatego dostarczając program w wersji binarnej zakładamy, że nasze rozwiązania są ukryte przed użytkownikiem. Nie są więc elastyczne, użytkownik nie może ich modyfikować. Kod źródłowy natomiast jest, a przynajmniej powinien być, czytelny, dostęp do niego pozwala poznać i zrozumieć strukturę programu i użyte algorytmy, tym samym je modyfikować i dostosowywać do własnych potrzeb. Programy pisane w językach interpretowanych są elastyczne, ponieważ użytkownik ma zagwarantowany dostęp do kodu źródłowego.

W naszych aplikacjach istnieją fragmenty, które powinny być wydajne, więc należy je tworzyć w językach kompilowanych, oraz fragmenty, które wygodniej dostarczać jako fragmenty kodu interpretowanego, aby umożliwić użytkownikowi dostosowywanie aplikacji do własnych potrzeb. Pewne moduły chcemy chronić, ukryć rozwiązania przed użytkownikiem, więc dostarczamy je w formie binarnej. Języki interpretowane wybieramy także ze względu na to, że implementacja trwa krócej: nie trzeba kompilować i mamy bogatszy zbiór udogodnień, np. możliwość tworzenia nowych klas w czasie działania programu.

Jeżeli chcemy, aby nasz produkt miał kilka wymienionych cech (wydajny, elastyczny, szybka implementacja, ukryte fragmenty zawierające istotne algorytmy), to warto rozważyć wykorzystanie kilku różnych języków programowania. Narzędzia (translatory) pośrednie, które mają cechy kompilatorów i interpreterów (np. kompilacja do kodu pośredniego, kompilacja przy pierwszym uruchomieniu), nie pozwalają osiągnąć takiej wydajności, jak kompilatory do kodu binarnego i takiej elastyczności jak interpretery. Stosowanie wielu języków jednocześnie wymaga większych umiejętności niż używanie tylko jednego, trzeba je znać i zapewnić komunikację pomiędzy modułami utworzonymi w różnych językach. Dodatkowo trzeba starannie opracować interfejsy modułów, uwzględniając ograniczenia języków, które będą z nich korzystać. W wielu wypadkach proponowane podejście jest uzasadnione. Warto stosować specjalistyczne narzędzia (a takim jest język programowania) w zależności od potrzeb, przy okazji zmniejszając nakład pracy (np. rezygnując z tworzenia własnego interpretera do plików konfiguracyjnych, który będzie działał w aplikacji dostarczanej binarnie).

Artykuł przedstawia sposoby łączenia fragmentów kodu napisanych w C++ (język kompilowany do kodu binarnego) i w Pythonie (język interpretowany), przy użyciu biblioteki **boost_python**. Biblioteka ta pozwala tworzyć moduły binarne, które mogą być używane w Pythonie jak pakiety. Dostarcza także obiektowy interfejs do interpretera Pythona, upraszczając jego używanie w aplikacjach tworzonych w C++.

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do interpretera języka Python (http://python.org), kompilatora C++ (http://gcc.gnu.org, http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express lub http://mingw.org) oraz edytora tekstu. Dodatkowo należy zainstalować zbiór bibliotek boost (http://www.boost.org). Aby poprawnie zbudować przedstawione przykłady, należy dołączyć bibliotekę **boost_python**. Dla konsolidatora **g++** należy dodać opcję **-lboost_python**, dla konsolidatora Visual Studio (program **link**) dodatkowe opcje nie są potrzebne, biblioteka **boost_python** jest dodawana automatycznie. Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła umieszczono jako materiały pomocnicze. Materiały te wykorzystują narzędzie do budowania aplikacji Scons (http://scons.org). Przykłady testowano na platformie Ubuntu 11.10 64 bit i Windows 7 32 bit, wykorzystując Python2.7, g++ 4.6, Visual Studio 2010 Express, boost 1.49.

PODSTAWOWE INFORMACJE O BOOST_PYTHON

Moduły utworzone w C++ woła się z języka Python, gdy wykorzystywane algorytmy powinny wykonywać się wydajnie. Taki proces nazywa się rozszerzaniem Pythona w C++. Innym powodem rozszerzania jest potrzeba ukrycia kodu przed użytkownikiem.

Biblioteka **boost_python** dostarcza zbioru szablonów, które pozwalają utworzyć interfejs do funkcji i klas C++. Tak obudowane klasy i funkcje będą dostępne dla programów tworzonych w języku Python. Kod utworzony w C++ musi być dostarczany jako biblioteka dzielona, biblioteka **boost_python** dodaje odpowiedni interfejs. Przykład pokazano na Listingu 1. Dostarczamy tam moduł zawierający funkcję **f** oraz klasę **Foo**.

Listing 1. Przykład modułu C++, który można wołać z Pythona

```
int f() { return 43; }//przykładowa funkcja
class Foo { //przykładowa klasa
public:
   Foo() : val_(0) {}
   int get() const { return val_; }
   void set(int v) { val_ = v; }
private:
   int val_;
};
BOOST_PYTHON_MODULE( cppmodule ) //nazwa pakietu
{
   boost::python::def( "f", f );//eksportuje funkcję
   boost::python::class_<Foo>("Foo",
boost::python::init<>())//eksportuje klasę
      .def( "get", &Foo::get )
      .def( "set", &Foo::set )
   ;
} //koniec definicji pakietu
```

Definicję interfejsu rozpoczynamy, podając nazwę pakietu Pythona (argument makrodefinicji **BOOST_PYTHON_MODULE**). Nazwa ta musi być taka sama jak nazwa biblioteki dzielonej, która zawiera udostępniane byty. Wewnątrz bloku umieszczonego za makrodefinicją umieszczamy adaptery eksportujące funkcje i klasy. Funkcję eksportujemy za pomocą szablonu **def**, podając nazwę, która będzie używana w Pythonie oraz wskaźnik na funkcję w C++. Klasę eksportujemy za pomocą szablonu **class_**, parametrem tego szablonu jest typ w C++, natomiast argumentami: nazwa klasy, która będzie używana w Pythonie, oraz argumenty dla metody **__init__** (wykorzystywany jest szablon pomocniczy **boost::python::init<>**). Możemy dodać (wykorzystując literał **.def**) metody, dostarczając napis i wskaźnik do kodu.

Po poprawnej kompilacji i konsolidacji przedstawionego kodu tworzona jest biblioteka dynamiczna o nazwie **cppmodule.so** dla Linux, **cppmodule.pyd** dla Windows. Jeżeli jest ona dostępna dla interpretera Pythona (znajduje się na ścieżce, np. jest w katalogu, z którego interpreter został uruchomiony), możemy wykonać następujące instrukcje:

```
>>> import cppmodule
>>> cppmodule.f()
43
>>> foo = cppmodule.Foo()
>>> foo.get()
0
>>> foo.set(7)
>>> foo.get()
7
```

Jeżeli chcemy, aby pewne akcje uruchamiać przy imporcie pakietu (inicjować zmienne, startować wątki itp.), to możemy dodać kod wewnątrz bloku definiowanego przez makrodefinicję **BOOST_PYTHON_MODULE**. Na Listingu 2 pokazano przykład, podczas importu pakietu będzie uruchamiana funkcja **initModule()**.

Listing 2. Akcje uruchamiane podczas importu pakietu

```
BOOST_PYTHON_MODULE( cppmodule )
{
   initModule(); //f. wołana podczas importu pakietu
   //...
}
```

KONWERSJE OBIEKTÓW

Biblioteka **boost_python** pozwala przekazać do C++ argumenty, które są typami wbudowanymi (**int, double, bool** itp.) oraz napisami (**std::string**). Odpowiednie konwersje są dodawane przez tę bibliotekę i są wołane automatycznie (wykorzystywaliśmy to na Listingu 1, w metodach klasy **Foo**). Ponadto, możemy przekazywać obiekty klas, które zostały wyeksportowane, więc przykładowo możemy pobierać lub zwracać wartości typu **Foo** (z Listingu 2).

Jeżeli chcemy przekazywać jako argument lub zwracać obiekty innych typów, biblioteka dostarcza klasę **boost::python::object**. Reprezentuje ona obiekt zarządzany przez interpreter Pythona i dostarcza m.in. metodę **attr** umożliwiającą odczyt składowych. Możemy więc funkcje i metody przekazywane do Pythona tworzyć, używając tych udogodnień. Dostarczane są także klasy pochodne po **object**, m.in. **dict** – reprezentuje słownik z Pythona, **list** – reprezentuje listę. Dla wspomnianych kontenerów istnieje przeciążony operator indeksowania oraz metoda **len** (zwraca ilość elementów). Przykład użycia pokazano na Listingu 3, gdzie funkcja **getDate** konwertuje zmienną Pythona typu **datetime.date** na obiekt typu **boost::gregorian::date**. Funkcja ta odczytuje poszczególne składowe dostarczonego obiektu (**year**, **month** itd.) i wykorzystuje konwersje dla typów wbudowanych, wołając szablon **extract**.

Aby użytkownik nie musiał jawnie wołać funkcji konwertującej (takiej jak **getDate** z Listingu 3), możemy je dodać (zarejestrować) w strukturach biblioteki **boost_python**. Wtedy funkcje te będą wołane automatycznie przy pobieraniu i przekazywaniu argumentów odpowiedniego typu, oraz przy wołaniu szablonu **extract<nasz_typ>**, podobnie jak dla typów wbudowanych. Własne konwersje przy tworzeniu obiektu klasy na podstawie obiektu Pythona są rejestrowane przez metodę obiektu **boost::python::converter::registry**, zaś przy tworzeniu obiektu Pythona na podstawie obiektu C++ wykorzystujemy szablon **boost::python::to_python_converter**.

Na Listingu 4 pokazano przykład użycia wspomnianego mechanizmu do konwersji pomiędzy **boost::posix_time::ptime** a **datetime**. Funkcje konwertujące są w tym przykładzie metodami (statycznymi) klasy **BoostPosixPTimeConverter**.

Konwersja z obiektu Pythona na obiekt C++ jest realizowana przez metodę **construct**. Obiekt źródłowy jest pierwszym argumentem (argument **object**), zaś bufor, w którym ma być utworzony obiekt docelowy (obiekt C++), jest drugim argumentem. Konwersja odbywa się przy użyciu funkcji **extract**, która woła wcześniej zarejestrowane konwertery (np. dla typów wbudowanych) lub przy pomocy innych funkcji – w przykładzie są to funkcje Python C API do konwersji daty i czasu. Na koniec inicjujemy wskazywany bufor, umieszczając w nim obiekt C++. W przykładzie do tego celu jest wykorzystywany operator **new** dla wskazywanego bufora (ang. *placement new*). Przy rejestracji przedstawionego konwertera dostarczana jest także funkcja, która bada, czy konwersja jest możliwa, tutaj rolę tę pełni metoda **convertible**.

Listing 4. Klasa zawierająca funkcje konwertujące obiekty typu `boost::posix_time::ptime` na Python `datetime` i odwrotnie

```
struct BoostPosixPTimeConverter {
   static void registerConverter() { //rejestruje konwertery
      boost::python::converter::registry::push_back(  &convertible, &construct,
      boost::python::type_id<boost::posix_time::ptime>() );
      boost::python::to_python_converter<boost::posix_time::ptime, BoostPosixPTimeConverter>();
   }
   //konwersja z boost::posix_time::ptime na obiekt Pythona
   static PyObject* convert(const boost::posix_time::ptime& time) {
      PyDateTime_IMPORT;
      int year = time.date().year();
      int month = time.date().month();
      int day = time.date().day();
      int hours = time.time_of_day().hours();
      int minutes = time.time_of_day().minutes();
      int seconds = time.time_of_day().seconds();
      int mi_s = time.time_of_day().total_microseconds() % 1000000;
      return PyDateTime_FromDateAndTime(year, month, day, hours, minutes, seconds, mi_s);
   }
   //bada, czy jest możliwa konwersja z obiektu Pythona na wskazany typ C++
   static void* convertible(PyObject *object) {
      PyDateTime_IMPORT;
      if(!PyDateTime_Check(object))
         return 0L;
      return object;
   }
   //konwersja z obiektu Pythona na obiekt boost::posix_time::ptime
   static void construct(  PyObject *object,
      boost::python::converter::rvalue_from_python_stage1_data *data) {
      PyDateTime_IMPORT;
      PyDateTime_DateTime const* py_dt = reinterpret_cast<PyDateTime_DateTime *>(object);
      boost::gregorian::date d( PyDateTime_GET_YEAR(py_dt),
      PyDateTime_GET_MONTH(py_dt),
      PyDateTime_GET_DAY(py_dt) );
      boost::posix_time::time_duration t =
      boost::posix_time::hours(PyDateTime_DATE_GET_HOUR(py_dt)) +
      boost::posix_time::minutes(PyDateTime_DATE_GET_MINUTE(py_dt)) +
      boost::posix_time::seconds(PyDateTime_DATE_GET_SECOND(py_dt)) +
      boost::posix_time::microseconds(PyDateTime_DATE_GET_MICROSECOND(py_dt));
      void *storage =
      ((boost::python::converter::rvalue_from_python_storage<boost::posix_time::ptime> *)data)->storage.bytes;
      new (storage) boost::posix_time::ptime(d,t);
      data->convertible = storage;
   }
};
```

Listing 5. Funkcja konwertująca obiekty klasy Foo na słownik Pythona (`dict`). Założono, że klasa Foo dostarcza metodę `get` dla typu, który już posiada konwersję (np. jest to typ wbudowany)

```
PyObject* convert(Foo& foo) { //nazwa tej funkcji jest dowolna
   boost::python::dict result; //zwracamy słownik Pythona
   result["value"] = foo.get(); //element słownika, dla klucza "value"
   return boost::python::incref(result.ptr()); //trzeba zwiększyć licznik odniesień do obiektu, bo będzie on zarządzany
przez interpreter Pythona
}Listing 6. Udostępnianie wyliczeń zdefiniowanych w C++
class Connection { //klasa definiuje wyliczenie
   enum ConnectionState { UNKNOWN, NOT_CONNECTED, CONNECTED };
   //inne elementy interfejsu klasy
};
BOOST_PYTHON_MODULE(nazwa)
{
   boost::python::enum_<Connection::ConnectionState>("ConnectionState")
      .value("UNKNOWN", Connection::UNKNOWN)
      .value("NOT_CONNECTED", Connection::NOT_CONNECTED)
      .value("CONNECTED", Connection::CONNECTED)
      .export_values()
      ;
}
```

Listing 7. Wykorzystanie konwerterów dla `boost::tuple` dostarczonych wraz z `boost_python`.

```
boost::tuple<int, int> getMinMax(int a, int b) {
   return boost::minmax(a, b); //funkcja z biblioteki boost algorithm
}
```

Listing 8. Udostępnianie kontenera `std::vector` wykorzystując szablony zdefiniowane w `boost/python/suite/indexing`. Przedstawiona definicja powinna się znaleźć wewnątrz bloku `BOOST_PYTHON_MODULE`

```
class_< std::vector<Foo> >("VectorFoo", no_init ) //zakładamy dostępność konwersji dla Foo
   .def( vector_indexing_suite< std::vector<Foo> > () )
   ;
```

Utworzenie obiektu Pythona z obiektu C++ jest, w przedstawionym przykładzie, realizowane przez metodę **convert**. Na podstawie argumentu, który jest obiektem C++, metoda ta tworzy obiekt typu **PyObject**. Przy implementacji konwerterów musimy zadbać o odpowiednią inicjację licznika odniesień do tworzonych obiektów Pythona, ponieważ będzie on zarządzany przez środowisko interpretera. Pokazano to na Listingu 5, gdzie konwertujemy obiekt klasy **Foo**, w ostatniej linii zwiększamy licznik odniesień do obiektu. Na Listingu 4 tego nie widać, do konwersji wykorzystujemy funkcje Python C API, które same wykonują przedstawione zabiegi. Obiekty tworzone w C++ i zwracane do Pythona przez wartość są zarządzane przez Pythona i prawidłowo usuwane, więc w prostych przypadkach nie musimy się martwić o czas życia obiektów.

Listing 3. Funkcja w C++, która odczytuje składowe obiektu utworzonego w Pythonie

```
//funkcja demonstrująca odczyt składowych (attr) i
//konwersje (extract)
//Do konwersji wygodniej używać rozwiązań pokazanych na
//Listingu 4.
boost::gregorian::date(const object& d) {
   return boost::gregorian::date(extract<int>(d.
attr("year") ),
   extract<int>(d.attr("month") ),
   extract<int>(d.attr("day") ) ) );
}
```

Do tworzenia modułów, które będą wołane w Pythonie, zazwyczaj wystarczą nam omówione poprzednio udogodnienia: rejestracja funkcji, klas oraz własnych konwersji. Dodatkowo biblioteka pozwala uprościć rejestrację typów wyliczeniowych (patrz Listing 6) oraz wspiera automatyczne konwersje dla obiektów typu **tuple** (**boost::tuple**), **vector** i innych. Upraszcza to przekazywanie najczęściej wykorzystywanych struktur danych.

Listing 6. Udostępnianie wyliczeń zdefiniowanych w C++

```
class Connection { //klasa definiuje wyliczenie
   enum ConnectionState { UNKNOWN, NOT_CONNECTED, CONNECTED };
   //inne elementy interfejsu klasy
};
BOOST_PYTHON_MODULE(nazwa)
{

boost::python::enum_<Connection::ConnectionState>("ConnectionState")
       .value("UNKNOWN", Connection::UNKNOWN)
       .value("NOT_CONNECTED", Connection::NOT_CONNECTED)
       .value("CONNECTED", Connection::CONNECTED)
       .export_values()
       ;
}
```

Obiekty **boost::tuple** są automatycznie konwertowane na krotki Pythona (obiekty typu **tuple**), tak jak pokazano na Listingu 7. Warto wykorzystać tę cechę biblioteki **boost_python**, ponieważ pozwala ona przekazywać proste obiekty z kilkoma składowymi (dla większości kompilatorów C++ z obiektami do 10 składowych) bez konieczności tworzenia konwerterów.

Zbiór szablonów **suite/indexing**, który jest częścią biblioteki, pozwala dostarczać jednowymiarowe kontenery biblioteki standardowej (**vector**, **list**, **set** itp.). Przykład (Listing 8) pokazuje definicje dla typów generowanych z szablonu **std::vector**.

OSADZANIE PYTHONA W C++

Kod interpretera jest elastyczny, można go zmieniać i uruchamiać bez konieczności kompilacji, więc zmiany tego kodu są proste i dostępne dla użytkownika oprogramowania. Jeżeli pliki konfiguracyjne aplikacji są skryptami interpretera (np. Pythona), to użytkownik może je zmieniać, a ponadto może umieszczać własne rozszerzenia, np. obliczać wartości parametrów konfiguracyjnych, co często jest wygodne. Używanie interpretera, dostarczonego jako biblioteka, w module napisanym w C++ nazywamy osadzaniem Pythona w C++. Aby poprawnie zbudować taką aplikację, musimy konsolidować (linkować) interpreter dostarczany w formie biblioteki dzielonej, np. **libpython27.so** lub **libpython27.dll**. Przykład wykorzystujący **boost_python** pokazano na Listingu 9, gdzie interpreter otrzymuje napis **res = 2*3*4**, który wykonuje, traktując go jak program. Następnie (w C++) badamy zawartość zmiennych globalnych interpretera, które odczytujemy za pomocą szablonu **extract**.

PODSUMOWANIE

Przedstawiony tekst ma zachęcić czytelnika do tworzenia aplikacji, wykorzystując jednocześnie różne języki programowania. Aplikacje takie mogą posiadać pożądane cechy (np. wydajność i elastyczność) stosunkowo niewielkim kosztem – trzeba opanować sposób komunikacji pomiędzy modułami pisanymi w różnych językach. Istnieje wiele narzędzi upraszczających to zadanie, w artykule pokazano proste przykłady łączenia C++ i Pythona, wykorzystując bibliotekę **boost_python**.

W Sieci

- http://hepunx.rl.ac.uk/BFROOT/dist/releases/24.2.1h/boost/libs/python/doc/PyConDC_2003/bpl.pdf – artykuł wprowadzający do boost_python (Building Hybrid Systems with Boost. Python);
- http://steve.vinoski.net/pdf/IEEE-Multilingual_Programming.pdf – artykuł o korzyściach z programowania w wielu językach.

Więcej w książce

Omówienie współcześnie stosowanych technik, wzorce projektowe, programowanie generyczne, prawidłowe zarządzanie zasobami przy stosowaniu wyjątków, programowanie wielowątkowe, łączenie C++ i Pythona, ilustrowane przykładami stosowanymi w bibliotece standardowej i bibliotekach boost, opisano w książce Robert Nowak, Andrzej Pająk ,,Język C++: mechanizmy, wzorce, biblioteki'', BTC 2010.

Robert Nowak rno@o2.pl

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji wykorzystujących algorytmy sztucznej inteligencji i fuzji danych. Autor biblioteki faif.sourceforge.net. Programuje w C++ od ponad 15 lat.

M. Bagiński, F. Kwiatkowski, M. Szewczyk

Własny bot na GG

w oparciu o platformy BotAPI i Google App Engine

Platforma GG BotAPI pozwala na bardzo łatwe stworzenie własnego bota, np. takiego jak Infobot dostępny pod numerem GG:100. Infobot jest przykładem bota informacyjnego, ale boty mogą pełnić również inne funkcje. W artykule opiszemy, jak stworzyć bota czatowego, który będzie mógł służyć do komunikacji wśród grupy współpracowników.

DZIAŁANIE PLATFORMY BOTAPI

Rdzeniem platformy jest Botmaster, który obsługuje w tej chwili ponad 4 tysiące botów i 250 milionów wiadomości miesięcznie. Botmaster to napisana w C++ aplikacja serwerowa, która zapewnia połączenie numerów GG botów z siecią, odbieranie i wysyłanie wiadomości oraz obrazków, a także zmianę statusów i opisów. Twórca bota musi jedynie napisać skrypt, który będzie określał, co na poszczególne wiadomości odpowie bot, oraz umieścić go na publicznym serwerze HTTP. Gdy użytkownik napisze wiadomość do bota, Botmaster połączy się z adresem URL skryptu bota, podając jako parametry GET numer GG bota, numer GG użytkownika oraz wiadomość od niego jako ciało żądania POST. Na podstawie tych danych skrypt powinien przygotować odpowiedź, którą Botmaster odeśle do użytkownika. Odpowiedzią może być czysty tekst, ale jeśli chcielibyśmy użyć tekstu formatowanego czy też obrazków, należy skorzystać z biblioteki BotAPI. Tak działają tzw. wiadomości PULL. Możliwe jest również wysyłanie wiadomości PUSH, które wysyłane są w dowolnym momencie na żądanie twórcy bota. Należy tu zaznaczyć, że platforma BotAPI uniemożliwia wysyłanie wiadomości do dowolnych numerów GG (czyli zwyczajne spamowanie). Aby bot mógł wysyłać wiadomość do użytkownika, to użytkownik musi wcześniej przynajmniej jeden raz napisać do bota. W każdym momencie można zrezygnować z otrzymywania wiadomości PUSH, pisząc do bota WYPISZ. Analogicznie można ponownie włączyć otrzymywanie takich wiadomości, pisząc ZAPISZ. Dotyczy to każdego bota działającego na platformie BotAPI i jest niezależne od skryptu bota.

BIBLIOTEKA BOTAPI

W tej chwili udostępniamy bibliotekę dla języków PHP oraz Python. Można ją pobrać na stronie http://boty.gg.pl/pobierz/. Biblioteka zapewnia obsługę binarnego protokołu BotAPI, który umożliwia formatowanie tekstu, używanie kolorów oraz dodawanie do treści wiadomości obrazków. Protokół jest dokładnie opisany na stronie http://boty.gg.pl/dokumentacja/#2, nic więc nie stoi na przeszkodzie, by napisać własnego bota również w innym języku niż PHP czy Python. W niniejszym artykule skupimy się jednak na Pythonie, gdyż jest to jeden z języków wspieranych przez platformę Google App Engine. Jest to bardzo wygodne rozwiązanie, jeśli nie mamy własnego serwera HTTP, na którym moglibyśmy umieścić skrypt bota.

GOOGLE APP ENGINE

Google App Engine (GAE) to platforma developerska przeznaczona do programowania i hostowania aplikacji. Pozwala ona tworzyć aplikacje internetowe o wysokim natężeniu ruchu bez konieczności zarządzania infrastrukturą obsługującą taki ruch. Do przechowywania danych służy baza nazywana Datastore. Platforma obsługuje języki Python, Java oraz Go. Każda aplikacja uruchomiona na GAE posiada darmowe limity użycia. Limitowane są między innymi ruch wychodzący i wchodzący (na każdy przypada 1GB dziennie) oraz ilość przechowywanych danych w Datastore (maksymalnie 5GB). Limity można zwiększać, opłacając poszczególne zasoby zgodnie z cennikiem.

KONFIGURACJA ŚRODOWISKA PRACY

Do stworzenia bota będziemy potrzebować numeru GG, na którym go uruchomimy. Jeśli nie dysponujemy wolnym numerem, możemy go założyć, wchodząc na stronę https://login.gadu-gadu.pl/account/register. W artykule numerem naszego bota będzie gg:42990004. Musimy również posiadać konto Google. Jeśli jeszcze takiego nie mamy, możemy założyć je, wchodząc na stronę https://accounts.google.com/SignUp. Nasze środowisko pracy będziemy konfigurować w systemie Windows. Najpierw pobieramy i instalujemy Pythona w wersji 2.5.4 dostępnego na stronie http://www.python.org/ftp/python/2.5.4/python-2.5.4.msi. Pobieramy i instalujemy Google App Engine SDK dla Pythona dla Win: https://developers.google.com/appengine/downloads. Kolejnym krokiem będzie stworzenie nowej aplikacji w Google App Engine. W tym celu logujemy się na stronie https://appengine.google.com i wybieramy przycisk *Create Application*. Jeśli nie mamy jeszcze zweryfikowanego konta Google, będziemy musieli w tym momencie wpisać numer naszej komórki poprzedzony kierunkowym do Polski 48 bez znaku +. Po poprawnej weryfikacji za pomocą otrzymanego SMSa powinien otworzyć się formularz tworzenia aplikacji, w którym wypełniamy tylko dwa pola: *Application Identifier* oraz *Application Title*. W tym artykule *botgg42990004* będzie naszym identyfikatorem aplikacji.

HELLO, BOT!

Nasza pierwsza aplikacja, która posłuży jako skrypt bota, będzie zwracać jedynie krótki tekst, ale pokaże nam, jak wygląda

szkielet aplikacji GAE, którą potem rozwiniemy w pełnoprawnego bota czatowego. Najpierw tworzymy na pulpicie folder o nazwie *botgg42990004*. Następnie umieszczamy w nim plik *main.py* przedstawiony na Listingu 1.

Listing 1. Kod źródłowy pliku main.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app

class Bot(webapp.RequestHandler):
   def post(self):
      self.response.out.write('Hello, Bot!')

application = webapp.WSGIApplication([('/',
Bot)],debug=True)

def main():
   run_wsgi_app(application)

if __name__ == "__main__":
   main()
```

Kolejne linie kodu odpowiadają za zaimportowanie modułów biblioteki GAE. Główna klasa **Bot** posiada tylko jedną metodę **post**, która zostanie wywołana w przypadku żądania typu POST na adres skryptu. Opcjonalnie moglibyśmy oczywiście stworzyć również metodę get dla żądań GET, ale ponieważ Botmaster wysyła żądania typu POST, nie będzie nam ona potrzebna. Osobom, które nie miały do tej pory styczności z Pythonem, przyda się kilka istotnych informacji. Przede wszystkim bardzo ważne są wcięcia w kodzie. To one określają, do jakiego bloku kodu należy dana linia. W Pythonie nie stosuje się do tego, tak jak w wielu innych językach, nawiasów klamrowych. Prawidłowo użyte wcięcia są więc wymuszane przez interpreter Pythona.

Przejdźmy teraz do rejestracji bota na platformie BotAPI GG. Wchodzimy na stronę https://boty.gg.pl/rejestracja/ i wypełniamy wszystkie pola odpowiednimi danymi. Następnie na podany przez nas adres e-mail otrzymamy maila z linkiem aktywującym drugi krok rejestracji. W formularzu podajemy jako adres do wiadomości PULL ten, który zarejestrowaliśmy na Google App Engine - w naszym przypadku będzie to http://botgg42990004.appspot.com/. Musimy również potwierdzić, że to my jesteśmy właścicielami skryptu. W tym celu pobieramy plik HTML, który posłuży do weryfikacji, i umieszczamy go w folderze *botgg42990004*. W naszym przypadku plik ma nazwę *botggafc1248b.html*

Aby uruchomić naszego bota na GAE, potrzebujemy jeszcze pliku konfiguracyjnego o nazwie *app.yaml* przedstawionego na Listingu 2.

Listing 2. Plik konfiguracyjny app.yaml

```
application: botgg42990004
version: 1
runtime: python
api_version: 1

handlers:
- url: /botggafc1248b.html
   static_files: botggafc1248b.html
   upload: botggafc1248b.html

- url: /.*
   script: main.py
```

Przed zapisaniem *app.yaml* w folderze *botgg42990004* podmieniamy nazwę aplikacji na tę, którą zarejestrowaliśmy na GAE oraz nazwę pliku weryfikacyjnego na taką, jaką faktycznie posiada. W przypadku plików typu *yaml* również należy pamiętać o stosowaniu właściwych wcięć. Na koniec uruchamiamy Google App Engine Launcher i wybieramy z menu *File → Add Existing Application*, wskazując ścieżkę do naszego katalogu botgg42990004. Następnie wybieramy przycisk *Deploy* i podajemy dane do konta Google. Po udanym deployu aplikacji możemy wrócić do drugiego kroku rejestracji i wybrać przycisk *Potwierdź teraz*. W ten sposób uruchomiliśmy naszego pierwszego bota GG na platformie BotAPI. Aby sprawdzić, czy działa, wystarczy wysłać wiadomość na numer bota. Odpowie Hello, Bot!

FUNKCJE BOTA

Bot czatowy, który napiszemy, będzie służył do komunikacji między pracownikami jednego działu czy też osób pracujących nad jednym projektem. Bot będzie posiadał zdefiniowaną listę administratorów. Osoby te będą miały uprawnienia do dodawania oraz usuwania numerów GG, które będą otrzymywać wiadomości z bota. Gdy osoba, która została dodana przez administratora, wyśle do bota wiadomość, ta zostanie rozesłana do pozostałych użytkowników bota. Nasz bot będzie również umożliwiał przesyłanie obrazków. W Tabeli 1 znajduje się lista wszystkich funkcji, jakie będą obsługiwane.

BOT CZATOWY

Zastąpimy teraz nasz prosty przykład *Hello, Bot!* skryptem bota, który widać na Listingach od 5 do 20. Skrypt składa się z trzech klas - głównej klasy **Bot**, klasy **User** oraz pomocniczej klasy **DailyScrum**. Ta pierwsza zastąpi klasę z poprzedniego przykładu i będzie stanowić główny kod bota. Druga określa model danych, jakie będziemy przechowywać. Dzięki trzeciej klasie zademonstrujemy, jak skorzystać z cyklicznych powiadomień o Daily Scrumie (przydatne dla zespołów pracujących w tej metodyce). W naszym nowym skrypcie importujemy dodatkowo dwa moduły biblioteki BotAPI - **MessageBuilder** oraz **PushConnection**, które ułatwią nam stworzenie bota. Pierwszy implementuje tworzenie wiadomości zgodnej z protokołem BotAPI - zawierającej formatowanie, kolory oraz obrazki zawarte w wiadomości. Drugi odpowiada za autoryzację i wysyłanie wiadomości typu PUSH oraz zmianę statusu i opisu bota. Bibliotekę należy pobrać ze strony https://boty.gg.pl/pobierz/. Po rozpakowaniu archiwum zip z biblioteką pliki *MessageBuilder.py* oraz *PushConnection.py* znajdziemy w katalogu python.

Główna metoda klasy **Bot** - **post** - widoczna na Listingu 9 - jest wywoływana przy każdym żądaniu typu POST naszego skryptu. Dziedziczy ona po klasie **RequestHandler** z modułu **webapp**. Gdy użytkownik wyśle wiadomość do bota, adres skryptu zostanie wywołany z parametrami *from*, oznaczającym numer GG osoby piszącej, oraz *to*, oznaczającym numer GG bota. Treść wiadomości użytkownika

Funkcja	Opis	Dostępność
/pomoc	wyświetla wszystkie dostępne polecenia	administrator/użytkownik
/dodaj <gg>	dodaje nowego użytkownika czatu	administrator
/usuń <gg>	usuwa użytkownika czatu	administrator
/lista	wyświetla użytkowników czatu	administrator/użytkownik
/nick <nick>	ustawia wyświetlaną nazwę użytkownika	administrator/użytkownik
/opis <opis>	zmienia opis bota	administrator/użytkownik

Tabela 1. Lista funkcji bota czatowego

Znacznik	Efekt
[b]tekst pogrubiony[/b]	**tekst pogrubiony**
[i]tekst pochylony[/i]	*tekst pochylony*
[u]tekst podkreślony[/u]	tekst podkreślony
[color=ff0000]kolorowy tekst[/color]	kolorowy tekst
[br]	przejście do nowej linii

Tabela 2. Znaczniki obsługiwane przez metodę addBBcode

zostanie wysłana w ciele żądania POST jako string kodowany UTF-8. Dane te można łatwo wyciągnąć z pola *request*. Metoda **post** sprawdza, o jaką funkcję chodziło użytkownikowi i wywołuje przypisaną do niej metodę bądź, gdy nie jest to funkcja, wywołuje metodę **broadcast** rozsyłającą wiadomość do pozostałych użytkowników bota. W każdej z tych metod sprawdzane są uprawnienia użytkownika do danej czynności przy pomocy metody **permissionGranted**.

KLASA MESSAGEBUILDER

Do stworzenia wiadomości protokołowej, która zostanie wysłana do odbiorcy, należy użyć klasy **MessageBuilder**. Posiada ona między innymi metodę **addBBcode**, która ułatwia tworzenie wiadomości z formatowaniem. Obsługuje ona formatowanie w popularnym standardzie BBCode. W Tabeli 2 znajdują się obsługiwane przez metodę **addBBcode** znaczniki.

Przykładowe użycie metody **addBBcode** widoczne jest na Listingu 19 w metodzie **cmdShowHelp**. Gdy wiadomość będzie gotowa, należy użyć metody **reply**, która spowoduje zwrócenie przez skrypt binarnej wiadomości protokołowej BotAPI. Domyślnie taka wiadomość trafi jako odpowiedź tylko do osoby, która pisała do bota. Możemy jednak użyć metody **setRecipients** (Listing 8), która pozwala na stworzenie listy odbiorców wiadomości. Należy oczywiście pamiętać o tym, że zostanie ona wysłana tylko do osób, które odezwały się wcześniej przynajmniej raz do bota. Klasa **MessageBuilder** pozwala również na dodanie do wiadomości obrazka - zarówno poprzez podanie ścieżki do pliku, jak i podanie danych binarnych. Służy do tego metoda **addImage** (Listing 13).

KLASA PUSHCONNECTION

Klasa **PushConnection** zawiera zestaw przydatnych metod służących do obsługi bota. Aby z nich skorzystać, musimy wcześniej dokonać autoryzacji. W tym celu umieszczamy w kodzie, co widać na Listingu 6, login i hasło do BotAPI, które otrzymaliśmy w mailu informującym o poprawnej rejestracji bota.
Jedną z metod udostępnionych przez klasę **PushConnection** jest **setStatus**, która pozwala zmienić status i opis bota. Jej przykładowe użycie widać na Listingu 18 w metodzie **cmdSetDescription**. Kolejną metodą, którą wykorzystujemy w bocie, jest **getImage** (Listing 13, metoda **broadcast**) umożliwiająca pobranie obrazków przesłanych w wiadomości od użytkownika do bota. Botmaster pozwala na przesłanie do pięciu obrazków w jednej wiadomości do bota. W połączeniu z metodą **addImage** opisaną wyżej możemy pozwolić użytkownikom bota czatowego na dołączanie do wiadomości obrazków, co widać na Rysunku 1. Klasa **PushConnection** posiada również metodę **push** pozwalającą wysłać na żądanie właściciela bota wiadomość do użytkownika bądź listy użytkowników bota. Metoda ta przyjmuje jako parametr wiadomość zdefiniowaną w obiekcie klasy **MessageBuilder**. Przykład użycia tej metody widać na Listingu 8, który przedstawia klasę **DailyScrum** wysyłającą cykliczne powiadomienia do użytkowników bota czatowego.

CRON W GOOGLE APP ENGINE

Usługa cron w GAE pozwala na skonfigurowanie wykonywania zaplanowanych zadań o określonym czasie lub regularnie. Zadania są automatycznie wywoływane przez crona za pomocą żądań HTTP GET o czasie uprzednio zdefiniowanym w pliku *cron.yaml*. Nasz bot będzie każdego dnia od poniedziałku do piątku o godzinie 10:45 wysyłał do wszystkich osób dodanych do bota wiadomość przypominającą o Daily Scrumie. W tym celu umieszczamy w folderze *botgg42990004* plik *cron.yaml* przedstawiony na Listingu 3.

Listing 3. Plik cron.yaml

```
cron:
- description: Daily Scrum
   url: /scrum
   schedule: every mon,tue,wed,thu,fri 10:45
   timezone: Europe/Warsaw
```

Natomiast w kodzie bota umieszczamy obsługę adresu http://botgg42990004.appspot.com/scrum przedstawioną na Listingu 20 oraz klasę **DailyScrum** widoczną na Listingu 8 zajmującą się tworzeniem i wysłaniem wiadomości.

Rysunek 1. Okno rozmowy na bocie czatowym

GOOGLE APP ENGINE DATASTORE

Datastore to obiektowa, pozbawiona schematu (ang. *schemaless*) baza zapewniająca niezawodne, skalowalne przechowywanie danych. Cechami Datastore są między innymi brak planowanych przestojów, wsparcie dla transakcji atomowych, wysoka dostępność przy odczycie i zapisie oraz silna spójność (ang. *consistency*) przy odczycie. Pracę z Datastore rozpoczynamy od zdefiniowania struktury danych poprzez jej model. Listing 7 przedstawia klasę **User**, która dziedziczy po klasie **db.Model** oraz tworzy jedno pole o nazwie nick. Porównując tradycyjną, relacyjną bazę danych z Datastore oraz biorąc pod uwagę terminologię zaproponowaną przez GAE, to nazwę klasy **User** będziemy nazywać *kind* i jest ona odpowiednikiem nazwy tabeli, nick to *property* i należy o nim myśleć jako o kolumnie tabeli. Wpisy powiązane z danymi, które będziemy umiesz-

czać w bazie, określane są jako *entities*. Tak jak w przypadku tradycyjnych baz danych musimy zdefiniować typ danych przechowywanych w *property*. W naszym przypadku będzie to **StringProperty**, który może przechowywać w unicode do 500 znaków. Każdy wpis powiązany z danymi (*entity*) posiada swój własny unikalny klucz (*key name*), który można zdefiniować przy zapisie danych do Datastore. Skorzystamy z tej możliwości i jako klucz będziemy ustawiać numer GG użytkownika bota, co widać na Listingu 14. Datastore API udostępnia dwa interfejsy służące do tworzenia zapytań o dane. Pierwszy interfejs jest obiektowy, udostępniony przez klasę **Query**, która wystawia zestaw metod pozwalających na pobieranie danych. Na Listingu 16 przedstawiona jest metoda ustawiająca nick, która korzysta tylko z dwóch metod tej klasy: **filter** oraz **get. Filter** pozwala wyciągać dane spełniające określony warunek, korzystając z operatorów. Natomiast **get** wykonuje zapytanie i zwraca pierwszy znaleziony wynik lub None. Drugi interfejs udostępnia klasa **GqlQuery**, która pozwala na tworzenie zapytań w języku GQL. Jego składnia podobna jest do języka SQL, co można zobaczyć na Listingu 11 zawierającym metodę **getUserData**. Obie klasy **Query** i **GqlQuery** są zdefiniowane w module **google.appengine.ext.db**. Datastore pozwala również na zakładanie własnych indeksów na kolumnach. Aby je określić, umieszczamy w folderze *botgg42990004* plik konfiguracyjny o nazwie *index.yaml* przedstawiony na Listingu 4. Linia *direction* określa kierunek sortowania wpisów – w tym przypadku jest to *ascending*, czyli rosnąco.

Listing 4. Plik index.yaml

```
indexes:

- kind: User
   properties:
   - name: __key__
      direction: asc
   - name: nick
      direction: asc
```

Poniżej znajduje się cały kod bota czatowego podzielony na listingi omówione w artykule. Należy pamiętać o poprawnym ustawieniu zmiennych globalnych widocznych na Listingu 6 oraz o wcięciach.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

Listing 5. Importowanie potrzebnych bibliotek

```
import re
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app
from google.appengine.ext import db
from google.appengine.api.datastore import Key
from MessageBuilder import *
from PushConnection import *
```

Listing 6. Definicje zmiennych globalnych

```
BOT_GG = 42990004
PushConnection.BOTAPI_LOGIN = 'wojtek@gg.pl'
PushConnection.BOTAPI_PASSWORD = 'bNpOPRFZ3dCWm4Jj'
ADMIN_LIST = [16303,16305,16492]
PERMISSION_IS_ADMIN = 1
PERMISSION_IS_ADDED = 2
PERMISSION_HAS_NICK = 4
```

Listing 7. Klasa definiująca model danych

```
class User(db.Model):
   nick = db.StringProperty(multiline=False)
```

Listing 8. Klasa wysyłająca cykliczne powiadomienia

```
class DailyScrum(webapp.RequestHandler):
   def get(self):
      M = MessageBuilder()
      P = PushConnection(BOT_GG)
      M.addBBcode('[b]<[color=0000ff]Bot[/color]>[/b] 10:45 - czas na Daily Scrum')
      recipients = Bot.getUsers().keys()
      M.setRecipients(recipients)
      P.push(M)
```

Listing 9. Główna klasa bota

```
class Bot(webapp.RequestHandler):
   def post(self):
      self.msg = self.request.body
      self.uin = self.request.get('from')
      self.M = MessageBuilder()

   functions = {
      'dodaj' : self.cmdAddUser,
```

```
        'usun' : self.cmdDelUser,
        'nick' : self.cmdSetNick,
        'lista' : self.cmdListUsers,
        'opis' : self.cmdSetDescription,
        'pomoc' : self.cmdShowHelp
    }
    r = re.compile(r'^\/(dodaj|usu[ńn]|nick|lista|opis|pomoc)')
    arguments = ''
    function = ''

    if r.search(self.msg):
        function, arguments = (re.sub(r'^/','',self.msg.strip()).split(' ',1) + [''])[:2]
        if function == 'usuń':
            function = 'usun'
        functions[function](arguments.strip())
    else:
        self.broadcast(self.uin, self.msg)
    self.response.out.write(self.M.reply(self))
```

Listing 10. Sprawdzenie uprawnień użytkownika

```
    def permissionGranted(self, mask):
        if mask & PERMISSION_IS_ADMIN and int(self.uin) not in ADMIN_LIST:
            self.M.addBBcode('[b][color=ff0000]Nie masz uprawnień administratora.[/color][/b]')
            return False
    elif (mask & PERMISSION_IS_ADDED or mask & PERMISSION_HAS_NICK) and not self.userExists(self.uin):
        self.M.addBBcode('[b][color=ff0000]Nie masz uprawnień do wysyłania wiadomości.[/color][/b]')
        return False
    elif mask & PERMISSION_HAS_NICK and not self.userHasNick(self.uin):
        self.M.addBBcode('[b][color=0000ff]Skorzystaj najpierw z polecenia[/color] [color=008800]/nick[/color][/b]')
        return False
    else:
        return True
```

Listing 11. Metody określające, czy użytkownik istnieje i czy ma ustawionego nicka

```
    def getUserData(self, uin):
    return db.GqlQuery("SELECT * FROM User WHERE __key__ = KEY('User', :1)", uin)

    def userExists(self, uin):
        q = self.getUserData(uin)
     return q.count() == 1

    def userHasNick(self, uin):
        q = self.getUserData(uin)
        return q.count() == 1 and q.get().nick is not None

    def userNick(self, uin):
        nick = ''
        q = self.getUserData(uin)
        if q.count() == 1 and q.get().nick is not None:
            nick = q.get().nick.encode('utf-8')
        return nick
```

Listing 12. Metoda zwracająca strukturę z użytkownikami

```
    @staticmethod
    def getUsers():
        q = db.GqlQuery("SELECT nick FROM User WHERE nick!=NULL")
        users = {}
        if q.count() > 0:
            for p in q.run():
                users[p.key().name().encode('utf-8')] = p.nick.encode('utf-8')
        return users
```

Listing 13. Metoda rozsyłająca wiadomość do użytkowników bota

```
    def broadcast(self, uin, message):
        if self.permissionGranted(PERMISSION_HAS_NICK):
            sender = self.userNick(uin)
            recipients = self.getUsers().keys()
            self.M.setRecipients(recipients)
            self.M.addBBcode('[b]<[color=0000ff]%s[/color]>[/b] %s' % (sender, message))

            if len(self.request.get('images')) > 0:
                P = PushConnection(BOT_GG)
                imgs = self.request.get('images').split(',')
                for img in imgs:
                    self.M.addImage(P.getImage(img), IMG_RAW)
```

Listing 14. Metoda dodająca nowego użytkownika

```
    def cmdAddUser(self, arguments):
       if self.permissionGranted(PERMISSION_IS_ADMIN):
          r = re.compile(r'^([0-9]+)$').search(arguments)
          if r is None:
             self.M.addBBcode('[b][color=0000ff]Użycie:[/color][/b][br]')
             self.M.addBBcode('[b][color=008800]/dodaj[/color] numer gg[/b]')
          else:
             uin = r.groups()[0]
             if self.userExists(uin):
                self.M.addBBcode('[b][color=0000ff]Użytkownik został już wcześniej dodany.[/color][/b]')
             else:
                newUser = User(key_name=uin)
                newUser.put()
                self.M.addBBcode('[b][color=0000ff]Użytkownik został dodany.[/color][/b]
```

Listing 15. Metoda usuwająca istniejącego użytkownika

```
    def cmdDelUser(self, arguments):
       if self.permissionGranted(PERMISSION_IS_ADMIN):
          r = re.compile(r'^([0-9]+)$').search(arguments)
          if r is None:
             self.M.addBBcode('[b][color=0000ff]Użycie:[/color][/b][br]')
             self.M.addBBcode('[b][color=008800]/usuń[/color] numer gg[/b]')
          else:
             uin = r.groups()[0]
             if not self.userExists(uin):
                self.M.addBBcode('[b][color=0000ff]Użytkownik nie został wcześniej dodany.[/color][/b]')
                self.M.addBBcode('[br][b][color=0000ff]Napisz:[/color] [color=008800]/lista[/color][/b]')
             else:
                key = db.Key.from_path('User',uin)
                db.delete(key)
                self.M.addBBcode('[b][color=0000ff]Użytkownik został usunięty.[/color][/b]')
```

Listing 16. Metoda ustawiająca nick

```
    def cmdSetNick(self, arguments):
       if self.permissionGranted(PERMISSION_IS_ADDED):
          r = re.compile(r'^([a-zA-Z0-9ąćęłńóśźżĄĆĘŁŃÓŚŹŻ ]+)$').search(arguments)
          if r is None:
             self.M.addBBcode('[b][color=0000ff]Użycie:[/color][/b][br]')
             self.M.addBBcode('[b][color=008800]/nick[/color] nazwa użytkownika[/b]')
          else:
             nick = r.groups()[0]
             q = db.Query(User)
             q = q.filter('nick =', nick.decode('utf-8'))
             result = q.get()
             if result is not None:
                self.M.addBBcode('[b][color=0000ff]Ten nick jest już zajęty.[/color][/b]')
             else:
                newNick = User(key_name=str(self.uin), nick=nick.decode('utf-8'))
                newNick.put()
                self.M.addBBcode('[b][color=0000ff]Twój nick został ustawiony na:[/color] %s[/b]' % nick)
```

Listing 17. Metoda wyświetlająca listę użytkowników

```
    def cmdListUsers(self, arguments):
       if self.permissionGranted(PERMISSION_HAS_NICK):
          users = self.getUsers()
          self.M.addBBcode('[b][color=0000ff]Lista użytkowników bota:[/color][/b]')
          for uin, nick in users.iteritems():
             self.M.addBBcode("[br][b][color=008800]%s[/color] - [u]gg:%s[/u][/b]" % (nick,uin))
```

Listing 18. Metoda ustawiająca opis bota

```
    def cmdSetDescription(self, arguments):
       if self.permissionGranted(PERMISSION_HAS_NICK):
          if len(arguments) > 0:
             P = PushConnection(BOT_GG)
             P.setStatus(arguments, STATUS_BACK)
             self.M.addBBcode('[b][color=0000ff]Opis został ustawiony na:[/color] %s[/b]' % arguments)
          else:
             self.M.addBBcode('[b][color=0000ff]Użycie:[/color][/b][br]')
             self.M.addBBcode('[b][color=008800]/opis[/color] tekst[/b]')
```

Listing 19. Metoda wyświetlająca listę dostępnych poleceń

```
    def cmdShowHelp(self, arguments):
```

```
        if self.permissionGranted(PERMISSION_IS_ADDED):
           self.M.addBBcode('[b][color=0000ff]Dostępne polecenia:[/color][/b]')
           if int(self.uin) in ADMIN_LIST:
              self.M.addBBcode('[br][b][color=008000]/dodaj[/color][/b] - dodaje nowego użytkownika')
              self.M.addBBcode('[br][b][color=008000]/usuń[/color][/b] - usuwa istniejącego użytkownika')
           self.M.addBBcode('[br][b][color=008800]/nick[/color][/b] - ustawia nick')
           self.M.addBBcode('[br][b][color=008000]/lista[/color][/b] - wyświetla listę użytkowników')
           self.M.addBBcode('[br][b][color=008000]/opis[/color][/b] - ustawia opis bota')
           self.M.addBBcode('[br][b][color=008000]/pomoc[/color][/b] - wyświetla pomoc')
```

Listing 20. Lista aplikacji Web Server Gateway Interface

```
application = webapp.WSGIApplication([('/', Bot),('/scrum', DailyScrum)],debug=True)

def main():
   run_wsgi_app(application)

if __name__ == "__main__":
   main()
```

Wszystkie potrzebne do stworzenia bota czatowego pliki z powyższego przykładu można pobrać ze strony: http://boty.gg.pl/bot_programistamag.zip. Zainteresowanych stworzeniem własnego projektu w oparciu o platformy BotAPI i Google App Engine zachęcamy do zapoznania się z poniższymi linkami.

W sieci

- http://boty.gg.pl/ – dokumentacja BotAPI, biblioteka oraz przykłady
- https://developers.google.com/appengine/ – strona Google App Engine
- http://forum.gg.pl/forumdisplay.php?54-Boty – oficjalne forum poświęcone platformie BotAPI GG
- http://forum.gg.pl/showthread.php?621 – szczegółowy tutorial opisujący, jak założyć bota na platformie BotAPI GG, korzystając z Google App Engine

Marcin Bagiński xmoki9@gmail.com

Pracownik GG Network. Na co dzień programuje boty. Tworzy oprogramowanie w PHP i C++ od 10 lat.

Filip Kwiatkowski filip.kwiatkowski@gmail.com

Absolwent Informatyki na Uniwersytecie Marii Curie-Skłodowskiej w Lublinie. Współtwórca projektu Infobot.pl. Obecnie pracuje w GG Network jako programista C++ i poza Infobotem zajmuje się również rozwojem Botmastera. W wolnym czasie fotografuje, jeździ na rowerze oraz buduje z Lego Technic i Mindstorms.

Maciej Szewczyk rodion.infobot@gmail.com

Prawnik z wykształcenia, project manager z wyboru. Współtwórca projektu Infobot.pl. W GG Network odpowiedzialny jest za rozwój platformy BotAPI oraz niestandardowe kampanie reklamowe i promocyjne w oparciu o boty. Prywatnie mąż i tata Adasia.

Łukasz Mazur

Koncepcja i narzędzia Continuous Integration

Zastosowanie ciągłej integracji (continuous integration) zmniejsza ryzyko w projekcie. Projekt jest pod ciągłą kontrolą i monitoringiem. Minimalizuje również ilość czynności, które należy wykonać w ramach procesu, dzięki automatyzacji budowania oprogramowania oraz automatyzacji czynności wdrożeniowych.

WPROWADZENIE

Ciągła integracja (*continuous integration*,CI) jest praktyką tworzenia oprogramowania, w którym członkowie zespołu integrują często swoją pracę, zazwyczaj każda osoba integruje co najmniej raz dziennie do kilku integracji dziennie. Każda integracja jest weryfikowana przez zautomatyzowany build (w tym testy) w celu jak najszybszego wykrycia błędów integracji. W wielu zespołach okazuje się, iż takie podejście prowadzi do znaczącej redukcji problemów integracyjnych i pozwala zespołowi opracować szybciej spójne oprogramowanie.

W projekcie zawsze mamy do czynienia z oprogramowaniem gotowym do wdrożenia dzięki zastosowaniu CI, zespół jest zobligowany do tworzenia projektu, który umożliwia jego wdrożenie w dowolnym momencie.

Osoby odpowiedzialne za projekt mogą samodzielnie zbadać postęp prac w projekcie – wymiernym wskaźnikiem, który określa postęp projektu, jest stan oprogramowania, które można wdrożyć.

Większa pewność w odniesieniu do produktu CI wymusza, aby wytwarzane oprogramowanie weryfikować pod kątem zaimplementowanych funkcji.

Obecnie pracując w firmie rozwijającej oprogramowanie, wykorzystywane są serwery CI. Ustawiono na nich wiele różnych projektów. Są to zarówno projekty serwerów, jak i klientów (na różnych platformach Java, iOS/Objective-C etc.). Cały system działa sprawnie, bardzo rzadko potrzeba coś w nim poprawić. Ogólnie platforma jest przyjemna do zarządzania i daje bardzo dużo korzyści. Ogranicza również niepewność i ryzyko integracji. Korzystając z tego doświadczenia, chciałem się podzielić pewnymi wytycznymi i wskazówkami na opisywany temat, bo uważam, iż warto. W artykule zostanie omówione zagadnienie ciągłej integracji, jak również narzędzia wykorzystywane do realizacji tego procesu. Tematyka będzie rozbudowana poprzez zaprezentowanie przykładu konfiguracji środowiska. Jako podsumowanie wymienione zostaną korzyści z zastosowania omawianego podejścia.

W praktyce, każdy członek zespołu programistycznego powinien przynajmniej raz dziennie umieścić wykonaną przez siebie pracę w repozytorium. Niezbędnym elementem jest także zapewnienie poprawności kompilacji kodu po wykonaniu integracji. Każda integracja jest weryfikowana przez automatyczne budowanie (w tym testy oraz przegląd i ocenę kodu) w celu wykrycia błędów integracji, tak szybko jak to tylko możliwe. W wielu zespołach podejście to prowadzi do znacznego zmniejszenia problemów z integracją i pozwala na tworzenie spójnego oprogramowania w szybszym tempie, eliminując jednocześnie wiele błędów. Prawidłowo przeprowadzana ciągła integracja powinna prowadzić do:

- zmniejszenia kosztów i ilości pracy niezbędnej do łączenia prac wykonanych przez różne osoby,
- wcześniejszego wykrywania błędów.

Termin ciągła integracja w odniesieniu do systemów informatycznych wszedł do powszechnego użycia po opublikowaniu przez Martina Fowlera artykułu „Continuous Integration” (Martin Fowler: http://martinfowler.com/articles/continuousIntegration.html).

Proces ciągłej integracji w wielu przypadkach nie zastępuje testowania systemu informatycznego, a jest jedynie jego dopełnieniem pozwalającym znacznie zwiększyć efektywność testów poprzez ich częste, w pełni automatyczne wykonywanie. W swoim artykule Martin Fowler przedstawia etap integracji dużego projektu informatycznego jako długi i niepewny proces. Proces tym dłuższy i tym bardziej niepewny, im dalej w czasie odwlekana jest integracja i testowanie projektu. Na bazie własnych doświadczeń oraz doświadczeń swoich współpracowników z firmy ThoughtWorks (http://www.thoughtworks.com) zdefiniował on szereg praktyk, którymi powinny się kierować projekty informatyczne w celu uproszczenia procesu integracji systemu i możliwości jego wykonania na żądanie w dowolnej chwili. Poniżej najważniejsze z nich zostaną krótko scharakteryzowane.

Pojedyncze repozytorium kodu

Według Fowlera, narzędzia kontroli wersji są integralną częścią każdego, nawet najmniejszego projektu. Mnogość plików i częstotliwość zmian, jakie w nich zachodzą, wymusza stosowanie repozytorium kodu źródłowego w celu monitorowania modyfikacji, jakim poddawana jest aplikacja, i w razie konieczności przywracania poprawnie działających wersji. Oprócz utrzymywania historii poszczególnych plików repozytorium ma za zadanie ułatwić nowym programistom rozpoczęcie pracy z projektem. Jednak aby to osiągnąć, systemy kontroli wersji powinny zawierać nie tylko kod źródłowy aplikacji, ale również wszystkie skrypty kompilacyjne, uruchomieniowe, a także biblioteki używane w projekcie.

Automatyzacja budowy

Każdy system powinien być wyposażony w skrypt budujący poszczególne komponenty i składający je w całość (w postaci bibliotek lub plików wykonywalnych). Każdorazowe, ręczne wykonywanie procesu budowy poszczególnych elementów programu wprowadza niepotrzebny narzut czasowy oraz umożliwia popełnienie błędów. Tworzone programy zawierają zestawy testów dla poszczególnych komponentów. W związku z tym integralną częścią procesu budowy aplikacji powinno być uruchomienie tych testów w celu sprawdzenia, czy aplikacja nie zawiera

błędów. Każdy projekt powinien posiadać dedykowany serwer integracyjny uruchamiający skrypt budujący i testujący aplikację po każdorazowym wykryciu zmiany w repozytorium kodu źródłowego. Dzięki temu możliwe jest wczesne wykrywanie wprowadzonych błędów i naprawianie ich niewielkim kosztem.

Częsta synchronizacja

Warunkiem pomyślnego wprowadzenia poprzedniej praktyki jest zapewnienie odpowiednio częstej synchronizacji kodów tworzonych przez poszczególnych programistów z centralnym systemem kontroli wersji. Sugerowanym schematem pracy jest synchronizacja posiadanej wersji kodów przed rozpoczęciem pracy nad kolejnym zadaniem oraz po jego ukończeniu. Jeżeli synchronizacja kodu odbywa się odpowiednio często, odnalezienie źródła problemu jest niemal natychmiastowe, ponieważ liczba zmienianych plików jest niewielka. W przypadku gdy synchronizacja odbywa się jedynie na końcu projektu, wprowadzone błędy są często nie do odnalezienia i zostają przeniesione do systemów produkcyjnych, gdzie ich naprawa często jest nie do wykonania.

Wszyscy widzą co się dzieje

Wielokrotnie w swoim artykule autor podkreśla, iż głównym zadaniem ciągłej integracji jest zapewnienie komunikacji pomiędzy wszystkimi osobami zaangażowanymi w projekt. Co za tym idzie, każdy powinien być w stanie w jak najprostszy sposób określić status, w jakim aktualnie znajduje się system, oraz zmiany, jakie do niego wprowadzono. W związku z tym proces automatycznego budowania projektu musi być wyposażony w narzędzie pozwalające na raportowanie aktualnego stanu, na przykład interfejs webowy pokazujący wyniki przeprowadzanych testów. Powyższe wskazówki stworzyły swoistą specyfikację dla pojawiających się w użyciu systemów ciągłej integracji. Przykładami takich systemów są takie narzędzia jak CruiseControl czy Hudson/Jenkins.

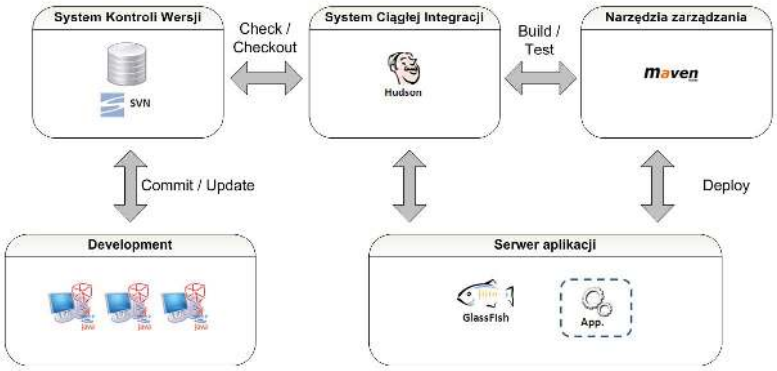
SERWERY CIĄGŁEJ INTEGRACJI

Hudson

Wskazówki udzielone przez Martina Fowlera w jego artykule „Continuous Integration" stały się bazą dla wielu projektów mających na celu stworzenie serwera ciągłej integracji. Narzędzie Hudson zapoczątkowane zostało przez Kohsuke Kawaguchi, po czym projekt szybko znalazł szerokie grono sympatyków i rozwijany był na zasadzie opensource przez developerów z całego świata. Interfejs użytkownika aplikacji pozwala na zdefiniowanie zadań, które realizują cykl budowy aplikacji.

Rysunek 1. Logo Hudsona

Hudson podczas definiowania zadań potrafi skorzystać z istniejących skryptów dla narzędzi Maven oraz Ant, a także pozwala na wywołanie dowolnego skryptu powłoki realizującego



Rysunek 2. Przebieg procesu ciągłej integracji

budowę aplikacji. Podczas budowy systemu serwer umożliwia uruchomienie zdefiniowanych testów, a następnie publikuje ich wyniki we wskazanej lokalizacji. Po ukończeniu wszystkich zdefiniowanych operacji serwer może poinformować o wynikach, wysyłając wiadomość na zdefiniowane adresy e-mail lub przez różne narzędzia np. Jabber/XMPP (Google Talk), IRC, Twitter, Skype itd.

Istotną zaletą Hudsona jest rozszerzalność jego funkcjonalności dzięki dostępnym dodatkom (*plugins*). Dzięki nim serwer jest w stanie na przykład zainicjalizować środowisko uruchomieniowe aplikacji (systemu) w celach testowych lub też wykonać instalację zbudowanego artefaktu na serwerach aplikacyjnych. Pluginy pozwalają również na pobieranie kodu z repozytoriów innych niż CVS i Subversion. Wykorzystanie Hudsona pozwala zastąpić pojedyncze, ręcznie wykonywane fazy integracji i testowania systemu, na pełną automatyzację całego procesu. Fazy te wykonywane są w określonych interwałach czasowych oraz są powtarzalne, co ma niebagatelne znaczenie.

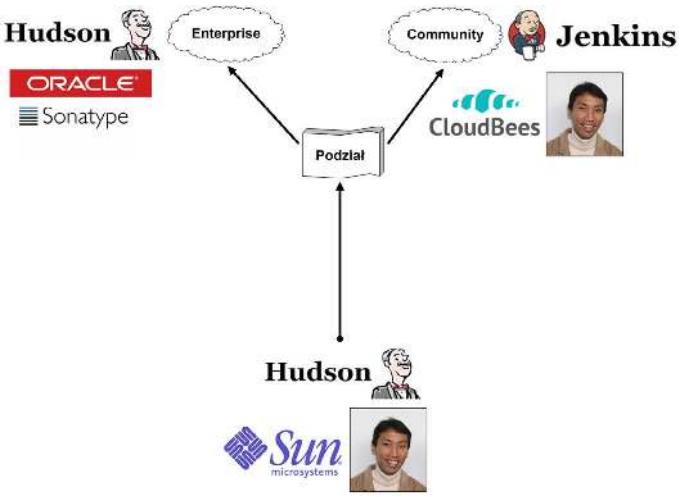
Dodatkowym atutem jest możliwość wyzwolenia cyklu budowy aplikacji przez systemy zewnętrzne, na przykład w odpowiedzi na załadowanie zmian do repozytorium Subversion. Serwer wraz z kodem źródłowym i dokumentacją jest dostępny do pobrania pod adresem: http://hudson-ci.org/.

Kim jest Jenkins

Jenkins znany był wcześniej właśnie jako Hudson. Jest oprogramowaniem opensource, serwerem ciągłej integracji (CI). Narzędzie zbytnio nie różni się od poprzednika, więc również napisane jest w Javie. Projekt zmienił nazwę po sporze z Oracle, który rości sobie prawo do znaku towarowego i nazwy Hudson.

Jenkins zapewnia „z pudełka" ciągłą integrację usług rozwoju oprogramowania, głównie dla platformy Java. Oczywiście wykorzystując rozszerzenia, skrypty i różne konfiguracje, możemy budować różne inne projekty niekoniecznie bazujące na Java. Jest to system, który może być uruchamiany w serwerach aplikacji takich jak Tomcat, WebSphere oraz JBoss. Integruje się z systemami kontroli wersji, w tym CVS, Subversion, Git i ClearCase. Może wykonywać skrypty związane z Apache Ant i Apache Maven, jak również dowolne skrypty powłoki i polecenia systemu Windows. Głównym twórcą Jenkins jest Kohsuke Kawaguchi. Opublikowane na Licencji MIT, Jenkins jest wolnym oprogramowaniem.

Buildy można uruchomić za pomocą różnych środków, w tym mogą być wyzwolone przez komitowanie kodu do systemu kontroli wersji, zaplanowane poprzez mechanizm typu Cron, wyzwalane po tym, jak inne zadania zakończyły swoje działanie oraz wywołując określony URL (co jest bardzo interesujące).



Rysunek 3. Podział projektu na dwa nurty

Istnieje możliwość integracji np. z Gerritem poprzez specjalne pluginy. Projekt zapoczątkowany został około 2007 r. jako popularna alternatywa dla CruiseControl i innych opensource'owych serwerów o podobnym charakterze. Na konferencji Java One w maju 2008 r. program został zwycięzcą *„Duke's Choice Award"* w kategorii Solutions Developer.

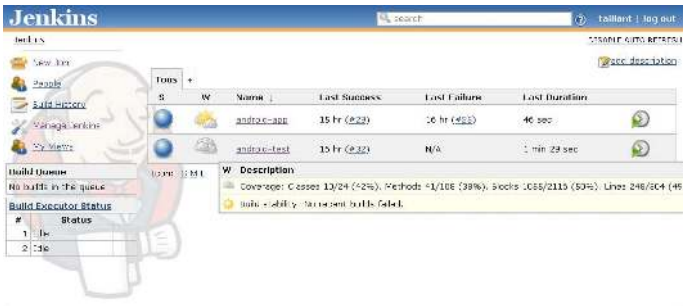
Przydatne dodatki dla serwera

W bieżącym podrozdziale postaram się zaprezentować kilka pomocnych wtyczek, które generalnie są bardzo użyteczne i wykorzystanie ich może przynieść wiele korzyści. Omawiane tutaj dodatki uczynią możliwości konfigurowania Jenkinsa i działanie budowania znacznie bardziej rozbudowanym i wydajnym. Przejdźmy zatem do istoty zagadnienia.

Jenkins jest rozszerzalny za pomocą pluginów. Obecnie jest ich do wyboru około 320. Generalnie instalacja wtyczek (które można zarekomendować) pomogła mi lepiej skonfugurować automatyzację buildów oraz stworzyć bardziej solidne i lepiej zarządzane środowisko serwera ciągłej integracji:

- **Disk Usage** (*Jenkins Disk Usage plugin*) **–** obecnie przestrzeń dyskowa jest tania, ale nadal dobrze posiadać informację dotyczącą obszaru zajmowanego miejsca na dysku, przez konkretne projekty. Można wtedy odpowiednio zareagować w razie potrzeby.
- **JobConfigHistory** (*JobConfigHistory plugin*) **–** zapisuje kopie wszystkich zadań (*jobs*) i konfigurację systemu. Plugin daje możliwość tworzenia kopii zapasowych i audyt zmian wprowadzonych do zadań. Nie zastępuje VCS dla twojej konfiguracji, ale zapewnia szybki start.
- **Email-Ext** (*Email-Ext plugin*) **–** ten plugin rozszerza wbudowane funkcje powiadamiania e-mailem, dając Ci więcej kontroli nad tym procesem. Zapewnia dostosowanie w trzech obszarach:
  - **`wyzwalacze`** – mamy tutaj możliwość definiowania warunków, które po spełnieniu będą uruchamiać powiadomienie e-mailem,
  - **`treść`** – możliwość określenia zawartości (tematu i treści) dla zdefiniowanych wiadomości e-mail,
  - **`odbiorcy`** – możliwość określenia odbiorców, którzy powinni otrzymać zdefiniowaną wiadomość e-mail.

  Email-Ext jest wysoce konfigurowalny. Mając to na uwadze, możemy go doskonale wykorzystać do sprawniejszego przekazywania informacji. W końcu celem narzędzia jest to, aby



Rysunek 4. Główny dashboard Jenkinsa

każdy, kto bierze udział w projekcie, był na bieżąco informowany o statusie buildu.

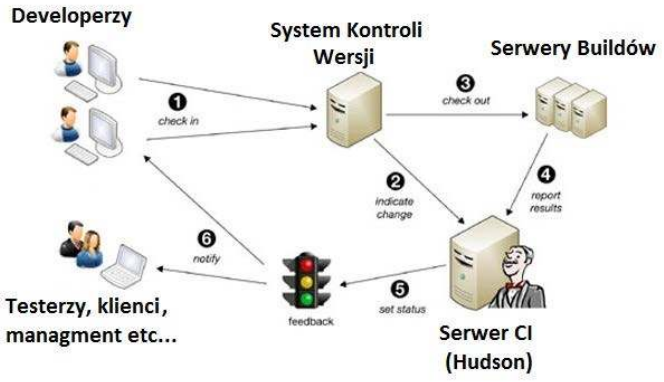
- **Jira Jenkins** (*Jira Jenkins plugin*) - ten plugin integruje Atlassian JIRA do Jenkinsa. Pozwala to programiście odwoływać się do Jira issue za pomocą odpowiednio przygotowanego komentarza (jak np issue DSA-324), a Jenkins automatycznie zaktualizuje właściwe issue w JIRA, dodając odpowiednie komentarze programistów oraz odniesienia (relacje) do zmian w kodzie, które zostały zrealizowane w ramach tego issue. To bardzo proste, ale potężne narzędzie, dzięki któremu proces testowania i poprawiania błędów uczynimy znacznie sprawniejszym.
- **Integracja z Sonar** (*Jenkins Sonar plugin*) – to narzędzie do raportowania jakości kodu (*code quality*). Jest to bardzo przydatna aplikacja, dzięki której można wygenerować raport dotyczący jakości kodu. Korzysta z Sonar'a, narzędzia opensource do monitorowania jakości kodu platformy, opartego na wielu znanych narzędziach analitycznych, takich jak: Checkstyle i PMD, Findbugs oraz Cobertura.
- Dla zaawansowanego użytkowania, który planuje budowę serii instalacji za pomocą Jenkinsa, można polecić dodatek **Build Pipeline plugin,** który oferuje nam bardzo przydatne wizualizacje procesów. W ramach rozszerzenia pojęcia CI, można utworzyć łańcuch zadań do wykonania. Każde z zadań ma przypisany Twój build do pewnego kroku opisanego w scenariuszach QA. Kroki te mogą być kombinacją kroków manualnych i zautomatyzowanych. Gdy build pomyślnie przebrnie przez wszystko, co zostało zdefiniowane przez nas, może być automatycznie wdrożony na produkcję. Plugin Build Pipeline został stworzony, aby lepiej wspierać tego typu procesy. Daje to możliwość tworzenia łańcucha zadań w oparciu o ich zależności upstream oraz downstream.

Użycie powyżej opisanych wtyczek może być bardzo korzystne dla każdego projektu, więc należy przeznaczyć trochę czasu, wgłębić się trochę w literaturę i próbować rozszerzyć środowisko o te i wiele innych dodatków, a z pewnością będziemy mieli z tego korzyści.

Hudson - przykład zastosowania

Jak wspomniałem, Hudson jest serwerem, który pozwala na wdrożenie podejścia ciągłej integracji. Prześledźmy więc typowy jego przebieg (Rysunek 5).

Najpierw developerzy dodają nowy kod do systemu kontroli wersji (1). Następnie Hudson sprawdza repozytorium kodu, regularnie poszukując zmiany, np. każda minuta (2). Jeśli zmiany są zauważone, nowy kod jest pobierany i budowana jest aplikacja (3). Po ukończonej budowie wyniki są przesyłane do Hudsona (4), który ustawia nowy (aktualny) status projektu (5).



Rysunek 5. Typowy cykl ciągłej integracji z wykorzystaniem Hudsona

Wyróżniamy trzy statusy: stabilny, niestabilny oraz nieudany. Finalnie powiadomienia (email, komunikatory) są wysyłane do developerów i innych zainteresowanych podmiotów (biorących udział w projekcie) (6), którzy będą (w zależności od wyniku) naprawiać wszystkie problemy lub kontynuować wdrażanie nowych funkcji w zależności od statusu projektu. Zazwyczaj ten kompletny cykl nie przekracza 15 minut.

Instalacja

Należy pobierać najnowszą wersję paczki z adresu: http://hudson-ci.org/latest/hudson.war, po czym uruchomić Hudsona z terminala za pomocą następującego polecenia:

```
java -jar hudson-x.x.x.war
```

Następnie można otworzyć aplikację w przeglądarce **`http://localhost:8080`**. Po tej czynności powinno się ujrzeć dashboard Hudsona. Jeżeli port **`8080`** jest już używany, potrzeba wybrać inny port (np: **`8888`**) przez uruchomienie Hudson poleceniem:

```
java -jar hudson.war –httpPort=8888
```

Hudson może być również uruchamiany w serwerach aplikacji takich jak Tomcat, WebSphere lub JBoss. Projekt Hudson publikuje nowe wersje mniej więcej co tydzień. Po instalacji użytkownik jest informowany o zmianach dostępnych w panelu administracji: „Administration page (Hudson / Manage Hudson)". Możliwość instalowania dodatkowych rozszerzeń jest realizowana za pomocą wbudowanego menedżera wtyczek: „Plugin Manager (Hudson / Manage Hudson / Manage Plugins)" (Rysunek 6b).

Rysunek 6. Startowy dashboard Hudsona z ustawionym zadaniem

Architektura rozproszona

Tworzenie i testowanie oprogramowania jest raczej obciążającym dla zasobów procesem. Ciągła integracja oprogramowania wymaga jeszcze większej mocy obliczeniowej. Hudson może rozproszyć swoje buildy do gridów build serwerów z węzłami typu *slave*. Przyspiesza to budowanie oraz pozwala na testowanie oprogramowania na wielu węzłach podrzędnych z różnymi systemami operacyjnymi, bazami danych, serwerami aplikacji, przeglądarkami internetowymi, pakietami serwisowymi itp. Węzły podrzędne mogą być uruchamiane na „prawdziwym sprzęcie" w twoim pokoju, w centrum danych lub na zwirtualizowanych systemach, np. na komputerze VMWare lub w chmurze Amazon EC2. Hudson zadba o monitorowanie węzłów, rozdział zadań oraz zbierze wyniki pracy, tak szybko jak tylko kompilacje zostaną zakończone w węzłach podrzędnych.

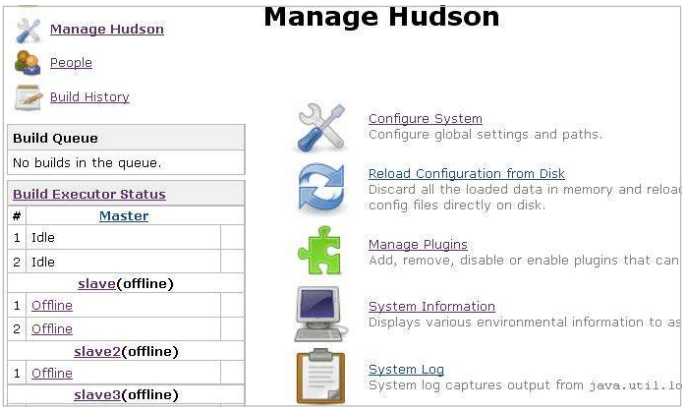
Możliwość rozszerzenia

Łańcuchy buildów oprogramowania zawsze różnią się w poszczególnych zespołach, dobry serwer CI musi wykazać się „infrastrukturą kameleona". Hudson jest rozszerzalny przez około kilkaset dodatków, oferując wsparcie dla alternatywnych narzędzi budowy (np. msbuild, gradle, rake), systemy kontroli wersji (np. CVS, Git, Perforce, Mercurial), kanały notyfikacji (np. rich email, jabber, twitter, eXtreme feedback devices), wirtualizacji i cloud computingu (np. VMWare, Amazon EC2), katalogami użytkowników (np. LDAP, ActiveDirectory), bugtrackery (np. JIRA, Bugzilla, Mantis) i wiele więcej. Przy obecnym tempie rozwoju społeczność Hudsona uwalnia ok. 12 nowych wtyczek i 35 aktualizacji tygodniowo. Kod źródłowy tych wtyczek jest dostępny, aby służyć jako punkt startowy dla własnych kastomizacji. W rzeczywistości, wiele hudsonowych „heavy users" przychodzi na zapakowany, gotowy do użycia produkt, ale otrzymuje wysoce konfigurowalną platformę budowy oprogramowania.

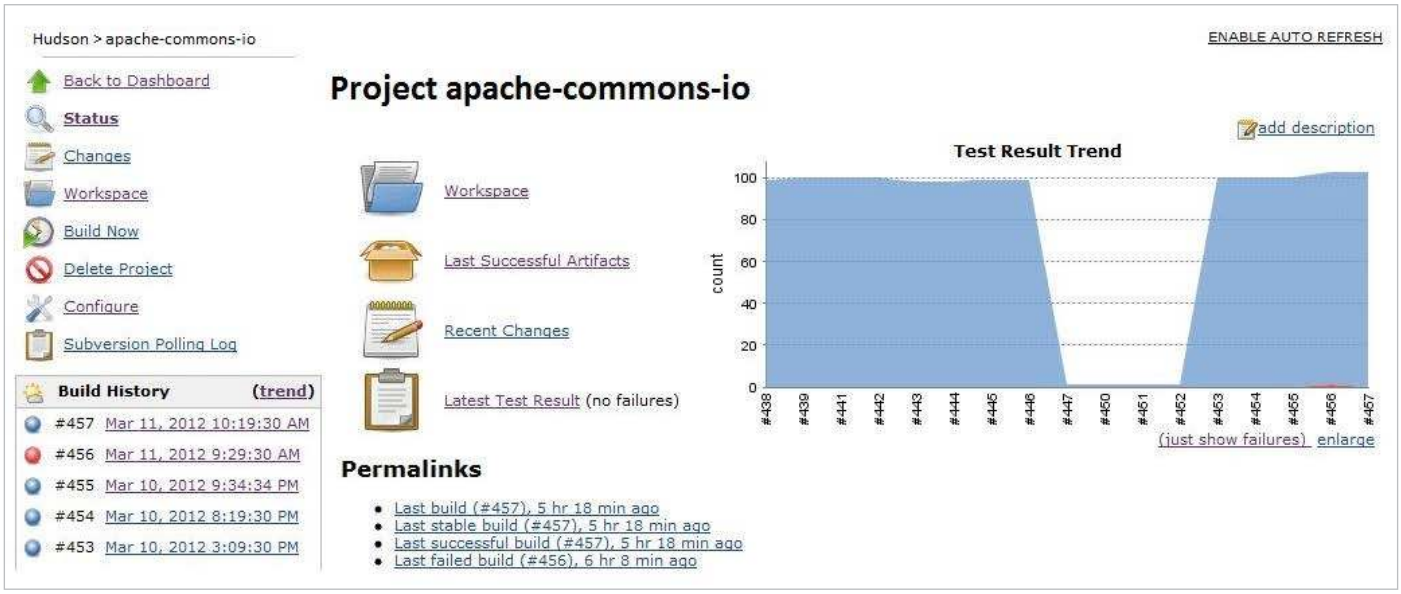
„Test Drive" Projekt

Najlepiej zrozumieć działanie Hudsona, widząc go podczas działania na żywo. Tak więc, na prostą jazdę próbną można zbudować dowolny projekt opensource, który posiada kod źródłowy publicznie dostępny. W omawianym przykładzie będziemy używać Apache Commons IO:

Przechodzimy do Hudson – Manage Hudson – Configuration: W sekcji „JDK" potrzeba dodać nowy pakiet Java Development Kit. W dziale „Maven" należy dodać nową instalację Maven. Po czym klikamy na „Save" na dole strony. Następnie przechodzimy do Hudson - New Job i wpisujemy „**`apache-commons-io`**"



Rysunek 6a. Panel zarządzania

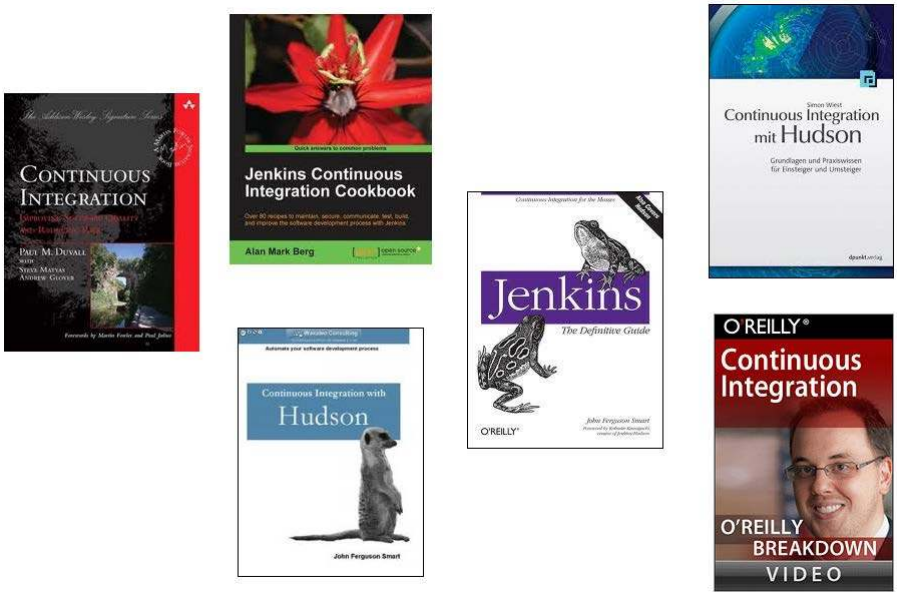


Rysunek 7. Dashboard trendu dla wykonanych testów

jako nazwę zadania, wskazując na opcję „**`Build a maven2 project`**”. Kliknamy „OK”, aby kontynuować i zatwierdzić konfigurację zadania. Na stronie konfiguracji zadania, w sekcji „Source Code Management”, wybieramy „Subversion” i wpisujemy lokalizację kodu źródłowego:
http://svn.apache.org/repos/asf/commons/proper/io/trunk

W polu „Repository URL” klikniamy na „Save” na dole strony. Po czym po pojawieniu się panelu dla właśnie skonfigurowanego projektu klikamy na „Build Now”. Nowy build rozpocznie się, pobierając kod źródłowy, a następnie wykonywana będzie kompilacja i testowanie go. Można śledzić postępy na wyjściu konsoli (*console output*). Aby to uzyskać, potrzeba kliknąć na pasku postępu na lewej stronie, po czym można otworzyć widok konsoli. Gdy kompilacja zostanie zakończona, można wrócić do strony głównej Hudsona, klikając na „Hudson” w górnym lewym rogu. Jeśli build zakończył się sukcesem, można ujrzeć niebieską ikonę (o kształcie małej kuli) na wykonywanym buildzie. Jeśli wystąpił jakikolwiek problem, ikona będzie żółta lub nawet czerwona. Należy zauważyć, że można również przeglądać wyniki testów interaktywnie poprzez kliknięcie na nazwę projektu, a następnie na „Latest Test Results”.

Rysunek 8. Wybrane książki traktujące o tematyce ciągłej integracji



Za nami jest już skonfigurowany serwer Hudson, dodane pierwsze zadanie i przebrnięcie przez proces zbudowania go. W przypadku produkcyjnej instancji, kompilacja i buildowanie będzie uruchamiane automatycznie. Uruchamianie odbywać się będzie poprzez zmiany w systemie kontroli wersji lub przez zegar. Podstawowy więc schemat prawdopodobnie będzie taki sam, mamy tylko inny typ wyzwalacza.

Zasoby informacji

Bardzo przydatna dokumentacja znajduje się w aplikacji jako pomoc kontekstowa. Jest również wiki na stronie internetowej projektu i na dwóch aktywnych listach mailingowych dla użytkowników i developerów.

Można również posiłkować się dostępnymi książkami na rynku, w których znajdziemy mocno rozbudowane opisy konfiguracji, instalacji oraz wielu przydatnych aspektów potrzebnych do pracy w tego typu środowisku. Na Rysunku 8 przedstawiono wybrane pozycje z tego zakresu.

DLACZEGO CIĄGŁA INTEGRACJA JEST WAŻNA

Minimalizacja ryzyka

Ze względu na utrzymanie kodu i minimalizację ryzyka utraty produktów pracy programiści powinni jak najczęściej zgłaszać nowy kod do repozytorium kodu. W ten sposób również programiści pracujący nad tymi samymi lub zależnymi komponentami mogą wcześnie zacząć pracować z nowym kodem. Jednakże samo zgłoszenie nowego kodu do repozytorium może jedynie wykryć konflikty w samym kodzie. To nie wystarcza, żeby określić, czy zmiana integruje się z kodem istniejącym w repozytorium.

Propagacja błędu

Kolejne problemy mogą pojawić się przy kompilacji, uruchamianiu czy korzystaniu

z aplikacji. Spójrzmy, jakie problemy taka sytuacja stwarza w zespole. Przede wszystkim późno wychwycony błąd może spowodować, że cały zespół będzie pracował na kodzie zawierającym błąd i budował kolejne funkcjonalności w oparciu o wadliwe działanie aplikacji, propagując błąd w systemie. Może także okazać się, że nie można uruchomić aplikacji i szukanie przyczyny usterki zajmie dużo czasu. Testerzy czy osoby odpowiedzialne za umieszczanie nowych wersji stracą czas na pobranie kodu z błędami, próbę kompilacji i uruchomienia aplikacji w środowisku testowym czy wreszcie na próbie przetestowania aplikacji i znalezieniu błędu krytycznego, po czym działanie jej zostanie natychmiast przerwane.

Czas

Popatrzmy na ilość czasu traconą na pobieranie, kompilowanie, umieszczanie aplikacji w środowisku testowym, w szczególności gdy aplikacja i tak nie działa, a te same czynności są wykonywane przez 3, 5, a może nawet 20 osób. Jest to doskonała ilustracja straty (*waste*). Jedną z zasad Lean Software Development jest minimalizacja strat (*minimazing waste*). System ciągłej integracji pozwala nam w pełni zautomatyzować wcześniej wymienione kroki, dzięki czemu zespół może spożytkować zaoszczędzony czas na dużo bardziej twórcze zadania.

Automatyczne testy regresji

Dodatkową zaletą automatycznego procesu ciągłej integracji z punktu widzenia testowania są automatyczne testy regresji, które dają nam dużą dozę pewności, że wcześniej dostarczona funkcjonalność po wprowadzeniu zmian nadal działa poprawnie. W projektach prowadzonych przy użyciu metodyk zwinnych każda zmiana powinna być natychmiast zintegrowana i dostępna do testów, a na testy regresji nie ma po prostu czasu w standardowych Sprintach (inaczej jest w *Release Sprint* czy *End Game*). Tutaj automatyczny system ciągłej integracji jest po prostu niezbędny. Kolejnym elementem usprawniającym pracę zespołu i podnoszącym efektywność jest odpowiednie ustawienie priorytetów: naprawa defektów, przegląd kodu oraz tworzenie nowego kodu.

Ewentualne defekty naprawia osoba komitująca kod lub jeśli akurat tej osoby nie ma, to ktoś inny z zespołu. Jedną z zasad pracy w projektach agile jest to, że kod należy do zespołu, a nie indywidualnych osób.

KONFIGUROWANIE ZADAŃ DLA IOS

Platforma iOS i CI

Każdy programista aplikacji iPhone, który releasował co najmniej jedną aplikację do AppStore'a, powie, że ten proces nie jest łatwy... no cóż, nie jest to zabawne doświadczenie. Po drugim lub trzecim releasie zaczynamy się powoli przyzwyczajać do tego procesu, ale nadal nie jest to przyjemne i komfortowe, jak powinno być. Zarządzanie certyfikatami, poprawnie skonfigurowane ustawienia oraz kolejne iteracje pomiędzy rozwojem aplikacji a buildem do testów (beta Ad-Hoc buildy a finalne wydanie buildu do AppStore). Cały przedstawiony powyżej schemat postępowania jest męczący do wykonania oraz wyjątkowo podatny na ludzkie błędy.

W profesjonalnie działających firmach, gdzie inżynieria oprogramowania jest na wysokim poziomie, przeważnie używa się serwera ciągłej integracji, aby monitorować repozytorium kodu, pobierać zmiany, jak są one komitowane, kompilować je, testować i budować pakiety. Po czym powiadamiać developerów o stanie kompilacji za pośrednictwem poczty elektronicznej oraz stron www zawierających rozbudowane dashboardy, informujące o wielu aspektach dotyczących buildu. Warto więc pomyśleć o przeniesieniu dobrych praktyk rozwoju na swoją platformę iOS.

Dlaczego warto skonfigurować automatyczne kompilacje

Spójrzmy prawdzie w oczy: jesteśmy ludźmi i popełniamy błędy. Naturalną sytuacją jest gdy podczas procesu rozwoju oprogramowania zdarza się nam popsuć pewien fragment tworzonej(go) aplikacji/systemu. Nie jest to niczym nadzwyczajnym. Jednak przeważnie (a często tak bywa) odkrywamy to zbyt późno. Powodem jest konieczność sprawdzania kolejnych (jak nie wszystkich) przypadków użycia (use cases) danego komponentu, który był zmieniany. W naturalny sposób nie zrobimy tego dokładnie za każdym razem, co powoduje niezauważenie błędu. Więc z pewnością idealnym rozwiązaniem będzie konfiguracja automatyzacji takiego procesu. Wtedy za każdym razem, gdy nastąpi zmiana, system powiadomi nas o ewentualnym niepowodzeniu. A jeśli działamy w projekcie multi-developerskim, będziemy mogli dokładnie zobaczyć, kto uszkodził build i jaka konkretnie zmiana to spowodowała.

Zawsze gotowy do dystrybucji aplikacji

Wiele razy w naturalnym przebiegu procesu developingu można popsuć kod. Czasem potrzeba coś zniszczyć, aby poprawić coś innego. Ale czasem ktoś (żona, klient, beta tester) będzie chciał wypróbować aplikację przed tym, nim zakończysz swoje dotychczasowe modyfikacje. I akurat poproszą o ten build w tym właśnie czasie, jak aplikacja jest praktycznie niezdolna do testowania. Zamiast wieku spędzonych chwili na back-trackingu i revertowaniu swojej pracy (aby przywrócić aplikację do kompilacji), można ustawić automatyczny system buildowania, który przechowuje archiwa wcześniej zakończonych sukcesem buildów.

Ponieważ chcesz przetestować aplikację przed releasowaniem jej do AppStore, prawdopodobnie będziesz tworzył build typu Ad-Hoc, aby dać go znajomym, rodzinie lub oficjalny build dla beta testerów. Możliwe, iż testerzy znajdą błędy, ale może się zdarzyć, że nie. Skompilowana aplikacja, którą właśnie utworzyłeś, tak w rzeczywistości nie jest tym, co będzie releasowane do AppStore'a. Musisz skompilować zupełnie inny pakiet aplikacji z bardzo różnych plików, a jeśli nie jesteś ostrożny, możesz potencjalnie releasować coś innego niż to, co było testowane.

Czy nie lepiej było by skonfigurować automatyczny build naszego systemu, który tworzy zarówno Ad-Hoc, jak i wersje AppStore za każdym razem? W ten sposób jesteśmy nie tylko zawsze gotowi releasować aplikację do AppStore, ale możemy mieć gwarancję, że wysyłamy do AppStore dokładnie ten sam kod, który testerzy sprawdzili.

Jeśli jesteśmy naprawdę rozważni i stosujemy się do dobrych praktyk, prawdopodobnie będziemy chcieli pisać unit testy dla kodu, oraz potrzebujemy mieć je uruchomione po skompilowaniu kodu (ale zanim build jest pakowany i archiwizowany). To, że kod kompiluje się, nie znaczy, że będzie zachowywać się poprawnie. I powiedzmy sobie szczerze, jeśli mamy dużo testów, nigdy nie będziemy czekać na to, aby uruchamiać za każdym razem wszystkie z nich (w trakcie swojej codziennej pracy). Więc uruchamiając swoje testy, jako warunek wstępny do budowania, masz gwarancję pewnej jakości kodu i poczucie bezpieczeństwa.

Jest wiele innych dobrych praktyk, w których posiadanie automatycznego systemu budowania może pomóc, zatem te, które tutaj omawiam, to tylko wierzchołek góry lodowej. Jeśli powoli przekonuję Cię do automatyzacji buildów, to czytaj dalej.

Hudson – build serwer

Po dokonaniu oceny kilku rozwiązań, skończyło się na platformie Hudson jako build serwerze. Ważnym faktem jest, iż obsługuje agenty budowania typu *slave*, co jest niezbędne dla opracowania buildu aplikacji dla iPhone'a. Pierwszym krokiem będzie pobranie i uruchomienie Hudsona (co zostało wcześniej przedstawione). Najlepiej jest go uruchomić np. na odrębnym serwerze. Dobrym pomysłem jest serwer Linux, który ma skonfigurowany e-mail i zawiera repozytorium kodu, więc z pewnością możemy uruchomić tam Hudsona. Hudsona można używać do buildowania bardzo różnych aplikacji, więc posiadanie go w centralnym miejscu ułatwia życie.

Po uruchomieniu i sprawdzeniu jego działania, można doinstalować parę wtyczek, które pomogą połączyć się z repozytorium kodu źródłowego, lub pozwalają wygenerować różne typy wiadomości (np. build wysyła wiadomość błyskawiczną na komunikator GTalk, kiedy kompilacja się nie powiedzie). Zawsze można w opcjach konfiguracji zmienić ustawienia, co może być dla nas korzystne, więc warto poeksperymentować.

Zadanie budowania

Aby skonfigurować zadanie, należy iść do dashboardu Hudsona i kliknąć na „New Job". Po wpisaniu nazwy potrzeba wybrać projekt oprogramowania „Freestyle" (*Freestyle software project*). Oznacza to po prostu, że chcesz uruchomić skrypt powłoki, który opisuje, w jaki sposób budowa ma przebiegać.

Następnie specyfikujemy lokalizację kodu źródłowego, podając właściwe URL dla repozytorium. Serwer posiada wsparcie dla Git, Subversion, Perforce, a nawet CVS. Większość ustawień domyślnych potrzebnych do budowy z reguły jest wystarczające do pracy. Niemniej jednak to Ty jesteś ekspertem, więc sam możesz zdecydować, jaka konfiguracja będzie optymalna dla twojego projektu.

Po wprowadzeniu informacji o miejscu kodu źródłowego, należy określić, co następnie z nim zrobić.

Pod sekcją „Build", klikamy na „Add build step" i wybieramy „Execute shell". To pozwoli nam wpisać nazwę skryptu do wykonania. Wystarczy wpisać tam:

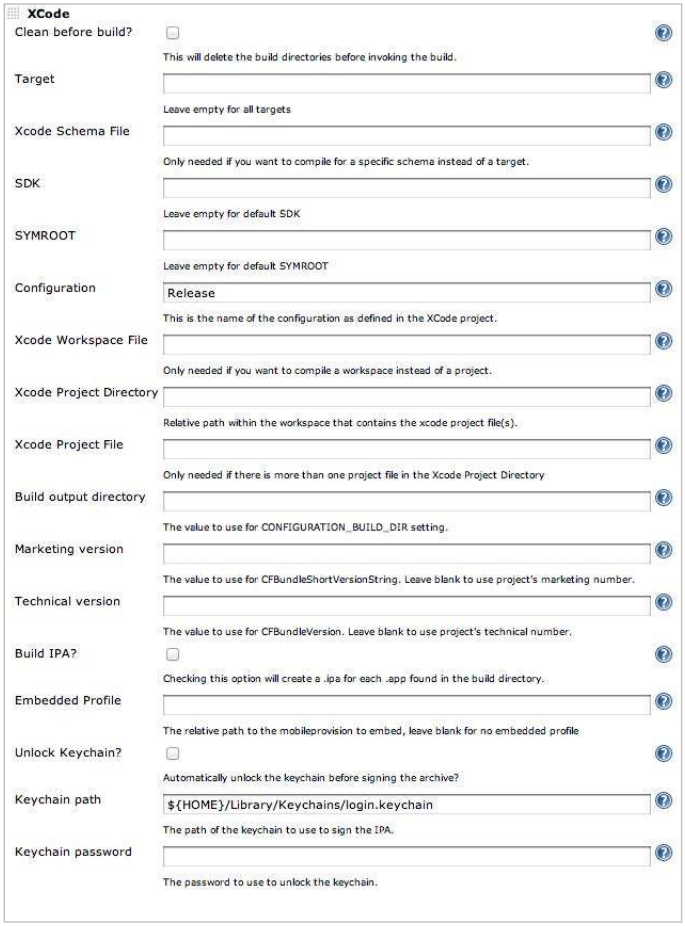
```
"$WORKSPACE/build/build.sh"
```

Hudson będzie pobierał kod do tego katalogu, w którym aplikacja zostanie zbudowana, oraz zmienna $WORKSPACE zostanie ustawiona na pełną ścieżkę miejsca, gdzie znajduje się ten katalog. Istnieją również inne zmienne zdefiniowane w środowisku, które są opisane na stronie konfiguracji zadania. Uruchamiając powyższe zadanie, tak naprawdę będziemy wykonywać polecenia znajdujące się w skrypcie „**build.sh**", który powinno się dodać do swojego projektu i repozytorium kodu źródłowego. W ten opisany sposób można budować swój projekt, może być on wersjonowany w zależności od swojego aktualnego kodu.

Skrypt kompilacji

Oczywiście, że potrzeba skryptu, który będzie wydawał wszystkie niezbędne komendy potrzebne do budowania aplikacji. Na szczęście firma Apple dostarcza narzędzie wiersza polecenia o nazwie „xcodebuild", które jest używane do kompilacji aplikacji. W rzeczywistości jest to ta sama komenda, którą XCode wykonuje po kliknięciu przycisku "Build" w interfejsie użytkownika. Wiele przykładów takiego skryptu można odnaleźć na stronach poświęconych konfiguracji buildu dla iOS. Mają one z reguły spore rozmiary (w zależności od potrzebnych opcji buildu), więc nie będą tutaj prezentowane (pozostawiam czytelnikowi to do sprawdzenia).

Obecnie również można doinstalować wtyczkę do serwera (XCode-plugin), która umożliwi konfigurację buildu bez użycia skryptu (działa podobnie jak wbudowana funkcjonalność dla Javy). Generalnie produkuje ona za nas polecenia skryptu, kolejno wykonując wymagane kroki.



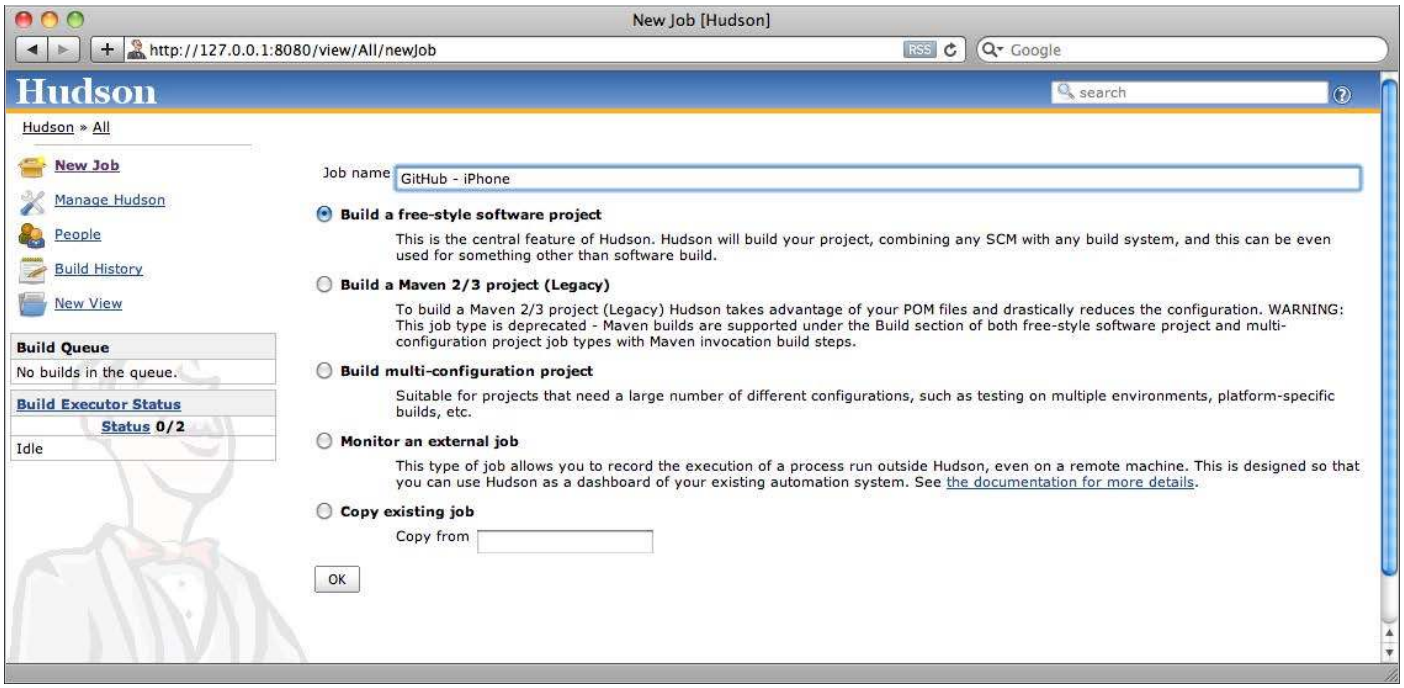
Rysunek 9. Rozszerzenie Hudsona poprzez XCode plugin – panel konfiguracji

Przykład tworzenia buildu iOS - z repozytorium typu GitHub

Zacznijmy więc konfigurację przykładowego projektu iOS na Hudsonie. Należy kliknąć na „New Job" i wpisać „GitHub - iPhone" jako przykładową nazwę projektu. Następnie potrzeba wybrać „Build a free-style software project" i nacisnąć przycisk „OK", zatwierdzając zmiany (Rysunek 10).

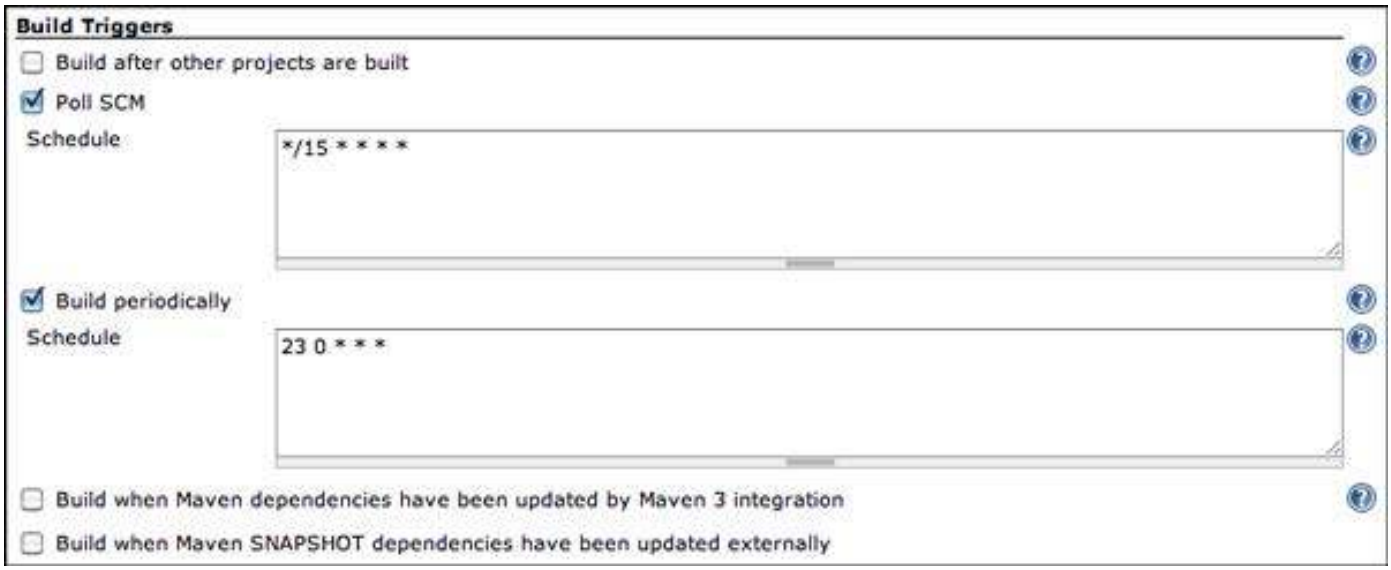
Ekran konfiguracyjny, który zobaczymy w następnym kroku, może wydawać się przytłaczający ze względu na dużą liczbę dostępnych opcji. Dalej pokażę kluczowe wartości, które potrzeba podać, aby sprawnie ustawić działający build. Absolutnie niezbędne jest połączenie naszego systemu do repozytorium kodu źródłowego. Musimy więc wybrać Git i wprowadzić adres URL repozytorium, jak pokazano na Rysunku 11.

Jeśli używamy starszej wersji Hudsona i pole wyboru Git nie jest dostępne, musimy wtedy zainstalować plugin Git, przed dalszą kontynuacją konfiguracji tego typu zadania.

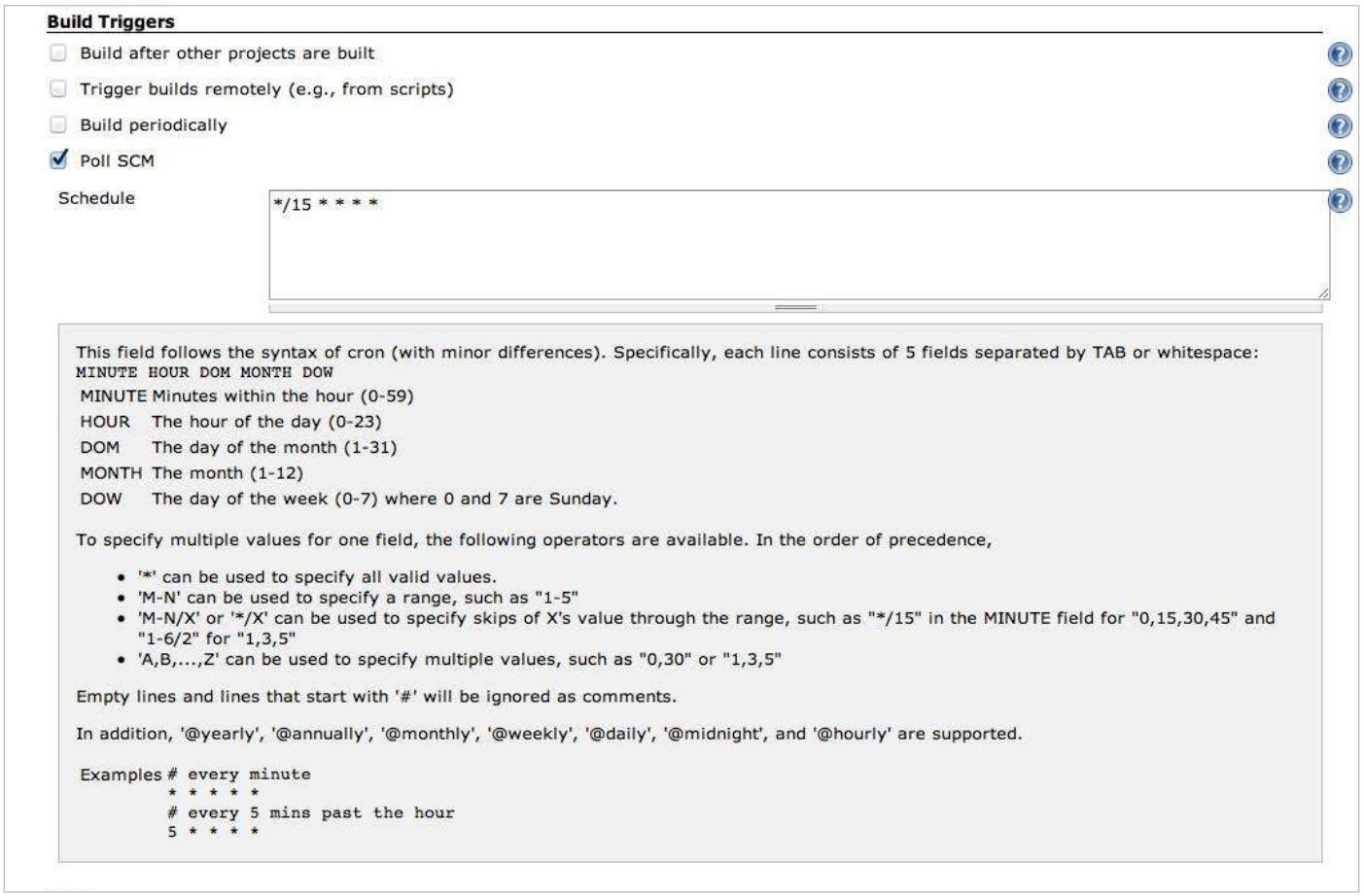


Rysunek 10. Konfiguracja nowego zadania

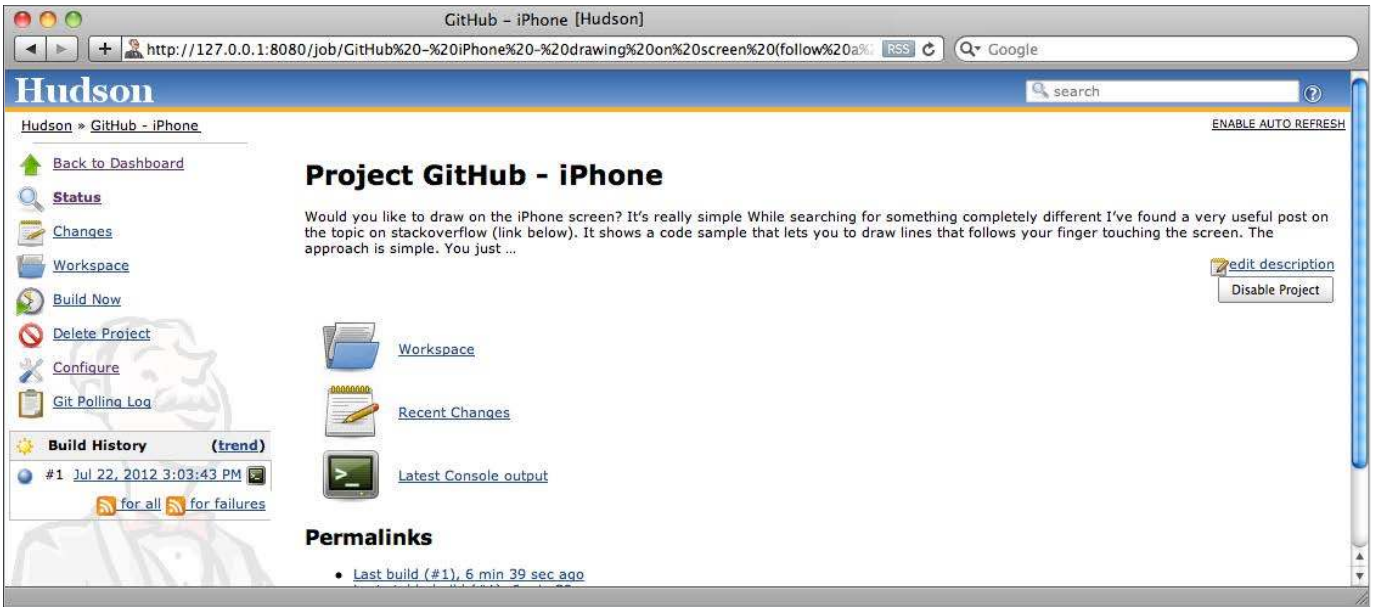
Rysunek 11. Definiowanie typu oraz adresu URL dla repozytorium kodu



Rysunek 12. Definiowanie częstotliwości budowania



Rysunek 13. Definiowanie wyzwalacza SCM pulling



Rysunek 14. Zadanie po konfiguracji

Rysunek 15. Panel historii buildów oraz możliwe statusy

Idąc dalej, musimy określić kiedy, nasz build powinien być wyzwalany. W poniższym przykładzie użyjmy dwóch wyzwalaczy. Okresowe codzienne budowanie, które rozpoczyna pracę o godzinie 23:00 – zobrazowane na Rysunku 12.

Drugim wyzwalaczem będzie sprawdzanie repozytorium, przy użyciu **SCM pulling** co 15 minut, aby zbadać, czy nie było nowych zmian w kodzie (Rysunek 13).

Podczas konfiguracji, jeżeli pewna opcja jest dla nas nowa i nie wiemy, co ona oznacza, warto kliknąć na ikonę pomocy po prawej stronie, aby zobaczyć opis wraz z przykładem użycia. Ostatnim obowiązkowym krokiem jest powiedzieć Hudsonowi, jak ma budować nasz projekt. Dla projektów iPhone/iOS oczywistym jest użycie polecenia **xcodebuild** (jako polecenia powłoki). Wywołanie komendy może wyglądać jak poniżej:

```
xcodebuild -project "SampleProject/SampleProject
.xcodeproj" -target "SampleProject" -sdk /Developer/
Platforms/iPhoneSimulator.platform/Developer/SDKs/
iPhoneSimulator5.0.sdk/ -configuration "Debug"
```

Gorąco zachęcam do zapoznania się ze wszystkimi dostępnymi opcjami konfiguracji. Warto używać wcześniej wspomnianej pomocy widocznej po prawej stronie, by otrzymać dodatkowe informacje o każdym z nich. Osobiście zawsze używam poniżej przedstawionych opcji:

- **Description** – dla przejrzystości warto opisać zadanie. Gdy posiadamy wiele projektów i nieznacznie się one różnią, można być niepewnym, co dane zadanie robi.
- **Discard Old Builds, Days to keep builds, Max # of builds to keep** – czas, ilość oraz przetrzymywanie starszych buildów, aby nie zaśmiecać naszego serwera niepotrzebnymi binariami.
- **E-mail Notification** - bardzo przydatna opcja, niestety do pracy będzie nam potrzebny serwer SMTP. Serwer więc potrzeba skonfigurować w sekcji: *Hudson/Manage Hudson/ Configure System*.

Możemy wywołać proces budowania, wykorzystując przycisk „Build Now", aby sprawdzić, czy wszystko jest poprawnie skonfigurowane. Build powinien być udany, gdy Hudson zakończy pracę.

Od tej pory automatyczne buildy będą uruchamiane co 15 minut, jeśli ktoś wkomituje dowolne zmiany do repozytorium, a także codziennie każdej nocy o godzinie 23:00.

Patrząc na pole „Build History" po lewej stronie, można sprawdzić, co jest aktualnie budowane oraz jaki ma status.

KORZYŚCI Z ZASTOSOWANIA CI W WYTWARZANIU OPROGRAMOWANIA

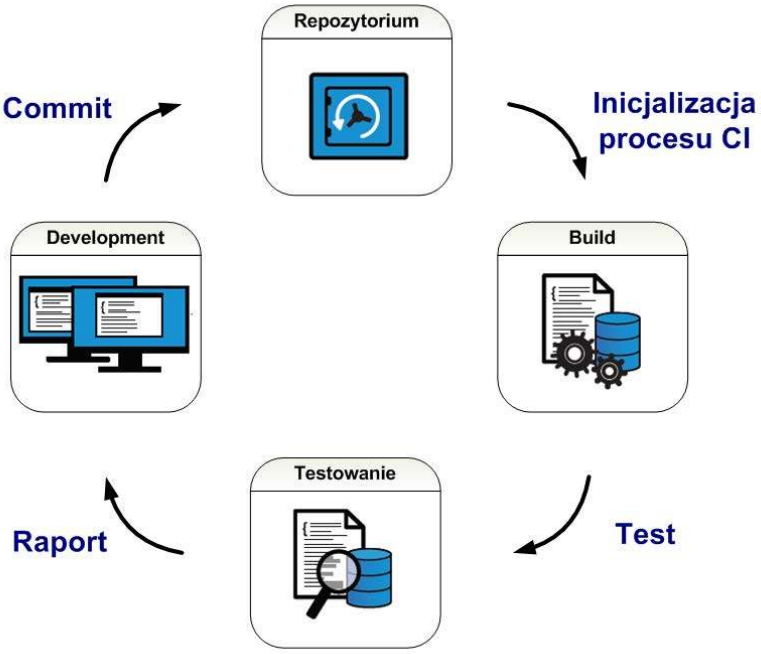
Zastosowanie ciągłej integracji niesie za sobą wiele korzyści – zarówno doraźnych, których efekty widać w danym projekcie, jak i długoterminowych, na których mogą skorzystać przyszłe projekty prowadzone przez organizację.

Jakość i jej kontrola

Serwery ciągłej integracji nie wpływają bezpośrednio na jakość produktu, jakim jest oprogramowanie. Pozwalają jedynie tę jakość kontrolować. Środowisko ciągłej integracji należy więc raczej postrzegać jako miernik jakości wytwarzanego oprogramowania.

Niemniej do korzyści z wdrożenia ciągłej integracji w procesie produkcji oprogramowania należy zaliczyć wymuszenie automatyzacji testów (i innych mechanizmów kontroli jakości).

Ogromną zaletą automatycznych testów jest ich powtarzalność. Raz skonstruowany test może zostać przeprowadzony przez komputer wielokrotnie, w przeciwieństwie do ręcznych testów, w których nakład czasu i zasobów potrzebny do przeprowadzania testu jest w zasadzie identyczny za każdym razem. Dzięki testom automatycznym możliwe jest ciągłe testowanie aplikacji, co z kolei pozwala stale kontrolować jej jakość, jak podkreśla w swojej pracy Martin Fowler.



Rysunek 16. Cykl CI (continuous integration)

Czas wykrycia błędu

Ogromne znaczenie ma też fakt, że przy zastosowaniu ciągłej integracji skraca się średni czas od powstania błędu do jego naprawienia. Dzieje się tak dlatego, że tuż po wprowadzeniu błędnego kodu do repozytorium, serwer ciągłej integracji pobiera go i sprawdza, próbując budować aplikację. W przypadku wykrycia błędu, serwer ciągłej integracji zgłasza błąd i powiadamia o tym zainteresowanych programistów w czasie rzędu minut od dokonania zmiany. Dzięki temu twórcy są na bieżąco ze zmienianym fragmentem aplikacji i nie muszą poświęcać czasu na przypomnienie architektury. Przykładowo, szybciej jest znaleźć przyczynę błędu, jeśli istnieją tylko dwie zmiany do zbadania od ostatniej integracji, a nie 50 zmian. Ponadto łatwiej jest wyeliminować błędy poprzez wstępną identyfikację przyczyny, gdy zmiana jest jeszcze świeża w umyśle developera. Może on dokładnie pamiętać, co zrobił, oraz dlaczego tak to zostało zrobione (czyli sposób implementacji).

Innym powodem skuteczności CI jest to, że gdy wiele zmian jest integrowane razem ze sobą, ich łączny wpływ może doprowadzić do nieprzewidywalnych błędów, a praktyka CI pozwala na znalezienie tych błędów przed faktem wysłania oprogramowania do QA.

W podejściu tradycyjnym błędy są najczęściej wykrywane dużo później, tj. pod koniec projektu, albo co gorsza po wdrożeniu produktu do użytku, kiedy naprawienie błędu jest dużo bardziej kosztowne.

Ogólnie rozumianą zasadą jest jak najszybsze poprawianie usterek. Dłuższe pozostawanie nienaprawionych błędów skutkuje mniejszą dbałością programistów. Widząc nienaprawione błędy, programiści są mniej ostrożni i łatwiej o nowe usterki.

Różne konfiguracje

Automatyzacja procesu budowania umożliwia ponadto wspieranie różnych konfiguracji systemowych – np. uruchamianie aplikacji na różnych platformach sprzętowych, testowanie z użyciem różnych systemów zarządzania bazą danych, testowanie webowego interfejsu użytkownika z użyciem różnych przeglądarek internetowych itd. W większości organizacji programiści na swoich komputerach instalują tylko jedną ze wspieranych konfiguracji. Nie sposób wymagać, by każdy z nich instalował u siebie różne systemy operacyjne, systemy baz danych, przeglądarki i stworzony przez siebie kod testował w działaniu w wielu konfiguracjach.

PODSUMOWANIE

W artykule zostało omówione wykorzystanie podejścia ciągłej integracji w trakcie rozwoju oprogramowania. Zostały również krótko opisane dwa serwery CI i możliwości, które otrzymujemy wraz z ich użyciem. Zaprezentowana została także krótko konfiguracja projektu na Hudsonie.

Ciągła integracja jest praktyką tworzenia oprogramowania, bez której większość zespołów nie mogła by istnieć. Z Hudsonem/Jenkinsem niezbędne oprzyrządowanie jest szybkie w instalacji, łatwe w utrzymaniu i skalowalne dla przyszłych rozszerzeń. Hudson/Jenkins nie ogranicza się tylko do świata Java. Wiele firm korzysta z serwera CI, aby zautomatyzować swoje wysiłki rozwoju oprogramowania w C/C + +, Objective-C, C#, PHP, Groovy, Scala i wielu innych.

Jednym z niezbędnych narzędzi potrzebnych do wdrożenia tego podejścia jest serwer CI. Hudson/Jenkins są jednymi z najbardziej popularnie stosowanych serwerów ciągłej integracji. Konkurują z wysoko stojącymi produktami komercyjnymi takimi jak Bamboo i TeamCity.

Zespoły praktykujące podejście ciągłej integracji są w stanie dostarczyć oprogramowanie o wiele szybciej niż zespoły, które czekają, aż projekt jest gotowy, aby zrobić integrację i testowanie. Dostarczane oprogramowanie zazwyczaj również charakteryzuje się dużo mniejszą ilością błędów. Istotą jest to, że błędy są odkrywane znacznie wcześniej, głównie w ciągu kilku minut od czasu ich wprowadzenia, a w tym czasie jest o wiele łatwiej i taniej je naprawić.

W sieci

- http://hudson-ci.org/ – strona domowa projektu Hudson
- http://wiki.hudson-ci.org/display/HUDSON/Extend+Hudson – rozszerzenia i dodatki
- http://wiki.eclipse.org/Hudson-ci/documentation – dokumentacja projektu Hudson
- http://jenkins-ci.org/ - strona domowa projektu Jenkins
- http://jenkins-ci.org/views/hudson-tutorials – wybrane tutoriale dla projektu Jenkins
- http://blog.jazzy.pro/jenkins-android/ – artykuł opisujący konfigurację Jenkinsa dla platformy Android
- http://bobbickel.blogspot.ie/2011/03/jenkins-vs-hudson-time-to-upgrade.html – artykuł opisujący migrację na platformę Jenkins
- http://www.youtube.com/watch?v=RRoLabeUQ88&feature=related – film przedstawiający tworzenie i konfigurowanie nowego zadania dla Jenkinsa
- http://nachbaur.com/blog/how-to-automate-your-iphone-app-builds-with-hudson – artykuł opisujący automatyzację buildów dla platformy iOS
- http://nachbaur.com/blog/building-iphone-apps-with-hudson-part-2 – automatyzacja buildów dla platformy iOS – rozszerzenie

Łukasz Mazur lukash.mazur@gmail.com

Autor obecnie jest programistą, architektem aplikacji biznesowych dedykowanych na platformy mobilne. Pracuje dla irlandzkiej firmy Mobile Travel Technologies Ltd. Jest jednym ze współtwórców deweloperów aplikacji mobile easyJet na system iOS, uznawanej za jedną z najlepiej opracowanych mobilnych aplikacji w swoim segmencie. Aktualnie autor bierze udział w dużych projektach mobilnych aplikacji, dla linii lotniczych Jetstar oraz Transavia.

Marek Sawerwain

Kropkowe nowości – czyli dot NET 4 i 1/2

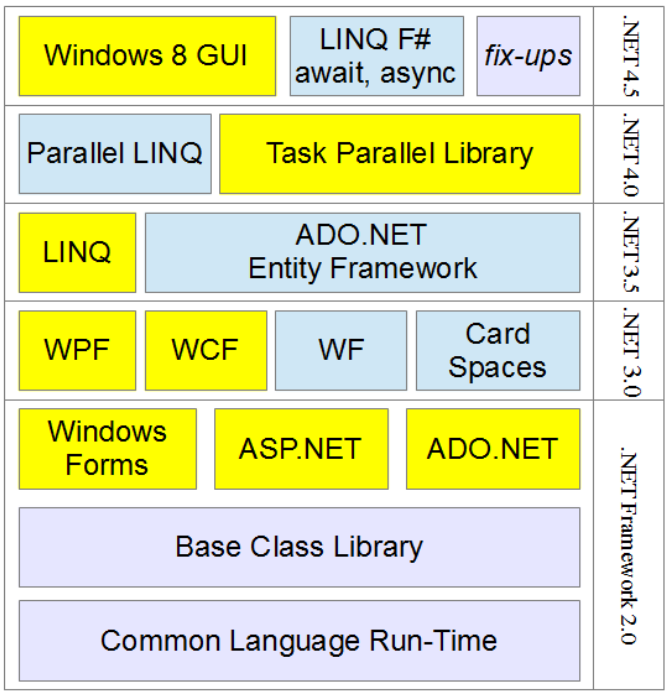
Jesień roku 2012 dla programistów systemu Windows będzie niezwykle bogata: nowy Windows 8, nowe Visual Studio 2012, a także nowa odsłona .NET o numerze 4.5. W artykule postaramy się przedstawić kilka wybranych nowości dostępnych w najnowszej wersji tej platformy.

Najnowsze wydanie platformy .NET, która oficjalnie zadebiutowała 15 sierpnia 2012, zbiega się z nowym wydaniem Windows 8 oraz z debiutem interfejsu użytkownika o nazwie Metro. W trakcie pracy nad tym artykułem pojawiły się pogłoski, iż Microsoft ma w planach inną nową nazwę dla interfejsu użytkownika, i okazało się, iż zamiast o interfejsie Metro będziemy mówić o interfejsie Windows 8. Niewątpliwie nowy system i obsługa funkcji interfejsu, jaki ze sobą przynosi, to najważniejsze nowości w .NET 4.5, ale nie tylko. Oprócz ulepszeń w głównych filarach .NET wprowadzono także pewne zmiany w głównych językach .NET, jak Visual Basic, C#, ale także F#.

Ogólnie nowe zmiany można podzielić na kilka obszarów platformy .NET. Niewątpliwie pierwszym elementem są zmiany związane z Windows 8, czyli aplikacje Metro (ang. *.NET for Metro style Apps*). Zmiany wprowadzono także do zbioru klas bazowych. Wprowadzono pomniejsze zmiany w .NET, ale unowocześnieniu uległy również rozwiązania programowania równoległego, a także pakiety związane z obsługą sieci oraz aplikacji WEB. Microsoft uaktualnił też inne pakiety wchodzące w skład .NET, a są to: pakiety do tworzenia GUI Windows Presentation Foundation (WPF), technologia komunikacji Windows Communication Foundation (WCF) oraz system zarządzania procesami, czyli Workflow Foundation (WF).

Z drugiej strony, jeśli chodzi o języki programowania związane z .NET, to nie ma tak dużych zmian w dwóch głównych językach platformy, czyli Visual Basic oraz C#. Wprowadzono dwa nowe słowa kluczowe, aby usprawnić programowanie asynchroniczne. Większą zmianą jest wprowadzenie obsługi zapytań LINQ do języka F#, co z pewnością zwiększy popularność tego języka.

W dalszej części artykułu omówimy pewne wybrane nowe elementy .NET, opisując zmiany w bibliotekach platformy oraz nowe elementy, jakie wprowadzono do języków dostępnych w .NET. Ominiemy nowości związane z Windows 8, bowiem jest to temat godny oddzielnego opracowania, do którego na łamach „Programisty" powrócimy po oficjalnej premierze nowych okienek.



Rysunek 1. Główne elementy platformy .NET wprowadzane w kolejnych edycjach

TROCHĘ HISTORII O GŁÓWNYCH EDYCJACH .NET

Warto wspomnieć o poprzednich edycjach .NET (pierwsza edycja pojawiła się w roku 2002), ponieważ nowe wydanie, sugerując się numerem wersji 4.5, pozornie nie wprowadza rewolucyjnych zmian, choć naturalnie współpraca z Windows 8 (.NET 4.5 jest obecny w Windowsie 8.0) to zasadnicza nowość w platformie .NET.

Należy przyznać, iż większe zmiany zostały wprowadzone w poprzednich edycjach, np. w edycji .NET 2.0 (luty 2006) pojawiły się typy uogólnione. Nowe technologie takie jak Windows

Visual Studio 2012 i kod źródłowy .NET 4.5

Nowe wydanie 4.5 to także nowa odsłona środowiska Visual Studio 2012. Podobnie jak .NET 4.5 nowa wersja VS 2012 w wersji RTM ukazała się w sierpniu 2012. Jak zawsze można ściągać wersję trial działającą przez 30 dni z możliwością przedłużenia do 90 dni, po darmowej rejestracji.

Nowe wydanie przynosi wiele przydatnych zmian, mamy inny interfejs graficzny dopasowany do Windows 8, który sprawdza się też w Windows 7. Jednak ważną zmianą są wymagania, nowe Visual Studio zainstalujemy tylko na Windows 7 albo 8.

Nowości jest sporo, warto np. podkreślić dobre narzędzia do debugowania aplikacji równoległych C#, wsparcie dla nowego standardu C++11, a także dla C++AMP (wydajne obliczenia realizowane za pomocą kart graficznych). Obsługę zyskał też JavaScript oraz HTML5.

Cenną innowacją jest możliwość sięgnięcia z poziomu Visual Studio do przykładów on-line dotyczących nowych elementów, np. F# czy systemu Windows 8. Ulepszono IntelliSense dla wszystkich głównych języków. Pośród licznych zmian i innowacji do Visual Studio wprowadzono możliwość, co może wydawać się dziwne, oglądania obiektów 3D w formacie Collada, FBX czy OBJ. Wszyscy ci, który korzystają z XNA i tworzą aplikacje oparte o grafikę 3D, z pewnością docenią takie rozszerzenie.

Należy też wspomnieć o możliwości pobrania kodu źródłowego części bibliotek .NET 4.5 spod adresu:

http://referencesource.microsoft.com/netframework.aspx

Warto to zrobić, jeśli chcemy samemu przekonać się, jak zostały zaimplementowane różne składniki .NET oraz uzyskać możliwość głębszego debugowania własnej aplikacji.

Presentation Foundation (WPF - technologia odpowiedzialna za interfejs użytkownika) czy Windows Communication Foundation (WCF - komunikacja pomiędzy obiektami i aplikacjami) pojawiły się w wydaniu 3.0 (listopad 2006). Zintegrowany język zapytań LINQ, który bardo dobrze został przyjęty przez programistów, pojawił się w wydaniu 3.5 (listopad 2007) i został wprowadzony do języków C# oraz Visual Basic. Wydanie 4.0 (kwiecień 2010) to przede wszystkim wprowadzenie klas do obsługi programowania równoległego, pojawiło się też wsparcie dla funkcyjnego języka programowania F#.

Pełna obsługa baz danych ADO.NET oraz aplikacje sieciowe ASP.NET pojawiły się w wersji 2.0. Ogólnie oceniając, to wersje 2.0 oraz 3.0 były przełomowe dla platformy .NET. Główną nowością wersji 4.0 było wprowadzenie klas do programowania równoległego oraz języka F#. Wersja 4.0 oferuje także wsparcie dla aplikacji sieciowych tworzonych w architekturze MVC (Model View Controller).

Najnowsza edycja, wprowadzona 15 sierpnia 2012, wnosi oprócz różnych zmian, które pokrótce przedstawimy, jeszcze jedną ważną zmianę, a dotyczy ona wymagań, gdyż wydanie 4.5 możemy instalować tylko dla systemów Windows Vista, Windows 7, Windows 8 - poprzednie wersje systemów nie są wspierane, co stanowi pewną klamrę w rozwoju systemu Windows.

OGÓLNIE O NOWOŚCIACH W 4.5 .NET

W dalszej części artykułu omówimy nieco dokładniej nowości związane z językiem C# oraz F#, a w tym miejscu przedstawimy ogólnie, jakie nowe elementy zostały wprowadzone do platformy .NET, oprócz wsparcia dla Windows 8.

Pierwszy nowy element, jaki warto wymienić, to określenie przenośnego zbioru klas .NET – Portable Class Library. Pozwala to na budowę aplikacji .NET z poziomu Visual Studio współpracującej z różnymi wersjami oraz edycjami .NET. Nie chodzi tylko o starsze wydania .NET, ale np. o budowę aplikacji, która będzie zdolna do pracy w ramach pełnej edycji .NET dla systemu Desktop, jak również dla technologii Silverlight, Windows Phone oraz konsoli XBOX 360.

Ważnym elementem jest wsparcie dla dużych struktur danych, np. tablic, bowiem nowy .NET pozwala w przypadku systemów 64bitowych tworzyć tablice większe niż 2 GB. Wykorzystanie tej funkcji wymaga jednak zmiany w pliku konfiguracyjnym, a dokładniej dopisania następującego wyrażenia XML, w ramach sekcji **configuration/runtime**:

```
<gcAllowVeryLargeObjects enabled="true" />
```

Pomimo iż, jak widać, nie jest to domyślna możliwość, może się okazać bardzo przydatna, jeśli pracujemy z dużymi tablicami.

Inną przydatną modyfikacją jest określenie czasu, jak długo system odpowiedzialny za wyrażenia regularne ma analizować dane wyrażenie. Można określić czas maksymalny, po którym analiza zostanie przerwana. Pomocne może się okazać wsparcie dla kodowania UTF-16 dla konsoli (klasa Console), dotychczas obsługiwane było kodowanie UTF-8.

Przydatną zmianą jest pojawienie się nowej przestrzeni nazw **System.IO.Compression**. Zgodnie z nazwą obsługiwana jest kompresja strumieni danych w formacie ZIP. Gotowe klasy do obsługi formatu ZIP oraz metod kompresji z pewnością ucieszą wielu programistów, choć naturalnie nie brakuje bibliotek do obsługi metod kompresji dla platformy .NET.

Już poprzednia wersja 4.0 przyniosła nowe rozwiązania dla przetwarzania równoległego. Wydanie 4.5 oferuje głównie ulepszenia względem poprzedniej edycji, dotyczące wydajności, większej kontroli. Ułatwiono tworzenie aplikacji asynchronicznych; odpowiedni przykład znajduje się w dalszej części artykułu. Ulepszony został także debugger oraz Concurrency Visualizer, czyli narzędzie pokazujące współbieżność aplikacji, co pozwala na łatwiejszą ocenę, czy istotnie projektowany przez nas program poprawnie rozkłada obliczenia na wiele wątków i czy obciążenia rozkładają się równomiernie. Z tego narzędzia skorzystamy naturalnie po instalacji Visual Studio 2012.

Wprowadzono także poprawki innego rodzaju, jak organicznie liczby restartów podczas instalacji frameworka .NET 4.5. W przypadku systemów wielordzeniowych, kompilator JIT funkcjonuje w tle, wykorzystując wolne rdzenie, co oczywiście przekłada się na większą wydajność pracy aplikacji .NET. Poprawiono także pracę „odśmiecacza" dla serwerów. Może on teraz pracować w tle, co również poprawia całkowitą wydajność aplikacji.

Istnieją też zmiany łamiące wsteczną kompatybilność, podstawową jest brak wsparcia dla Windows XP. Nieco inaczej mogą zachowywać się metody **MathCeiling** oraz **MathFloor** w przypadku stosowania .NET do aplikacji dla Windows Store. Dokładniejsze informacje o zmianach w zachowaniu różnych metod można odszukać pod adresem:

http://msdn.microsoft.com/en-us/library/hh367887.aspx

NOWE SŁOWA KLUCZOWE ORAZ CALLER-INFORMATION DLA C#

W języku C# pojawiły się dwie zasadnicze nowości. Pierwsza to wprowadzenie dwóch słów kluczowych **async** oraz **await**, ich zastosowanie pokrótce przedstawimy poniżej. Drugą nowością są dodatkowe atrybuty dostarczające informacje o źródłowej metodzie wywołującej. Krótko mówiąc, możemy dowiedzieć się, jaka metoda wywołała aktualną metodę. Mechanizm ten jest dostępny również w Visual Basicu i na nasze potrzeby będziemy go nazywać caller-information.

Listing 1 przedstawia fragment przykładu ilustrującego wykorzystanie mechanizmu caller-information. Dwie pierwsze metody **TestCaller1** oraz **TestCaller2** wywołują metodę **CallerInfoMessage**, która oferuje cztery argumenty, lecz w naszym przypadku tyko pierwszy jest dostępny dla programisty. W nim przesyłamy prosty komunikat, naturalnie mogą to być inne wielkości. Istotne są trzy dodatkowe parametry, opatrzone atrybutami **CallerMemberName**; atrybut ten jest odpowiedzialny za nazwę metody. Drugi atrybut - **CallerFilePath** - to ścieżka do pliku zawierającego wywołaną metodę oraz **CallerLineNumber**, zgodnie z nazwą to numer linii, z której nastąpiło wywołanie. Trzy dodatkowe argumenty posiadają wartości domyślne, jednak w momencie wywołania odpowiednie informacje są przekazywane do funkcji **CallerInforMessage**.

Użycie tej techniki wymaga dołączenia usług kompilatora oraz pakietu do diagnostyki:

```
using System.Runtime.CompilerServices;
using System.Diagnostics;
```

Najwygodniej wykorzystać obiekt **Trace**, do którego będą przesyłane wszystkie komunikaty. Wyjście tego obiektu można przekierować do pliku albo jak poniżej do strumienia wyjściowego skojarzonego z konsolą:

```
Trace.Listeners.Add(new TextWriterTraceListener(Console.Out));
Trace.AutoFlush = true;
```

Dwie powyższe linie należy umieścić przed metodami, które korzystają z techniki caller-information. Ważny element to ustalanie własności **AutoFlush** na **true**, w ten sposób nie trzeba wymuszać wyświetlania przekazywanych komunikatów, gdyż każde użycie **Trace.WriteLine(…)** będzie powodować wyświetlenie komunikatu. Mechanizm caller-information jest jak widać bardzo prosty do wykorzystania i aż szkoda, że należało czekać do wydania 4.5, aby korzystać z tego przydatnego mechanizmu.

Listing 1. Wykorzystanie mechanizmu Caller-Information

```
private void TestCaller1() {
   CallerInfoMessage("from TC1");
}

private void TestCaller2() {
   CallerInfoMessage("from TC2");
}

private void CallerInfoMessage(string message,
   [CallerMemberName] string memberName = "",
   [CallerFilePath] string sourceFilePath = "",
   [CallerLineNumber] int sourceLineNumber = 0) {
      Trace.WriteLine("message: " + message);
      Trace.WriteLine("member name: " + memberName);
      Trace.WriteLine("source file path: " + sourceFilePath);
      Trace.WriteLine("source line number: " +
sourceLineNumber);
}

public void TestCaller() {
   Trace.Listeners.Add(new
TextWriterTraceListener(Console.Out));
   Trace.AutoFlush = true;
   TestCaller1();
   TestCaller2();
}
```

SŁOWA KLUCZOWE ASYNC AND AWAIT

Dwa dodatkowe słowa kluczowe **async** oraz **await** upraszczają problem obsługi synchronicznych oraz asynchronicznych metod, która stosuje się w przetwarzaniu równoległym. Wbrew pozorom nie chodzi w tym momencie o skomplikowane i wyrafinowane algorytmy, ale o bardziej prozaiczne zadania jak np. wczytywanie większego pliku XML czy też odczyt danych z sieci Internet.

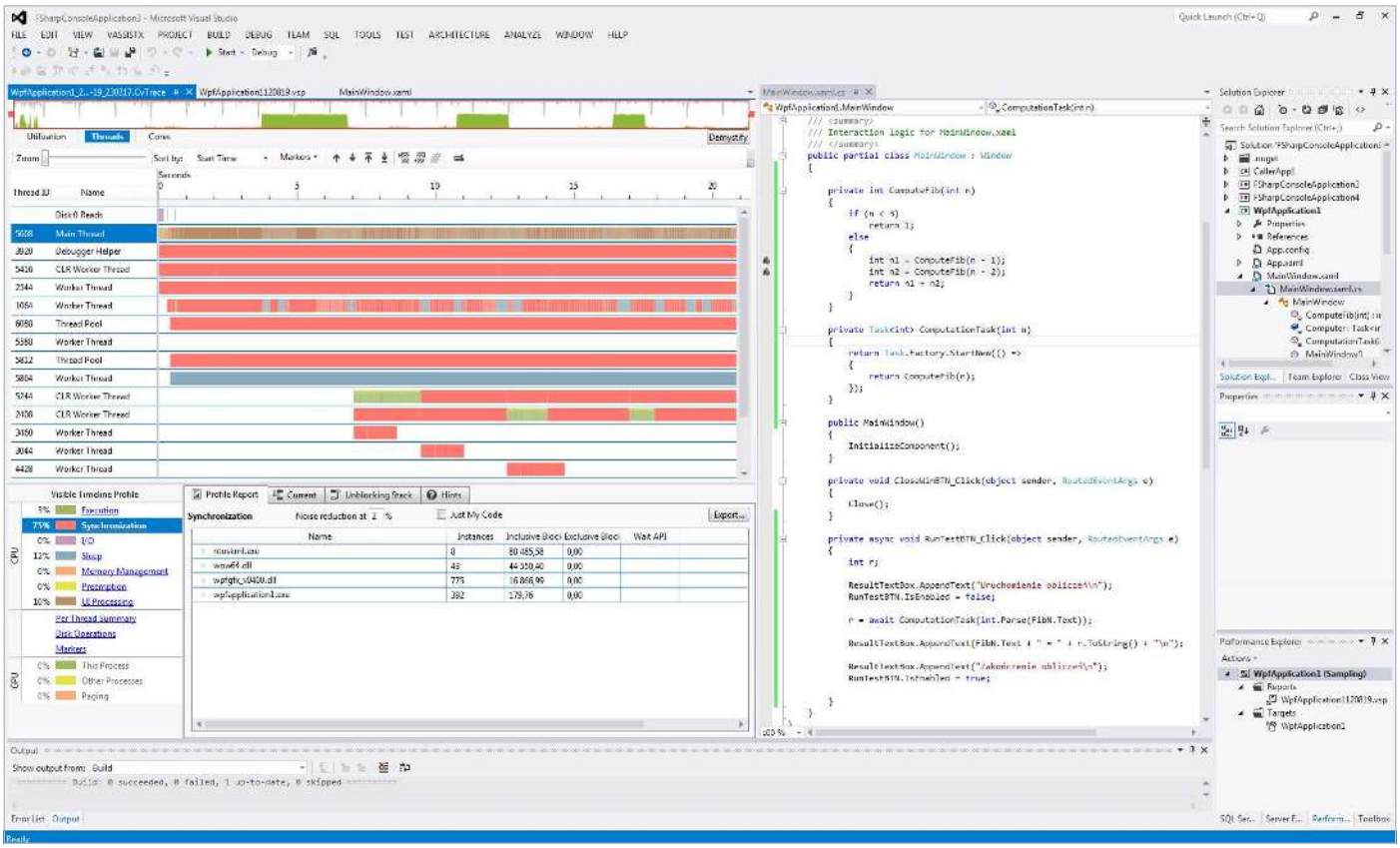
Użycie dwóch nowych słów kluczowych upraszcza tworzenie programów asynchronicznych w stosunku do poprzednich wydań .NET, gdzie problem ten był rozwiązywany na poziomie biblioteki TPL – Task Parallel Library.

W dokumentacji do Visual Studio 2012 znajduje się przykład wykorzystania **await** i **async** do ściągania danych ze strony WWW w tle, bez blokowania zdarzeń w oknie głównym. Nasz przykład będzie dotyczył obliczeń, bowiem będziemy w sposób rekurencyjny obliczać n-tą liczbę Fibonacciego. Dla n równego czterdzieści, funkcja rekurencyjna potrzebuje dwie, trzy sekundy, aby wyznaczyć pożądaną wielkość.

Listing 2 przedstawia najważniejsze metody klasy realizującej nasze zadanie. Metoda **ComputeFib** zgodnie z nazwą oblicza n-tą liczbę Fibonacciego w sposób rekurencyjny, i jest to zwykła szeregowa metoda niewykorzystująca równoległych technik programowania. Kolejna metoda to **ComputationTask**. W swoim argumencie przyjmuje numer liczby Fibonacciego, rezultat to naturalnie liczba typu int, ale .NET wymaga, aby taka wielkość została zwrócona jako **Task<int>**, bowiem obliczenia będą realizowane jako zadanie wykonywane w tle.

Uruchomienie zadania następuje po wywołaniu metody **Task.Factory.StartNew( )**, a argumentem jest funkcja lambda wywołująca rekurencyjną metodę **ComputeFib**. W obydwu funkcjach jak widać nie stosuje się jeszcze słowa kluczowego **async** - pojawia się ono dopiero w metodzie **RunTestBTN_Click**. Metoda jest odpowiedzialna za obsługę zdarzenia kliknięcia na przycisk i słowo **async** należy dopisać samodzielnie.

Rysunek 2. Wizualizacja współbieżności w programie obliczającym liczby Fibonacciego w Visual Studio 2012 Ultimate Trial



W metodzie tej kluczowym elementem jest następująca linia

```
r = await ComputationTask(int.Parse(FibN.Text));
```

gdzie wywołujemy metodę **ComputationTask** i zawieszamy dalsze wykonywanie się metody **RunTestBTN_Click** do czasu zakończenia zadania powołanego w **ComputationTask**. Jednakże sterowanie w metodzie **async** jest przekazywane do nadrzędnej metody, czyli do głównej pętli sterującej aplikacji. Powrót do realizacji obsługi kliknięcia na przycisk nastąpi naturalnie po zakończeniu obliczeń.

Jest to jak widać stosowanie słów **async**, **await** oraz zadań powoływanych przez klasę **Task**, pozwala na dość łatwe tworzenie asynchronicznych aplikacji, gdzie długo trwające zadanie nie będzie blokować innych czynność realizowanych przez program.

Listing 2. Obliczenie wartości n-tej liczby Fibonacciego bez blokowania zdarzeń w oknie głównym

```
private int ComputeFib(int n) {
   if (n < 3)
      return 1;
   else {
      int n1 = ComputeFib(n - 1);
      int n2 = ComputeFib(n - 2);
      return n1 + n2;
   }
}

private Task<int> ComputationTask(int n) {
   return Task.Factory.StartNew(() => {
      return ComputeFib(n);
   });
}

private async void RunTestBTN_Click(object sender,
RoutedEventArgs e) {
   int r;

   ResultTextBox.AppendText("Uruchomienie obliczeń\n");
   RunTestBTN.IsEnabled = false;

   r = await ComputationTask(int.Parse(FibN.Text));

   ResultTextBox.AppendText(FibN.Text + " = " +
r.ToString() + "\n");

   ResultTextBox.AppendText("Zakończenie obliczeń\n");
   RunTestBTN.IsEnabled = true;
}
```

NOWOŚCI W F# CZYLI LINQ

Visual Basic oraz C# to główne i najważniejsze języki .NET, ale wprowadzenie F# nie jest bynajmniej złym pomysłem czy też fanaberią, ponieważ w ten sposób, co trzeba przyznać firmie od „Okien", przykłada rękę do zwiększenia popularności języków funkcyjnych, które mają swoje zalety godne szerokiej popularyzacji. Niestety, nie są obecnie tak bardzo popularne jak być powinny.

F# w dwóch pierwszych odsłonach nie posiadał bezpośrednio wsparcia dla technologii LINQ. Zapytania, dzięki elastyczności F#, były niejako symulowane za pomocą dodatkowych bibliotek, jednak nie było bezpośredniego wsparcia dla LINQ.

Do innych nowości w F# należy też lepsze wsparcie do obsługi baz danych, można korzystać z protokołu OData, ułatwiającego nawiązywanie połączenia z bazą danych oraz korzystanie z danych. Możliwie jest też wykorzystywanie plików DBML opisujących strukturę bazy danych, a także korzystanie z danych znajdujących się w modelu bytów danych (ang. *Entity Data Model*). Naturalnie, lepsze wsparcie do obsługi danych jest bezpośrednio związane z LINQ. Jednak znalazło się też kilka pomniejszych nowości związanych z własnościami. Podobnie jak w C# można omijać sekcje **set** oraz **get**, w takim przypadku kompilator wygeneruje samodzielnie podstawowe przypisania. Można też tworzyć ciągi znaków objęte potrójnym cudzysłowem, w takim przypadku wewnątrz ciągu znaków można stosować pojedynczy cudzysłów.

W bibliotece standardowej F# (ang. *F# Core Library*) naturalnie mamy wsparcie dla wyrażeń związanych z zapytaniami, a także obsługę typów **nullable** oraz wsparcie dla miar SI, które dotychczas były obsługiwane przez dodatkowy pakiet o nazwie F# Power Pack.

Ponieważ główna nowość F# to LINQ, to warto im poświęcić nieco więcej miejsca. Analogicznie jak VB oraz C#, zapytania możemy kierować do danych znajdujących się w bazach danych, obiektach oraz do XML.

Realizacja zapytań przedstawia się podobnie jak w przypadku C#, warto to pokazać na przykładzie tablicy z liczbami (a może listy?, albo sekwencji?):

```
open System
let numbers = [ 2; 3; 1; 5; 8; 8; 1; 7; 3; 0 ]
```

Treść zapytania rozpoczyna się od słowa **query**, a następnie za pomocą **for** przeglądamy wszystkie elementy **numbers**, ale wybieramy tylko liczby mniejsze niż pięć:

```
query {
   for n in numbers do
   where (n < 5)
   select n;
} |> Seq.iter(fun n -> printfn "%A" n)
```

O ile pierwsza część zapytania jest bardzo klarowna, to z pewnością wyrażenie: **|> Seq.iter(fun n -> printfn "%A" n)** dla osób nie znających F# jest dość dziwne. Jednakże wynikiem zapytania jest zbiór liczb spełniających nasz warunek, co oznacza, iż możemy też napisać następujące wyrażenie:

```
let r = query { treść zapytania }
```

Czym jest LINQ?

Odpowiedź na tak postawione pytanie jest bardzo prosta. Jest to zintegrowany język zapytań – Language Integrated Query. Zamiast tworzyć oddzielne zapytania w języku SQL i przekazywać je do realizacji jako ciągi znaków, można owe zapytania wyrażać bezpośrednio np. w C#, stosując odpowiednie słowa kluczowe. Nie jest to jednak oddzielny język oparty o SQL, ale naturalna część C#, Visual Basic oraz ostatnio także F#. Popularność i użyteczność LINQ spowodowała, iż do wielu różnych języków, np. Java, PHP, JavaScript, powstały biblioteki naśladujące zapytania LINQ. Ze względu na ograniczania syntaktyczne, zazwyczaj są to jednak odpowiednio przygotowane klasy wykorzystujące mechanizm funkcji lambda. Jedną z takich bibliotek jest pakiet Boolinq, opracowany dla języka C++ z wykorzystaniem wzorców. Można go odszukać pod adresem: http://code.google.com/p/boolinq/.

Odpowiednik pierwszego zapytania, jaki podaliśmy dla języka F#, przedstawia się następująco:

```
int vec[] = {2, 3, 1, 5, 8, 8, 1, 7, 3, 0};
auto dst = boolinq::from(vec)
   .where( [ ]( int n ){ return n < 5; })
   .toVector();
```

Wyświetlenie danych zrealizujemy wykorzystując nową odmianę pętli **for**:

```
for ( auto i : dst )
   std::cout << i << " " ;
```

Przykład ten wymaga zgodności z C++11, ale nowe wydanie Visual Studio 2012 oferuje kompilator C++, który bez kłopotów współpracuje z pakietem Boolinq.

W takim przypadku identyfikator **r** zawiera wyniki zapytania i możemy je wyświetlić wprost, korzystając z **printfn**:

```
printfn "%A" r
```

W rezultacie na ekranie zobaczymy pierwsze elementy sekwencji stanowiącej rezultat zapytania. Aby wyświetlić wszystkie po kolei, możemy zastosować pętlę **for**:

```
for n in r do
   printfn "%A" n
Można to też zrobić inaczej, pisząc takie oto wyrażenie:
r |> Seq.iter(fun n -> printfn "%A" n)
```

Oznacza ono, iż tworzony jest iterator dla sekwencji, który przejrzy wszystkie elementy sekwencji oraz wywoła zdefiniowaną funkcję z argumentem **n**, która za pomocą **printfn** wyświetli poszczególne elementy. Wyrażenie **|>** to tzw. forward-pipe-operator, jego zadaniem jest przekazanie danych z lewej strony operatora do wyrażenia znajdującego się po prawej stronie.

Zapytanie, gdzie wybierzemy niepowtarzalne elementy, sprowadza się do dodania słowa **distinct**:

```
let r2 = query {
   for n in numbers do
   distinct
}
```

Można także dodać **select n**, jednakże nie jest to konieczne.

Zapytania mogą składać się z zapytań podrzędnych, prosty przykład pokaże wszystkie pary z dwóch ciągów liczb, gdzie liczba z pierwszego ciągu jest większa niż z drugiego. Na początek trzeba podać dwa zbiory danych:

```
let nA = [3; 2; 4; 7; 4; 6; 9; 12]
let nB = [1; 2; 9; 7; 3]
Treść zapytania przedstawia się następująco:
let r = query {
   for a in nA do
      for b in nB do
         where (a > b)
         select (a,b)
}
```

Mamy dwie pętle **for**, gdzie sprawdzamy, który element jest większy od drugiego za pomocą klauzuli **where**, jeśli warunek jest prawdziwy, to za pomocą **select** zwracana jest para dwóch liczb spełniająca nasz warunek - pierwsza liczba jest większa od drugiej. Wyświetlenie wyznaczonych par realizujemy w podobny sposób jak poprzednio

```
r |> Seq.iter(fun (n,e) -> printfn "%d jest większe niż
%d" n e)
```

Ostatni przykład korzystania z LINQ do obiektów jest nieco bardziej skomplikowany, będziemy bowiem grupować dane za pomocą odpowiednika wyrażenia **group by** z języka SQL. W przypadku LINQ dla F# stosujemy słowo kluczowe **groupByVal**. Problem ponownie będzie związany z liczbami, chcemy w grupach wyświetlić liczby, które mają taką samą resztę przy dzieleniu przez wybraną przez nas liczbę.

Podobnie jak poprzednio, definiujemy zbiór liczb oraz wartość D, gdzie podajemy, przez jaką liczbę będziemy dzielić:

```
let nums = [5; 4; 2; 16; 7; 10; 15; 12; 8]
let D = 3
```

Samo zapytanie przedstawia się następująco:

```
let r = query {
   for n in nums do
   groupValBy n (n % D) into values
   select (values.Key, values)
}
```

Za pomocą **for** wybieramy kolejne liczby, które przekazywane są do sekcji **groupValBy**, gdzie obliczana jest reszta z dzielenia wyrażeniem **(n % D)**, a następnie para liczba: **n** oraz obliczona reszta jest umieszczana w specjalnej sekwencji par. Powstała struktura przedstawia się następująco:

```
seq [(2, seq [5; 2; 8]); (1, seq [4; 16; 7; 10]); (0, seq [15; 12])]
```

Wyświetlenie tej struktury za pomocą **printfn** można zrealizować np. w następujący sposób:

```
r |> Seq.iter(fun (n,e) -> printfn "Liczby z resztą
%A dzielone przez %A:" n D ; (e |> Seq.iter(fun x ->
printfn"%d" x)))
```

Jest to jedna linia kodu, ale nie będziemy w tym miejscu jej dokładnie omawiać, na ekranie konsoli pokaże się następujący rezultat:

```
Liczby z resztą 2 dzielone przez 3:
5
2
8
Liczby z resztą 1 dzielone przez 3:
4
// oraz pozostałe elementy
```

ZAPYTANIE LINQ W F# DO BAZY DANYCH

Naturalnie zapytania do baz danych wymagają jakiejś bazy oraz systemu danych. Ponieważ po instalacji Visual Studio mamy serwer SQL Express, to warto korzystać z tej bazy danych. Najwygodniej wykorzystać przykładową bazę danych o uczniach, jaką znajdziemy w dokumentacji do .NET i Visual Studio. Łatwo także skorzystać z bazy Northwind udostępnionej poprzez sieć Internet.

Zapytanie o firmy dostępne w bazie Northwind w F# jest podobne jak wcześniejsze zapytanie do tablicy. Wystarczy tylko nakazać użycie **TypeProviders**, aby uzyskać dostęp do protokołu OData, która uprości proces uzyskania połączenia do bazy danych:

```
open Microsoft.FSharp.Data.TypeProviders
```

Uzyskanie połączenia do bazy Northwind to tylko dwie linie kodu:

```
type Northwind = ODataService<"http://services.odata.org/
Northwind/Northwind.svc">
let db = Northwind.GetDataContext()
```

Pierwsza linia to uzyskanie typu reprezentującego bazę, a następnie tworzony jest identyfikator reprezentujący strukturę bazy danych, czyli np. są tam nazwy tabel, zapytań i inne definicje opisujące bazę danych. Samo zapytanie to odczytanie z tabeli **Customers** wszystkich klientów, a uzyskane dane zostaną umieszczone w identyfikatorze **cust**:

```
let q1 = query {
   for cust in db.Customers do
   select cust
}
```

Podobnie jak we wszystkich poprzednich przypadkach forward-pipe-operator oraz **printfn** posłuży do wyświetlenia danych:

```
q1 |> Seq.iter (fun cust -> printfn "Firma: %s" cust.
CompanyName )
```

Łatwo korzysta się też z dodatkowych operatorów, np. wykorzystanie słowa kluczowego LIKE do selekcji danych o uczniach, których nazwisko rozpoczyna się na literą „a", przedstawia się następująco:

```
let q2 = query {
   for s in db.Student do
   where (SqlMethods.Like( s.Name, "[Aa]%") )
   select s
}
```

Należy jednak pamiętać, aby dołączyć odpowiedni pakiet:

```
open System.Data.Linq.SqlClient;
```

do obsługi dodatkowych operatorów SQL min. **LIKE**.

PODSUMOWANIE

Trzeba przyznać, iż nowa edycja 4.5, to naturalnie ewolucja pakietu .NET, która nie jest i nie była planowana jako rewolucja, bowiem nie ma nowych bibliotek o tak dużym znaczeniu jak np. WPF czy WCF. Główną nowością jest oczywiście obsługa Windows 8. Brak nowych technologii to wbrew pozorom bardzo dobry objaw, gdyż otrzymujemy lepsze znane nam narzędzia z nowymi funkcjami.

Łatwiejsza obsługa zadań współbieżnych czy wprowadzenie LINQ do F# i ulepszenie obsługi baz danych to naturalnie istotne innowacje. W przypadku F# pozwala to na szersze wykorzystanie tego języka w realizacji własnych programów. W efekcie F# z pewnością zyska większe grono użytkowników, a jak już o tym wspominaliśmy języki funkcyjne są warte poznania.

Na zakończanie warto podkreślić, iż .NET 4.5 jest jednak znaczącym krokiem w przyszłość, a wyrazem tego jest brak wsparcia Windowsa XP, co niewątpliwie zamyka pewną epokę w historii rozwoju systemu z Redmond.

W sieci:

- Podstawowa strona MSDN dotycząca .NET 4.5:
  http://msdn.microsoft.com/en-us/library/wox726c2%28v=vs.110%29
- Strona technologii .NET:
  http://msdn.microsoft.com/en-us/netframework
- Opis nowości w programowaniu równoległym .NET:
  http://blogs.msdn.com/b/pfxteam/archive/2011/09/17/10212961.aspx
- Sto jeden przykładów zapytań LINQ dla C#:
  http://msdn.microsoft.com/en-us/vstudio/bb688088
- Zbiór przykładów dla języka F# 3.0 zawierający także przykłady zapytań:
  http://fsharp3sample.codeplex.com/
- Dokument MSDN opisujący podstawy LINQ w języku F#:
  http://msdn.microsoft.com/en-us/library/hh225374

Marek Sawerwain redakcja@programistamag.pl

Autor, pracownik naukowy Uniwersytetu Zielonogórskiego, na co dzień zajmuje się teorią kwantowych języków programowania, ale także tworzeniem oprogramowania dla systemów Windows oraz Linux. Zainteresowania: teoria języków programowania oraz dobra literatura.

Michał Mac

C# async i await – asynchroniczność wbudowana w język

Długo oczekiwana funkcjonalność, która rozwiązuje problem, z którym spotkał się każdy developer tworzący aplikację z GUI – blokowanie i zawieszanie się interfejsu użytkownika. Od teraz obsługa kontynuacji operacji asynchronicznych została wbudowane w sam język, co znacząco uprościło obsługę tego typu sytuacji.

JAK TO WSZYSTKO SIĘ ZACZĘŁO?

Zacznijmy od prostego scenariusza. Weźmy aplikacje napisaną np. w WPFie oraz serwis umożliwiający dodawanie kontrahentów i ich adresów stworzony w WCFie. Z aplikacji okienkowej w reakcji na naciśnięcie przycisku chcemy wywołać zdalną metodę. Poniższy kod prezentuje proste synchroniczne podejście do tego problemu.

Listing 1. Synchroniczne wywołanie webservicu

```
StartPending();

var sync = new Service.SampleServiceClient();
CustomerId = sync.AddCustomer(CustomerName.Text);

StopPending();
```

Metody **StartPending** i **StopPending** służą do przyciemniania interfejsu tak by użytkownik widział, że odbywa się właśnie długotrwały proces. Niestety wątek interfejsu, który jest odpowiedzialny za odrysowywanie ekranu, nie jest w stanie odświeżyć zmian, gdyż jest on używany do obsługi wywołania serwisu. Skutkuje to wszystkim nam znanym efektem „zamrożenia" aplikacji na czas trwania operacji. Ponadto warto zauważyć, że takie blokujące operacje w Silverlightcie są całkowicie zabronione. Generator proxy klienckiego nie udostępnia nam nawet synchronicznego API.

AKTUALNE ROZWIĄZANIA

Do tej pory istniało kilka rozwiązań tego problemu, jednak były one trudne, nieczytelne i niewygodne. Pierwsze z nich to użycie znanego od dawna wzorca APM – metod Begin*, End* oraz interfejsu IAsyncResult. Poniżej możemy zobaczyć wykonanie analogicznej operacji do tej z Listingu 1.

Listing 2. Wzorzec APM

```
StartPending();

var apm = new Service.SampleServiceClient();
apm.BeginAddCustomer(CustomerName.Text,
   ar => Dispatcher.Invoke(() =>
      {
         CustomerId = apm.EndAddCustomer(ar);
         StopPending();
      }), null);
```

By rozpocząć operację, wywołujemy metody zaczynające się od Begin*, przekazując jako przedostatni parametr delegat, który zostanie wywołany po wykonaniu operacji. Ów delegat, by uzyskać rezultat, musi wywołać metodę End, na obiekcie proxy. Tutaj bazujemy na implementacji w miejscu, przez co możemy użyć zmiennej apm z zewnętrzengo zakresu – closure (C# generuje klasę wewnętrzną, by wesprzeć taką operację). Jeśli jednak użylibyśmy pełnej metody, to trzeba by było przekazać obiekt proxy poprze ostatni parametr metody **BeginAddCustomer** – asyncState. Kolejną rzeczą, o której trzeba pamiętać, jest synchronizacja wykonania do wątku UI. Jeśli tego nie zrobimy, dostaniemy wyjątek informujący o tym, że nie można modyfikować kontrolek z innego wątku, niż interfejsowy. Rozwiązanie to użycie Dispatchera i przekazanie kolejnej lambdy.

Drugim popularnym sposobem radzenia sobie w takich sytuacjach jest używanie zdarzeń (Listing 3).

Listing 3. Wywołanie asynchroniczne bazujące na zdarzeniach

```
StartPending();

var client = new Service.SampleServiceClient();
client.AddCustomerCompleted += (o, args) =>
   {
      CustomerId = args.Result;
      StopPending();
   };
client.AddCustomerAsync(CustomerName.Text);
```

To rozwiązanie jest dużo prostsze i czytelniejsze niż poprzednie. Ponadto nie trzeba już synchronizować się do UI, gdyż dzieje się to automatycznie. Warto zwrócić uwagę na brak odpięcia zdarzenia, które tutaj nie jest potrzebne, gdyż nasz obiekt nie jest cachowany. Jeśli jednak reużywalibyśmy go między wywołaniami, bądź oknami, to trzeba by odpinać zdarzenie w celu uniknięcia wycieków pamięci. Ponadto należałoby wtedy skorelować zdarzenia z wywołaniami poprzez obiekt stanu. Pozostawmy to jako ćwiczenie dla ciekawych czytelników.

OBSŁUGA BŁĘDÓW

Nasze aktualne rozwiązanie działa już w pełni asynchronicznie, aplikacja jest responsywna, a okno postępu wyświetla się poprawnie. Spróbujmy do powyższej techniki opartej na zdarzeniach (EAP) dodać obsługę błędów. Mogłoby to wyglądać jak na Listingu 4.

Listing 4. EAP - obsługa błedów

```
StartPending();

var client = new Service.SampleServiceClient();
client.AddCustomerCompleted += (o, args) =>
   {
      if (args.Error != null)
         Exception = args.Error;
      else
         CustomerId = args.Result;

         StopPending();
      };
client.AddCustomerAsync(CustomerName.Text);
```

W tym momencie ilość podobnego i powtarzalnego kodu jest już wystarczająco dużo, by spróbować wyekstraktować taką metodę. Wykorzystując koncepcje programowania funkcjonalnego, moglibyśmy użyć kodu z Listingu 5.

Listing 5. Enkapsulacja wywołania EAP

```
private void CallAddCustomer(string name,
   Action<int> continuation,
   Action<Exception> onException = null, Action onFinally = null)
{
   var client = new Service.SampleServiceClient();
   client.AddCustomerCompleted += (o, args) =>
      {
         if (args.Error != null)
         {
            if (onException != null)
            onException(args.Error);
         }
         else
            continuation(args.Result);

         if (onFinally != null)
            onFinally();
      };
   client.AddCustomerAsync(name);
}
```

W tym momencie użycie takiego kodu staje się dużo przyjemniejsze i co ważne czytelniejsze.

Listing 6. Wywołanie zenkapsulowanego algorytmu

```
StartPending();

CallAddCustomer(CustomerName.Text,
   id => { CustomerId = id; },
   ex => { Exception = ex; },
   () => StopPending());
```

WYWOŁANIA ZALEŻNE

Rozszerzmy teraz nasz scenariusz o wywołanie metody dodającej adres. Przyjmuje ona jako parametr wartość zwróconą z metody dodawania kontrahenta. Niestety, gdy spróbujemy zapisać wykonanie takich dwóch operacji nawet przy użyciu naszej fasady, kod będzie wyglądał jak istne spaghetti, gdyż będziemy musieli użyć zagnieżdżonych delegatów (Listing 7).

Listing 7. Wywołanie metod zależnych

```
StartPending();
CallAddCustomer(CustomerName.Text,
   id =>
   {
      CustomerId = id;
      CallAddAddress(id, Address.Text,
         addressId => { AddressId = addressId; },
         ex => { Exception = ex; },
         () => StopPending());
   },
   ex =>
   {
      Exception = ex;
      StopPending();
   });
```

Oczywiście można sobie wyobrazić, co będzie się działo, jeśli tych metod będzie więcej, bądź będziemy chcieli użyć zasobów, które trzeba zwolnić przy użyciu wzorca Disposable – słówko kluczowe using.

ASYNC

W celu uproszczenia procesu obsługi operacji asynchronicznych w Visual Studio 2012 mamy możliwość skorzystania z nowych dobrodziejstw języka C# (w VB.Net jest identyczna funkcjonalność). Otóż kod asynchroniczny adekwatny do kodu z Listingu 2 i 3 jest przedstawiony na Listingu 8. Samo proxy klienckie jest już generowane z użyciem nowych sygnatur wspierających cały wzorzec.

Listing 8. Wywołanie metody z użyciem async i await

```
private async void AddCustomerAsync(
   object sender, RoutedEventArgs e)
{
   StartPending();

   var asyncSerice = new AsyncService.
      SampleServiceClient();
   CustomerId = await asyncSerice.
      AddCustomerAsync(CustomerName.Text);

   StopPending();
}
```

Metoda została opatrzona modyfikatorem **async** oraz każde wywołanie asynchroniczne zostaje poprzedzone słówkiem **await**.

Wersję obsługującą błędy oraz ekran postępu używając wzorca Disposable, można zobaczyć na Listingu 9. Kod strukturalnie niczym nie różni się od kodu synchronicznego, nie wliczając w to użycia tych dwóch słów kluczowych. Wszystkie konstrukcje językowe (try, using) działają tutaj poprawie. Wewnętrznie kompilator rozkłada naszą metodę i buduje z niej maszynę stanów – podobnie jak w przypadku iteratorów (yield return, yield break).

Listing 9. Pełne możliwości async i await

```
private async void AddCustomerAndAddressAsync
   object sender, RoutedEventArgs e)
{
   using (Pending())
   {
      try
      {
         var asyncSerice = new SampleServiceClient();
         CustomerId = await asyncSerice.AddCustomerAsync(
            CustomerName.Text);
         AddressId = await asyncSerice.AddAddressAsync(
            CustomerId.Value, Address.Text);
      }
      catch (FaultException exception)
      {
         Exception = exception;
      }
   }
}
```

Listing 10. Maszyna stanów wygenerowana przez kompilator

```
private void MoveNext()
{
   try
   {
      TaskAwaiter<int> CS$0$0001;
      MainWindow CS$0$0002;
      bool <>t__doFinallyBodies = true;
      switch (this.<>1__state)
      {
         case -3:
            goto Label_0107;

         case 0:
            break;

         default:
            this.<>4__this.StartPending();
            this.<asyncSerice>5__17 = new SampleServiceClient();
            CS$0$0001 = this.<asyncSerice>5__17.
AddCustomerAsync(this.<>4__this.CustomerName.Text).
GetAwaiter();
            if (CS$0$0001.IsCompleted)
            {
               goto Label_00C7;
            }
            CS$0$0002 = this.<>4__this;
```

```
            this.<>t__stack = CS$0$0002;
            this.<>1__state = 0;
            this.<>u__$awaiter18 = CS$0$0001;
            this.<>t__builder.
AwaitUnsafeOnCompleted<TaskAwaiter<int>,
MainWindow.<AddCustomerAsync>d__16>(ref CS$0$0001, ref
this);
            <>t__doFinallyBodies = false;
            return;
        }
        CS$0$0002 = (MainWindow) this.<>t__stack;
        this.<>t__stack = null;
        CS$0$0001 = this.<>u__$awaiter18;
        this.<>u__$awaiter18 = new TaskAwaiter<int>();
        this.<>1__state = -1;
    Label_00C7:
        CS$0$0001 = new TaskAwaiter<int>();
        CS$0$0002.CustomerId = new int?(CS$0$0001.
GetResult());
        this.<>4__this.StopPending();
    }
    catch (Exception <>t__ex)
    {
        this.<>1__state = -2;
        this.<>t__builder.SetException(<>t__ex);
        return;
    }
Label_0107:
    this.<>1__state = -2;
    this.<>t__builder.SetResult();
}
```

ASYNC I AWAIT

Słowo kluczowe async niejako włącza słowo kluczowe await w metodzie. Nie powoduje ono w żaden sposób wykonania metody w wątku z puli. Jeśli uruchomimy taką metodę, wykona się ona po prostu synchronicznie.

W momencie, gdy kompilator natrafi na słowo kluczowe await, to w tym miejscu może wprowadzić asynchroniczność. Await przyjmuje jeden argument, którym jest tzn. typ **awaitable**. Podczas wykonania sprawdzany jest stan obiektu awaitable:

- Jeśli dostępne są już rezultaty, metoda wykonuje się dalej w sposób synchroniczny.
- Jeśli nie, to metoda asynchroniczna jest przerywana, a wywołanie wraca do metody wołającej metodę asynchroniczną – pozwalając jej się dalej wykonywać. Dzięki temu m.in. nasz interfejs się nie zawiesza. W momencie, gdy awaitable zasygnalizuje, że operacja już się zakończyła, wznawiane jest wykonanie metody asynchronicznej od ostatniego miejsca przystanku.

CZYM JEST AWAITABLE?

Awaitable to obiekty, które mogą współpracować z Async. W .NET są to dobrze znane taski: Task oraz Task<T>. Ponadto jest kilka innych miejsc, gdzie możemy spotkać typy awaitable, np. metoda Task.Yield.

Warto zauważyć, że cała infrastruktura została stworzona z myślą o rozszerzalności – możemy tu mówić o wzorcu Async. W zaawansowanych scenariuszach mamy możliwość tworzenia swoich typów awaitowalnych – będą one musiały mieć metodę **GetAwaiter** (bądź extension metodę) i zwracać klasę o specyficznej budowie. Szczegóły implementacji można zobaczyć w przykładach załączonych do artykułu (ramka „W sieci”).

To obiekt definiuje, czy można na nim czekać, czy nie. Nie potrzebujemy zatem metody oznaczać jako async, jeśli metoda zwraca taska, jeśli natomiast sami w metodzie czekamy, to wtedy już tak (Listing 11).

Listing 11. Róże wersje z async i await

```
public async Task AsyncMethod()
{
    await Task.Delay(1000);
}

public Task TaskMethod()
{
    return Task.Delay(1000);
}

public async void Wait()
{
    await AsyncMethod();
    await TaskMethod();
}
```

TYPY I WARTOŚCI ZWRACANE

Jako rezultat dla metody asynchronicznej możemy zdefiniować 3 typy:

- Task<T> - kiedy chcemy zwrócić wartość typu T i mieć możliwość zaczekania na to wyliczenie (Listing 12).

Listing 12. Zwracanie wartości z metody async

```
public async Task<string> GetName()
{
    await CallWebSerivce();

    return "Big Bird";
}
```

- Task - kiedy nie chcemy nic zwrócić, ale chcemy mieć możliwość czekania - awaitowania wykonania tej metody. Warto zauważyć, że jako typ zwracany występuje tu Task, ale nie możemy zwrócić jego instancji (Listing 13).

Listing 13. Metoda, na wykonanie której można zaczekać

```
public async Task GetName()
{
    await CallWebSerivce();
}
```

- void – nie można nic zwrócić, ani na nic zaczekać. Taka postać jest głównie używana, gdy chcemy mieć asynchroniczny EventHandler (Listing 14).

Listing 14. Metoda, na wykonanie której nie można zaczekać

```
public async void DoLongWork()
{
    await CallWebSerivce();
}
```

Inna ciekawą możliwością, o której warto wspomnieć, jest komponowanie awaiterów. Można uruchomić kilka operacji na raz i czekać na skończenie wszystkich, bądź dowolnej z nich (Listing 15).

Listing 15. Kompozycja

```
public async Task<int> GetGrandTotalPrice()
{
    Task<int> t = GetOrderTotalPrice(1);
    Task<int> t2 = GetOrderTotalPrice(2);

    await Task.WhenAll(t, t2);
    return t.Result + t2.Result;
}
```

KONTYNUACJA A WĄTEK

Dociekliwy czytelnik może zapytać, na jakim wątku jest uruchamiane wznowienie metody asynchronicznej. Otóż w aplikacjach UI będzie to wątek interfejsu, w aplikacjach ASP będzie to wątek zapytania, a w każdym innym domyślnie będzie to wątek z puli wątków. Wewnętrznie infrastruktura async korzysta z klasy SynchronizationContext, która odpowiada za uruchomienie kontynuacji przy użyciu odpowiedniego wątku. Każda z powyższych technologii wstrzykuje swoją implementację, która pozwala na takie zachowanie.

Czasami jednak po wykonaniu jakiejś operacji asynchronicznej nie ma potrzeby sychronizowania się do UI, np. wywołujemy webservice, następnie wykonujemy lokalne długotrwałe obliczenia i zapisujemy wynik na dysk. W takiej sytuacji możemy powiedzieć infrastrukturze, by nie korzystała z oryginalnego kontekstu synchronizacji. Jeśli tego nie zrobimy, to nasz interfejs znowu zostanie zamrożony.

Listing 16. Brak synchronizacji do oryginalnego kontekstu

```
public async void DoWork()
{
   await CallWebSerivce().ConfigureAwait(false);

   // Long running local calculation
   // Save to disk, no UI access
}
```

PODSUMOWANIE

Async i await znacząco upraszcza obsługę asynchroniczności w interfejsie użytkownika. Kod staje się prosty, czytelny i co ważne zarządzalny. Jednakże użycie przedstawionego rozszerzenia językowego nie jest oczywiście limitowane tylko do celów interfejsowych. Możemy go także używać do asynchronicznego wczytywanie czy zapisywanie plików oraz do tych wszystkich sytuacji, gdzie dotąd pojawiały się wzorce takie jak APM oraz EAP.

W sieci

- Kod przykładów do artykułu: https://async.codeplex.com
- Seria postów C# pod lupą: http://blog.macmichal.pl/category/blog/c-sharp/

Michał Mac kontakt@macmichal.pl

Jest niezależnym konsultantem, www.macmichal.pl, specjalizującym się w optymalizacji procesu tworzenia oprogramowania. Zawodowo ściśle współpracuje z firmą szkoleniowo-doradczą www.bottega.com.pl, gdzie zajmuje się inżynierią oprogramowania i odpowiada za sekcję .NET i T-SQL. Dzięki całościowemu spojrzeniu na inżynierię oprogramowania korzysta z najlepszych rozwiązać zarówno z .Neta, jak i Javy. Prowadzi blog: http://blog.macmichal.pl

Microsoft Summit Technology - 7. edycja konferencji

Microsoft Technology Summit

29 października 2012 r. odbędzie się 7. edycja jednej z największych w Polsce i Europie Środkowej konferencji technologicznych Microsoft Technology Summit.

Podczas MTS, które odbędzie się w warszawskim Centrum Expo XXI przy ul. Prądzyńskiego, będziecie mogli się zapoznać z najnowszymi trendami w IT i spotkać najbardziej uznanych specjalistów z branży.

W ciągu dwóch dni konferencji zostaną zaprezentowane ostatnie osiągnięcia i nowe produkty firmy Microsoft.

Rejestracja na stronie i szczegółowe informacje: http://www.mtskonferencja.pl.

Na konferencji Microsoft Technology Summit 2012 będzie miał przyjemność poprowadzić sesję na temat zaawansowanych technik DDD również nasz autor Michał Mac. http://blog.macmichal.pl/mts-2012-moja-sesja/

Serdecznie polecamy!

Gynvael Coldwind

Diabeł tkwi w szczegółach: C/C++ (część 2)

Ciąg dalszy rozważań na temat niezdefiniowanych zachowań (ang. Undefined Behavior, dalej UB) i ich potencjalnych skutków, na przykładzie niezamierzonych wycieków danych, oraz kolejnych problemów związanych z operacjami na liczbach całkowitych.

POPULARNE PLATFORMY

Programista tworzący kod nie powinien polegać na danym, często wywnioskowanym na podstawie testów, niezdefiniowanym zachowaniu (UB). Nawet zmiana sposobu kompilacji (np. użytych flag optymalizacyjnych) czy wersji kompilatora może spowodować zmianę danego zachowania, a więc kod przestanie robić to, czego programista od niego oczekuje - świadczą o tym choćby przykłady z poprzedniej części artykułu (Programista 3/2012).

Niemniej jednak w kontekście danej platformy i kompilatora możemy wskazać najbardziej prawdopodobne, rzeczywiste skutki wykonania kodu zawierającego UB - często są one dobrze znane i rozumiane. W niniejszym artykule zostaną wskazane niektóre prawdopodobne skutki UB dla popularnych platform x86-32 oraz x86-64 (AMD64) przy założeniu, że kod został skompilowany kompilatorem z rodziny GCC dla systemów GNU/Linux (Ubuntu) oraz Windows.

WYCIEKI DANYCH

Niejako oczywistym jest, że w pamięci procesu znajdują się wszystkie dane, na których dany proces operuje. Trochę mniej oczywisty jest fakt, że często znajdują się tam również fragmenty niektórych danych, na których proces wcześniej operował. Co więcej, nowo utworzone zmienne mogą zostać umieszczone w tym samym obszarze pamięci, w którym leżą stare dane. To wszystko powoduje, że możliwym jest nieintencjonalne spowodowanie ujawnienia (wycieku) istotnych danych, np. podczas wysyłania pakietów sieciowych lub zapisu danych do pliku.

Dlaczego w ogóle przejmować się wyciekiem jakichś tam „losowych" bajtów z pamięci? W niektórych przypadkach faktycznie dane, które zostaną ujawnione, nie zawierają żadnych wartościowych informacji. Rozważmy jednak, co może znajdować się w pamięci np. przeglądarki internetowej:

- Fragmenty przeglądanych stron internetowych
- Historia przeglądania
- Wartości ciasteczek
- Ostatnio użyte hasła do logowania na poszczególnych stronach
- Ostatnio wpisywane w formularzach dane
- Różne wewnętrzne struktury danych

Z punktu widzenia prywatności i bezpieczeństwa użytkownika nie było by dobrze, aby któraś z powyższych informacji została ujawniona. Dotyczy to również wewnętrznych struktur danych - zawarte w nich informacje (np. wartości wskaźników) mogą stanowić bardzo ważną podpowiedź dla atakującego, który próbuje obejść współczesne mechanizmy zabezpieczeń utrudniające wykorzystanie podatności (w szczególności ASLR [1]).

Czytelnikowi zainteresowanemu konkretnymi przykładami chciałbym wskazać lukę znalezioną przeze mnie kilka lat temu w popularnych przeglądarkach internetowych [V1], podobną lukę znalezioną przez Mateusza Jurczyka [3], oraz nominowaną do Pwnie Awards[1] lukę w Adobe Flash znalezioną przez Fermina Serna [4].

Wracając do tematu artykułu, czyli języków C oraz C++, można wskazać trzy główne powody niezamierzonego wycieku danych:

- Niezainicjowane zmienne (w tym pola w strukturze itp.)
- *Padding* (pl. dopełnienie, bajty wyrównujące; dalej będę korzystał ze spolszczenia „padding")
- Nieprawidłowy dostęp do pamięci
- Rozważmy szczegółowo poszczególne przypadki.

NIEZAINICJOWANE ZMIENNE

Jedną z pierwszych rzeczy, których programista uczy się o C/C++, jest fakt, że nie wszystkie deklarowane zmienne są zainicjowane (tj. mają od początku istnienia ustaloną wartość) [N1570 6.7.9] [N3337 8.5]. Do grupy zainicjowanych można włączyć:

- Zmienne o tzw. statycznym czasie przechowywania (ang. *static storage duration*), a konkretniej zmienne globalne oraz statyczne są inicjowane zerem (lub **`NULL`**, bądź **`nullptr`** w przypadku wskaźników). W przypadku struktur, itp., każdy element inicjowany jest osobno.
- Zmienne lokalne wątku (ang. *thread-local*) są inicjowane zerem (jw.).
- W pełni jawnie zainicjowane zmienne (np. **`int a = 5;`**).
- Częściowo jawnie zainicjowane instancje struktur oraz tablice (np. **`int a[5] = {1,2};`** - pozostałe elementy tablicy **`a`** zostaną wyzerowane).
- Obiekty z konstruktorami inicjującymi wszystkie pola.

Natomiast pozostałe zmienne przy tworzeniu mają nieokreśloną zawartość, tj. nie można czynić żadnych założeń co do ich wartości (w szczególności oznacza to, że kompilatory nie mają obowiązku czyścić pamięci zajmowanej przez takie zmienne). Do tej grupy zaliczamy:

- Zmienne lokalne (np. **`int a;`**).
- Zmienne alokowane na stercie (np. za pomocą **`malloc`** czy **`new`**, oczywiście z wyłączeniem przypadku wywołania konstruktora inicjującego wszystkie pola obiektu).

1 Pwnie Awards są prestiżowymi nagrodami przyznawanymi za wybitne osiągnięcia w dziedzinie bezpieczeństwa IT, rozdawanymi corocznie na konferencji Black Hat w Las Vegas.

O ile teoretycznie wartość niezainicjowanej zmiennej jest nieokreślona, o tyle w praktyce na przykładowych platformach można wymienić kilka potencjalnych możliwych zawartości takiej zmiennej. Przede wszystkim, należy zaznaczyć, że po użyciu ani stos, ani sterta nie są w żaden sposób czyszczone. Dodatkowo, co zostało wspomniane wcześniej, zarezerwowana pamięć dla nowej zmiennej mogła być już wcześniej używana. Tak więc:

- Zarezerwowana pamięć na stosie może zawierać:
  - "Ostatnie" (najnowsze) wartości zmiennych istniejących w tym samym miejscu wcześniej.
  - Argumenty wywoływanych wcześniej funkcji.
  - Adresy powrotu.
  - Zachowane w pewnym momencie wartości rejestrów.
  - Tzw. ciasteczka bezpieczeństwa (ang. *security cookies*, ew. *stack canaries* [5]).

- Zarezerwowana pamięć na stercie może zawierać:
  - "Ostatnie" wartości wcześniej istniejących w tym miejscu zmiennych.
  - Fragmenty wcześniej używanych wewnętrznych struktur sterty.

Jako przykład niezamierzonego wycieku danych ze stosu niech posłuży kod tworzący i wysyłający prosty pakiet sieciowy (patrz Listing 1). Pakiet składa się z pola typu (**int type**) oraz danych - w tym wypadku jest to tablica 256 znaków przeznaczona na imię. W funkcji **SendNamePacket** najpierw tworzona jest lokalna instancja struktury **pn**, a następnie uzupełniany jest typ oraz za pomocą bezpiecznej funkcji **strcpy_s** [6] kopiowane jest podane w parametrze imię do **pn.name**. Następnie pakiet jest wysyłany.

Niestety, jeśli podane imię nie składa się z dokładnie 255 znaków (w przykładzie składa się jedynie z 4 + terminator), to część tablicy **pn.name** jest niezainicjowana, a więc potencjalnie zawiera sporo "starych" informacji ze stosu.

Listing 1. Przykładowy kod ujawniający dane ze stosu

```
...
const int PACKET_TYPE_NAME = 5;
...
struct PacketName {
   int type;
   char name[256];
};
...
bool SendNamePacket(Socket *s, const char *name) {
   PacketName pn;
   pn.type = PACKET_TYPE_NAME;
   if(strcpy_s(pn.name, sizeof(pn.name), name) != 0)
      return false;
   return s->Send(&pn, sizeof(pn));
}
...
SendNamePacket(s, "John");
```

W tym konkretnym przypadku potencjalnym rozwiązaniem byłoby wyzerowanie całej pamięci zajmowanej przez tablicę **pn.name** (np. za pomocą funkcji **memset**, lub np. alokując strukturę za pomocą funkcji **calloc**, która zeruje zaalokowany fragment pamięci). Niestety, w przypadkach bardziej skomplikowanych struktur ustawienie wartości wszystkich pól może nie wystarczyć.

PADDING

Języki C oraz C++ przewidują możliwość wyrównania zmiennych w pamięci, w tym w tablicach oraz strukturach, do konkretnych adresów (zazwyczaj podzielnych przez 2, 4 lub 8) [N1570 6.2.8] [N3337 3.11]. Ma to związek z optymalizacją - dostęp do zmiennych znajdujących się na wyrównanych adresach jest szybszy na niektórych architekturach procesorów (temat ten wykracza poza tematykę artykułu; patrz [7]), oraz z wymaganiami niektórych architektur (np. częstym wymaganiem jest, aby czterobajtowa zmienna znajdowała się na adresie podzielnym przez 4).

Oczywiście, w przypadku konieczności wyrównania tworzy się "wolna przestrzeń" pomiędzy kolejnymi zmiennymi - ta przestrzeń nazywana jest paddingiem.

W C/C++ padding występuje przede wszystkim w następujących miejscach (zaczynając od najbardziej oczywistych):

- Między polami struktury (zazwyczaj w przypadku gdy kolejne pola mają różną wielkość).
- W uniach (nadmiarowe bajty względem używanego pola unii).
- W polach bitowych (nieużywane bity).
- W zmiennych (np. **long double**).

Ani standard C, ani C++ nie gwarantują konkretnych wartości jeśli chodzi o padding, a w szczególności nie gwarantują, że bajty stanowiące padding będą wyzerowane [N1570 6.2.6.1] [N3337 8.5]. Również, standardy nie stawiają żadnych wymagań względem kopiowania paddingu podczas przypisania (tj. kompilator w konkretnej sytuacji decyduje, czy padding zostanie skopiowany).

Listing 2A. Przykładowy niepoprawny kod ujawniający dane (struct, union)

```
// x86-32, gcc, GNU/Linux
struct my_st {
   char ch; // sizeof(my_st.ch) == 1
            // padding: 3 bajty
   int i;   // sizeof(my_st.i)  == 4
};          // sizeof(my_st)    == 8

void func1(Socket *s) {
   my_st *x = new my_st;
   x->ch = 'A';
   x->i  = 0x12345678;
   Send(s, x, sizeof(*x));
   // Zostały wysłane 3 niewyzerowane
   // bajty paddingu.
   delete x;
}

union my_u {
   int i;
   char ch;
}; // sizeof(my_u) == 4

void func2(Socket *s) {
   my_u u;
   u.i = 0x12345678;
   ...
   u.ch = 'A';
   Send(s, &u, sizeof(u));
   // Zostały wysłane 3 bajty
   // zawierające fragment wartości
   // wcześniej użytego pola u.i.
}
```

Ponieważ przypadki paddingu w strukturach oraz uniach są dość oczywiste (przykład umieszczony został na Listingu 2A), rozważmy implementację typu long double [N1570 6.2.5] [N3337 3.9.1] dla popularnej platformy x86-64/GCC.

Na omawianej platformie wielkość typu long double zwracana przez operator **sizeof** wynosi 16 bajtów. W praktyce typ **long double** implementowany jest przez *x86 Extended Precision Format* [8], którego wielkość to 80 bitów, czyli 10 bajtów. Tak więc pozostałe 6 bajtów stanowi padding.

Przykładowy kod ujawniający dane zawarte w paddingu znajduje się na Listingu 2B.

Listing 2B. Przykładowy niepoprawny kod ujawniający dane (long double)

```
struct V3D {
   long double x, y, z;
};

bool CacheHighPrecision(FILE *cache, V3D *v) {
   return fwrite(v, sizeof(*v), 1, cache);
}
```

W tym miejscu warto przeprowadzić eksperyment, polegający na:

1. Wypełnieniu stosu znaną wartością (np. poprzez stworzenie i wypełnienie lokalnej tablicy, a następnie jej uwolnienie).
2. Stworzeniu i zainicjowanie zmiennych typu long double.
3. Wypisaniu ich w formie zakodowanej razem z paddingiem.

Przykładowy kod takiego eksperymentu znajduje się na Listingu 3, natomiast na Rysunku 1 znajduje się przykładowy wynik wykonania takiego kodu (kod został wykonany na systemie Ubuntu 11.10).

Listing 3. Eksperyment z `long double`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void HexDump(void *p) {
   unsigned char *uc = p;
   size_t i;

   printf("v: [");
   for(i = 0; i < 10; i++)
      printf("%.2x", uc[i]);
   printf("], padding: [");

   for(i = 10; i < sizeof(long double); i++)
      printf("%.2x", uc[i]);
   printf("]\n");
}

void FillTheStack() {
   char buf[129] = {0};
   memset(buf, 'A', 128);
   fputs(buf, stderr);
}

void Test()
{
   long double a = 5.0;
   long double b = a + 1.0;

   HexDump(&a);
   HexDump(&b);
}

int main(void) {
   FillTheStack();
   Test();
   return 0;
}
```

W tym konkretnym wypadku wypisanie paddingu obu zmiennych spowodowało ujawnienie pewnych danych. W przypadku zmiennej **a** są to dane, którymi wcześniej był wypełniany stos (0x41 to kod znaku 'A'). Natomiast w paddingu zmiennej **b** pojawiła się inna wartość, która okazała się być fragmentem wspomnianego wcześniej ciasteczka bezpieczeństwa.

ODCZYT ZA BUFOREM

Kolejnym częstym powodem wycieków danych jest próba odczytania z bufora (tablicy, struktury) większej ilości danych, niż się tam znajduje - czyli następują odwołania do bajtów znajdujących się w pamięci bezpośrednio za danym buforem. Oczywiście z punktu widzenia standardu jest to UB, natomiast w praktyce spowoduje ujawnienie danych lub "crash" aplikacji (w przypadku gdy dalsza pamięć po buforze nie jest zamapowana).

```
> gcc -Wall -Wextra -std=c99 sz_ld.c -O3
> ./a.out 2>/dev/null
v: [0000000000000a00140], padding: [414141414141]
v: [0000000000000c00140], padding: [dd73f06900b7]
>
```

Rysunek 1. Wynik eksperymentu z long double

Sytuacja jest analogiczna do przepełnieniem bufora (o czym była już mowa w pierwszej części tego artykułu, a także jest mowa pod koniec niniejszej części), z tą różnicą, że dochodzi do próby odczytu, a nie zapisu, danego fragmentu pamięci.

USE-AFTER-FREE

Innym problemem jest tzw. *use-after-free* (dosłownie: użycie po zwolnieniu), czyli odwołanie się do zmiennej (często obiektu lub instancji struktury), która już nie istnieje - tj. została zwolniona, lub skończył się czas jej życia (np. nastąpiło wyjście z bloku kodu, w którym była zadeklarowana).

Przykładem niech będzie Listing 4 (inspirowany pewnym kodem, który ostatnio dostałem do debuggowania), na którym znajduje się niepoprawnie skonstruowana funkcja **`GetCStr`**. W momencie wywołania funkcji z argumentem **`my_string`** wykonywana jest lokalna kopia tego obiektu. Następnie pobierany jest adres wewnętrznego bufora obiektu zawierającego C-string odpowiadający temu stringowi, po czym następuje wyjście z funkcji, wraz ze zniszczeniem obiektu **`s`** (a więc i bufora, którego adres został pobrany). Niestety, adres tego nieistniejącego już bufora został zwrócony i zapamiętany w zmiennej **`n`**, która jakiś czas później jest użyta jako argument funkcji **`puts`**. Wywołanie **`puts`** w tym wypadku spowoduje próbę odczytu z pamięci z nieistniejacego już bufora, co wg. standardu jest UB [N1570 6.2.4] [N3337 3.7], a w praktyce może doprowadzić albo do ujawnienia danych znajdujących się akurat w miejscu wskazywanym przez **`n`** w pamięci (jeśli w międzyczasie zostały tam zapisane inne informacje, co jest możliwe), albo do „crashu" aplikacji, jeśli dany fragment pamięci został odmapowany.

Potencjalnych poprawnych rozwiązań w przypadku tej funkcji jest kilka. Listing 5 prezentuje jedno z nich, polegający na stworzeniu kopii bufora przed jego zniszczeniem (niestety, jednocześnie wprowadza to wymaganie sprawdzenia, czy alokacja wykonywana przez **`strdup`** się powiodła, oraz zwolnienia zaalokowanej pamięci później; w tym konkretnym wypadku najlepsze byłoby całkowite przeprojektowanie problematycznego fragmentu kodu).

Listing 4. Błąd typu `use-after-free`

```
const char* GetCStr(std::string s) {
   return s.c_str();
}
...
const char *n = GetCStr(my_string);
...
puts(n);
```

Listing 5. Potencjalne rozwiązanie problemu

```
const char* GetCStr(std::string s) {
  return strdup(s.c_str());
}
```

Warto w tym miejscu wspomnieć o terminie *dangling pointer* (dosłownie: „zwisający wskaźnik") [9], który określa wskaźnik wskazujący na już nieistniejący obiekt. Dereferencja dangling pointera prowadzi oczywiście do sytuacji opisanej powyżej, stąd

częstym zaleceniem jest umieszczenie wartości NULL/nullptr we wszystkich wskaźnikach wskazujących na dany obiekt po jego zwolnieniu.

PRZECIWDZIAŁANIE WYCIEKOM

Oprócz tworzenia dobrej jakości kodu (inicjowania zmiennych, pamiętania o paddingu, zerowania wskaźników itp.) warto wskazać również kilka narzędzi, które ułatwiają znalezienie tego typu błędów:

- Valgrind - popularny w środowiskach *nixowych program wykrywający niektóre błędy w zarządzaniu pamięcią.
- AddressSanitizer (ASan) [10] - moduł do LLVM pozwalający wykryć wiele różnych nieprawidłowości w trakcie operowania na pamięci.
- PageHeap lub GFlags - narzędzia umożliwiające włączenie dodatkowych opcji w menadżerze sterty pod Windowsem, ułatwiających detekcję nieprawidłowości związanych z operowaniem na stercie.

Dodatkowo, przy tworzeniu formatów plików i protokołów sieciowych warto zainteresować się istniejącymi bibliotekami do serializacji danych, takimi jak np. Protocol Buffers [11], a także pomyśleć o tekstowych formatach przechowywania danych jak np. JSON czy XML (choć mogą wprowadzić dodatkowe problemy innej natury).

DZIELENIE NA INTACH

Na koniec wróćmy na chwilę do tematu zmiennych typu **int** opisywanych w poprzedniej części artykułu. Warto wskazać jeszcze dwa przypadki, w których wynik, będący konsekwencją zakresu zmiennych oraz ich wewnętrznego kodowania, może być zaskoczeniem dla programisty.

Zacznijmy od dzielenia dwóch liczb całkowitych. Oczywiście, każdy praktykujący programista pamięta o specjalnym przypadku, jakim jest dzielenie przez zero - według standardów jest to UB [N1570 6.5.5, N3337 5.6], natomiast na popularnej platformie x86 skutkuje rzuceniem wyjątku #DE (*Division Error*, pl. błąd [podczas] dzielenia) w terminologii Intela [7], EXCEPTION_INT_DIVIDE_BY_ZERO w terminologii Microsoftu lub SIGFPE z parametrem FPE_INTDIV w przypadku systemów zgodnych z POSIX.

Na tej architekturze, oraz innych korzystających z kodowania U2, jest jeszcze jeden specjalny i niestety mniej znany przypadek:

```
int a = INT_MIN;
a /= -1; // *crash*
```

Przeanalizujmy powyższy kod: zmiennej **a** przypisywana jest wartość **INT_MIN**, czyli, zakładając 32-bity typ **int**, -2147483648 (kodowane jako 0x80000000), następnie wartość ta zostaje podzielona przez -1, o czym można myśleć jako o zmianie znaku - a więc wynikiem jest 2147483648. Niestety, ta liczba wykracza poza zakres zmiennej typu **int** (dla przypomnienia, zakres 32-bitowego typu **int** to -2147483648 do 21474836487), a więc nie da się jej zapisać w zmiennej typu **int** - mamy więc do czynienia z UB [N1570 6.5] [N3337 5]. Na rozważanej platformie zostanie wygenerowany wyjątek, podobnie jak w przypadku dzielenia przez zero (#DE). Listing 6 prezentuje poprawną funkcje dzielącą dla rozważanej platformy.

Listing 6. Poprawna funkcja dzieląca (dla kodowania U2)

```
bool Dzielenie(int *wynik, int dzielna, int dzielnik) {
  if(dzielnik == 0)
    return false;

  // wymaga limits.h
  if(dzielna == INT_MIN &&
     dzielnik == -1)
    return false;

  if(!wynik)
    return false;

  *wynik = dzielna/dzielnik;
  return true;
}
```

ZMIANA ZNAKU

Zmianę znaku można oczywiście zapisać również w sposób bezpośredni, wstawiając znak minus przed zmienną. Rozważmy następujący przykład (dla tych samych założeń co poprzednio):

```
int a = INT_MIN;
a = -a;
```

Analogicznie jak w poprzednim przypadku, wynikiem jest 2147483648, którego nie można zapisać w rozważanej zmiennej. W przeciwieństwie jednak do dzielenia, w tym wypadku nie zostanie wygenerowany wyjątek. Spotykanym zachowaniem będzie natomiast przypisanie zmiennej **a** ponownie wartości **INT_MIN**, co wynika ze sposobu zmiany znaku w kodowaniu U2 (patrz ramka **Zmiana znaku liczby w kodowaniu U2**). Ważną obserwacją jest więc fakt, że dla tej konkretnej wartości nie jest możliwe wyliczenie wartości bezwzględnej, ani "ręcznie", ani za pomocą funkcji **abs()** - w obu przypadkach otrzymany wynik będzie w praktyce nadal ujemny.

Jakie są konsekwencje zaniedbania obsłużenia opisanej sytuacji? Poza oczywistym błędem w wynikach lub "crashem" programu, można również w niektórych sytuacjach doprowadzić do możliwości wykonania kodu przez atakującego. Niech przykładem będzie obsługa (niesławnego) formatu zapisu bitmapowych obrazów - BMP.

W nagłówku BMP znajdują się między innymi dwa 32-bitowe pola ze znakiem (o których możemy myśleć jak o intach), opisujące szerokość oraz wysokość obrazu [2]. Format BMP zakłada, że bitmapa w pliku zapisana jest od dołu do góry, chyba że wysokość jest wartością ujemną - w takim wypadku w pliku bitmapa jest zapisana od góry do dołu, a faktyczna wysokość to wartość bezwzględna z wartości zapisanej w nagłówku.

Listing 7. Niepoprawne wyliczenie ilości potrzebnej pamięci w loaderze BMP

```
struct BMP {
   ...
   int w, h;
   int bpp;
   bool topdown;
   ...
};

void* AllocBitmap(BMP *b) {
  if(b->bpp != 8) {
     // Unsupported.
     return NULL;
  }

  // Top-Down?
  if(b->h < 0) {
     b->topdown = true;
     b->h = abs(b->h);
  }
```

Zmiana znaku liczby w kodowaniu U2

Obliczanie zmiany znaku w kodowaniu U2:
Krok 1: negacja wszystkich bitów
Krok 2: dodanie 1 do wyniku

Przykład dla 32-bitowej liczby 1 (zapis hexadecymalny):
00000001 ← początkowa wartość
FFFFFFFE ← po negacji
FFFFFFFF ← po dodaniu 1
Otrzymana wartość: -1

Specjalny przypadek dla 32-bitowej liczby -2147483648 (INT_MIN):
80000000 ← początkowa wartość
7FFFFFFF ← po negacji
80000000 ← po dodaniu 1
Otrzymana wartość: -2147483648 (INT_MIN)

```
    // Limit.
    if(b->w > 4096 || b->h > 4096) {
       return NULL;
    }

    // Alloc.
    size_t sz = b->w * b->h;
    return malloc(sz);
}
```

Listing 7 zawiera przykładowy, niepoprawny kod wyliczający ilość potrzebnej pamięci na podstawie danych z nagłówka. Programista tworzący ten kod założył, że po wywołaniu **abs()** wysokość będzie zawsze dodatnia, co, jak już wiemy, nie jest prawidłowym założeniem w C i C++. Dla wysokości równej **INT_MIN**, po **abs()** na rozważanej platformie wysokość będzie nadal wynosić **INT_MIN**, a więc będzie nadal bardzo dużą ujemną liczbą. Oczywiście, w tym wypadku test **b->h > 4096** zwróci **false**, przez co skrajnie duża liczba nie zostanie wykryta. Idąc dalej, w mnożeniu **b->w * b->h** nastąpi *integer overflow*, a dla parzystych szerokości dodatkowo wynikiem będzie 0 (np. 4 * 0x80000000 → 0x00000000 dla 32-bitowych zmiennych), co z kolei spowoduje alokację zdecydowanie za małej ilości bajtów na stercie (patrz część pierwsza artykułu).

Prawidłowym rozwiązaniem w tym przypadku byłoby sprawdzenie, czy **b->h** jest równe **INT_MIN** przed wywołaniem **abs()**, oraz wyjście z funkcji zwracając **NULL** w takim przypadku. Istotne jest, aby sprawdzenie nastąpiło faktycznie przed próbą zmiany znaku, ponieważ jak pamiętamy z poprzedniej części artykułu, kompilator może zdecydować usunąć kod opierający się na UB w fazie optymalizacji (a z matematycznego punktu widzenia, sprawdzenie, czy wynik jest ujemny po obliczeniu wartości bezwzględnej jest zupełnie zbędne).

PODSUMOWANIE

Kończąc niniejszą krótką serię artykułów, chciałbym raz jeszcze zachęcić czytelnika do przejrzenia standardów języków C oraz C++, zwracania uwagi na niezdefiniowane zachowania podczas tworzenia kodu, a także do zapoznania się z zaleceniami na temat tworzenia bezpiecznego i stabilnego kodu [12].

W sieci

Najnowsze szkice standardów C i C++:

- [N1570] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
- [N3337] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

Pozostała bibliografia:

- [1] Wikipedia. http://en.wikipedia.org/wiki/Address_space_layout_randomization
- [2] Microsoft. MSDN, „BITMAPINFOHEADER structure". http://msdn.microsoft.com/en-us/library/windows/desktop/dd183376(v=vs.85).aspx
- [3] Mateusz Jurczyk, Hispasec Labs. „Firefox, Opera, Safari for Windows BMP file handling information leak". http://goo.gl/5dDVW
- [4] Fermin Serna. „CVE-2012-0769, the case of the perfect info leak". http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf
- [5] Wikipedia. http://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries
- [6] Microsoft. MSDN, „strcpy_s". http://msdn.microsoft.com/en-us/library/td1esda9.aspx
- [7] Intel. „Intel® 64 and IA-32 Architectures Software Developer Manuals". http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- [8] Wikipedia. http://en.wikipedia.org/wiki/Extended_precision#x86_Extended_Precision_Format
- [9] Wikipedia. http://en.wikipedia.org/wiki/Dangling_pointer
- [10] Address Sanitizer. http://code.google.com/p/address-sanitizer/
- [11] Protocol Buffers. http://code.google.com/p/protobuf/
- [12] CERT. „Secure Coding Standards". https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards
- [13] Microsoft. MSDN, „GFlags and PageHeap ". http://msdn.microsoft.com/en-us/library/windows/hardware/ff549561.aspx

Materiały własne:

- [V1] „Firefox 2.0.0.11 and Opera 9.50 beta Remote Memory Information Leak" http://vexillium.org/?sec-ff

Wszelkie opinie wyrażone w artykule są prywatnymi opiniami autora.

Gynvael Coldwind gynvael@coldwind.pl

Na co dzień autor pracuje w firmie Google na stanowisku Information Security Engineer. Po godzinach prowadzi bloga oraz nagrywa podcasty o programowaniu (http://gynvael.coldwind.pl/). Hobbystycznie programuje od ponad 20 lat (w tym ponad 10 lat w C i C++).

PROGRAMUJESZ?

Musisz go mieć!

Miesięcznik dostępny
w prenumeracie

Prenumerata wydań drukowanych
lub czasopism elektroniczne (ePUB, mobi, PDF)

Zamów już dziś przez

www.programistamag.pl

programista

Cena 12. wydań w prenumeracie „druk": 235 zł brutto z wysyłką + prezenty wg aktualnej oferty.
Cena 12. wydań w prenumeracie elektronicznej to 99 zł brutto.
Magazyn „Programista" dostępny jest też w sprzedaży detalicznej w salonach Empik.

Domain Specific

Marek Mizier

Loose coupling w PHP, czyli co nowego w Symfony2

W poniższym artykule postaram się przybliżyć, czym jest i jakie korzyści niesie ze sobą wstrzykiwanie zależności. Wyjaśnię, w jaki sposób teoria ta jest wykorzystywana w Symfony2. Na koniec przedstawię dobre praktyki i najczęściej spotykane antywzorce.

Język PHP zyskał popularność dzięki temu, że jest bardzo łatwy w użyciu. W minimalistycznej wersji całą funkcjonalność strony zmieścić można w jednym pliku źródłowym. Jednak czasy, kiedy aplikacje faktycznie były budowane w ten sposób, odeszły już dawno w niepamięć. W ciągu ostatnich kilku lat zaobserwować można swoistą metamorfozę PHP. Wraz ze wzrostem złożoności rozwiązań, firmy wykorzystujące tę technologię zmuszone zostały do zastosowania wzorców architektonicznych znanych z innych języków. Pierwsza fala dojrzałych frameworków, która ujrzała światło dzienne w 2007 roku, zapoczątkowała szerokie wykorzystanie wzorca Model View Controller. Jednak aplikacje rozwijały się nadal i szybko okazało się, że samo MVC nie wystarczy. Obecnie, drugie już odsłony Symfony i Zend Framework, stawiają na modularność budowy i łatwą testowalność. Zgodnie z ogólnoświatowymi trendami realizują one paradygmat loose coupling.

TROCHĘ TEORII

Definicja paradygmatu luźnych powiązań (ang. *loose coupling*) jest bardzo prosta. Występuje on wtedy, gdy element systemu nie ma wiedzy o innych elementach, lub wiedza ta jest nikła. Dzięki loose coupling możliwe jest zbudowanie aplikacji z pojedynczych klocków, którymi są poszczególne funkcjonalności.

Najpopularniejszym aktualnie sposobem zastosowania loose coupling w praktyce jest wstrzykiwanie zależności (ang. *dependency injection*, DI). Jest to wzorzec, który przenosi odpowiedzialność za tworzenie obiektów zależnych na zewnątrz obiektu. Innymi słowy klasa, zamiast hardcodować zależności, dostaje je z zewnątrz.

Innym, często spotykanym terminem, jest odwrócenie sterowania (ang. *inversion of controll*, IoC). Polega ono na przeniesieniu odpowiedzialności za wykonanie pewnych czynności poza dany element. IoC jest często utożsamiane z depencdency incjection, lecz jego definicja jest o wiele szersza. W szczególności, DI jest jednym ze sposobów realizacji IoC, lecz IoC może zostać zrealizowane bez wykorzystania DI.

Aby lepiej zrozumieć to, w czym wstrzykiwanie zależności jest lepsze od tworzenia zależności wewnątrz klas, najlepiej posłużyć się przykładem. Przypuśćmy, że mamy mechanizm wysyłający newsletter do wszystkich osób, które subskrybowały go na stronie internetowej. W uproszczeniu kod może wyglądać tak:

```
class Newsletter
{
   private $_mailer;

   public function __construct()
   {
      $this->_mailer = new Mailer();
   }

   public function sendNewsletter($addresses)
   {
      $subject = 'Hello world!';
      $message = 'My first newsletter';

      foreach ($addresses as $address) {
         $this->_mailer->send($address, $subject, $message);
      }
   }
}

class Mailer
{
   public function send($to, $subject, $message)
   {
      //email-sending magic here
   }
}
```

Z architektonicznego punktu widzenia nie jest źle. Mam klasę **Newsletter**, która zajmuje się przygotowaniem treści, oraz klasę **Mailer**, której jedyną odpowiedzialnością jest przesłanie wiadomości. Kod, choć dość prosty, jest obiektowy. Można napisać do niego testy jednostkowe, choć sprawdzenie, czy wiadomość faktycznie została wysłana, może być trudne. Życie jednak niesie codziennie coś nowego. Następnego dnia po wdrożeniu dzwoni szef z informacją, że newsletter ma być wysyłany także za pośrednictwem Jabbera. Konieczna jest modyfikacja mechanizmu. Całe szczęście w repozytorium znalazła się klasa, która realizuje ten protokół. Wygląda ona tak:

```
class XMPPMessenger
{
   public function send($to, $subject, $message)
   {
      //xmpp-sending magic here
   }
}
```

Problem w tym, że klasa dystrybutora jest silnie związana z klasą wysyłki maili. Teoretycznie mógłbym stworzyć nową klasę dystrybutora, dziedzicząc z Newsletter i nadpisując tylko konstruktor. Co jednak, jeżeli będę musiał dodać kolejne sposoby transportu? Tak naprawdę klasa przygotowująca newsletter nie musi wiedzieć, w jaki sposób będzie on wysłany. Zależność ta może zostać przekazana (wstrzyknięta) z zewnątrz:

```
class Newsletter
{
   public function __construct($mailer)
   {
      $this->_mailer = $mailer;
   }

   […]
}
```

Jeszcze lepszym pomysłem jest przygotowanie interfejsu, który będą musiały zaimplementować klasy realizujące wysyłkę, aby mogły zostać wykorzystane przez Newsletter.

```
interface MessageTransportInterface
{
   public function send($to, $subject, $message);
}

class Newsletter
{
   public function __construct(
      MessageTransportInterface $mailer
   )  {
         $this->_mailer = $mailer;
   }

   […]
}

class XMPPMessenger implements MessageTransportInterface
{ […] }

class Mailer implements MessageTransportInterface { […] }
```

Dzięki wprowadzonym zmianom ograniczyliśmy wiedzę, jaką posiada klasa newslettera o metodzie transportu. W tej chwili wymaga ona tylko udostępnienia określonego interfejsu. Tak zrefaktoryzowany kod pozwala na wstrzykiwanie dowolnych sposobów transportu do klasy newslettera bez konieczności wprowadzania w nim jakichkolwiek zmian.

Ogromną zaletą DI jest możliwość użycia przygotowanego kodu w wielu miejscach oraz łatwej wymiany jednej implementacji na inną. Nowe funkcjonalności tworzy się poprzez tworzenie nowych klas, a nie modyfikację istniejących.

Wstrzykiwanie zależności sprawia, że testowanie kodu za pomocą testów jednostkowych staje się łatwiejsze. Zamiast testować funkcjonalność wraz z klasami powiązanymi, możemy wstrzyknąć uproszczone wersje zależności (ang. *mock objects*), które będą zachowywać się w oczekiwany przez nas sposób. W powyższym przykładzie, zamiast sprawdzać, czy mail z newsletterem doszedł, mogę przygotować klasę transportową, która przechowa przekazane jej dane. By mieć pewność, że treść wiadomości jest poprawna, wystarczy pobrać ją z obiektu mocka i porównać z oczekiwaną wartością. Pozwala to sprawniej przetestować warunki brzegowe, ponieważ ułatwione jest zasymulowanie konkretnych zachowań.

Różne sposoby wstrzykiwania zależności

W powyższym przykładzie zależności zostały wstrzyknięte przez konstruktor. Główną zaletą tego typu podejścia jest to, że mamy pewność, że zostaną one przekazane do danego obiektu. Konstruktor łatwo może sprawdzić, czy dostał wszystkie wymagane elementy. W niektórych przypadkach jednak, listy argumentów konstruktorów mogą rozrastać się do sporych rozmiarów, co pogarsza czytelność kodu. Istnieją inne sposoby przekazania zależności do wewnątrz klasy.

Zamiast przez konstruktor klasy mogą zostać wstrzyknięte przez setter. Takie podejście pozwala na uniknięcie problemu dużych konstruktorów. Niestety nie mamy wtedy pewności, czy setter został faktycznie wywołany. Konieczne jest zatem sprawdzanie obecności zależności już po utworzeniu obiektu, np. podczas wywołań metod klasy.

Innym sposobem jest wstrzyknięcie przez interfejs. Polega ono na tym, że klasa potrzebująca zależności musi zaimplementować przygotowany wcześniej interfejs. Obiekty zostają wstrzyknięte za pomocą metod przez niego definiowanych. Niestety dla każdego rodzaju zależności konieczne jest przygotowanie nowego interfejsu, co może znacznie wpłynąć na objętość kodu.

Na samym końcu pozostała do wymienienia metoda wstrzykiwania przez właściwość klasy. Polega ona na przypisywaniu zależności bezpośrednio do zmiennych publicznych. W tym wypadku nie ma jednak żadnej pewności, że to, co otrzymaliśmy, jest tym, czego się spodziewaliśmy. Nie ma też pewności, że wstrzyknięte obiekty nie zostaną podmienione na inne, już podczas wykonywania programu. Zdecydowanie odradza się tego podejścia.

Jaki sposób wybrać?

O ile większość ludzi programujących obiektowo zgadza się co do zalet wstrzykiwania zależności, o tyle nie ma jednego preferowanego sposobu realizacji. Najbardziej logiczne, o ile w danym przypadku jest możliwe, wydaje się połączenie wstrzykiwania przez konstruktor i przez setter. Tego pierwszego używamy dla zależności wymaganych, drugiego dla opcjonalnych. Pozwala to uniknąć nadmiaru argumentów w konstruktorze, lecz pozostawia pewność, że dostajemy wszystkie te obiekty, które są wymagane.

Dependency injection container

Wiemy już mniej więcej, na czym polega wstrzykiwanie zależności. Patrząc na kod przykładu newslettera, dojść można do wniosku, że brakuje elementu, który pozwoli spiąć ze sobą poszczególne elementy układanki. Tutaj przychodzi z pomocą kontener zależności (ang. *dependency injection container*). Jest to klasa, która wie, jakie funkcjonalności istnieją w systemie, jakie posiadają zależności i jak je powołać do życia.

Prosta implementacja kontenera dla przykładu newslettera może wyglądać tak:

```
class MyContainer
{
   public function getMailMessageTransport()
   {
      return new Mailer();
   }

   public function getXMPPMessageTransport()
   {
      return new XMPPMessenger();
   }

   public function getNewsletterViaMail()
   {
      $transport = $this->getMailMessageTransport();
      return new Newsletter($transport);
   }

   public function getNewsletterViaXMPP()
   {
      $transport = $this->getXMPPMessageTransport();
      return new Newsletter($transport);
   }
}
```

Powyższy przykład jest bardzo uproszczony. Zaawansowany kontener powinien poradzić sobie z różnego rodzaju problemami, napotykanymi podczas pisania kodu.

Może okazać się, że do utworzenia klasy transportu mogą być potrzebne jakieś dodatkowe parametry, np. login i hasło do konta pocztowego. O nich kontener też powinien wiedzieć. W przypadku klas, które nie są stanowe, nie jest konieczne podnoszenie nowej instancji przy każdym wywołaniu metody **get**. Z punktu widzenia wydajności, znacznie korzystniejsze jest, aby kontener zwracał zawsze dokładnie tą samą instancję klasy.

Ręczne definiowanie klasy kontenera może być uciążliwe. O wiele wygodniejsza jest konfiguracja za pomocą pliku konfiguracyjnego, albo automatycznie na podstawie type-hintingu, czy adnotacji (ang. *annotation*). Wygodny w użyciu kontener powinien udostępniać różne sposoby definiowania zależności.

W sieci dostępnych jest wiele implementacji kontenerów o różnym stopniu skomplikowania. W dalszej części artykułu skupię się jednak na tym, co oferuje Symfony2.

Dependency injection != dependency injection container

Zanim przejdę do konkretów, ważne jest zrozumienie jeszcze jednej kwestii. Często dependency injection jest utożsamiane z dependency injection containerem. Są to jednak dwie zupełnie różne rzeczy. DI może być realizowane „ręcznie", gdy programista sam składa zależności np. w fabryce lub za pomocą kontenera. Z drugiej strony kontener może służyć jako klasa wstrzykująca (ang. *injector*), który wstrzykuje tylko wybrane zależności, lub jako service locator. Drugi ze sposobów wykorzystania uważany jest przez niektórych jako antywzorzec DI. Polega on na wstrzyknięciu klasy, która wie, jak powoływać do życia inne klasy. Klasa korzystająca z locatora sama pobiera z niego potrzebne zależności. Wprawdzie uzyskany efekt jest podobny, lecz wykorzystując service locator, wiążemy się z konkretną implementacją kontenera. Jeżeli w przyszłości będziemy chcieli wykorzystać daną funkcjonalność w innej aplikacji, konieczne będzie także użycie tego samego (lub bardzo podobnego) kontenera. Utrudnione jest też testowanie, bo dokładnie nie wiadomo, jakie obiekty mogą zostać pobrane.

SYMFONY2

Dependency injection container jest sercem Symfony2. Autorzy drugiej wersji tego popularnego frameworka wzięli sobie za cel umożliwienie tworzenia aplikacji w architekturze SOA (ang. *Service Oriented Architecture*). Aby wyjaśnić, w jaki sposób jest to realizowane, konieczne jest zrozumienie, w jaki sposób przetwarzane jest zapytanie o stronę internetową.

Kiedy użytkownik przechodzi na adres strony, jako pierwszy zostaje wywołany front controller. Ma on za zadanie uruchomienie silnika aplikacji. Silnik ten ustala, która klasa ma zająć się dalszym przetwarzaniem zapytania. Klasy te, nazywane kontrolerami, przypisywane są do poszczególnych adresów URL, za pomocą routingu. Mechanizm routingu potrafi określić kontroler, a konkretnie metodę kontrolera, która powinna zostać wywołana. W kolejnym kroku silnik powołuje do życia instancję kontrolera i wykonuje na nim żądaną akcję. Metoda akcji jest elementem, który zawiera logikę biznesową stworzoną na potrzeby konkretnej funkcjonalności aplikacji. Zwraca ona obiekt Response, reprezentujący odpowiedź serwera. Obiekt ten może zostać utworzony ręcznie lub za pośrednictwem mechanizmu szablonów. Domyślnie do obsługi szablonów wykorzystywany jest silnik Twig.

Funkcję kontrolera pełnić może dowolna klasa, ale jeżeli programista zdecyduje się dziedziczyć po abstrakcyjnym kontrolerze Symfony2, kontener zależności zostanie automatycznie wstrzyknięty do kontrolera przez framework. Kontener jest w tym miejscu wykorzystany jako service locator. Kontener jest domyślnie tworzony z pewną liczbą standardowych funkcjonalności, jak np. mechanizm walidacji, system szablonów czy logger. Programista może dołączyć do niego swoje własne usługi. Dobrą praktyką jest to, aby poszczególne funkcjonalności zostały przeniesione do osobnych klas, przetestowane za pomocą testów jednostkowych oraz zarejestrowane w kontenerze. Sam kontroler jest wtedy jedynie elementem, który skleja ze sobą użycie zdefiniowanych usług, a następnie przygotowuje informacje zwrotne, wysyłane użytkownikowi.

Kontrolerem może być również usługa zdefiniowana w kontenerze. Daje to możliwość uniknięcia wykorzystania kontrolera jako service locatora, przez wstrzyknięcie konkretnych zależności zamiast całego kontenera.

Jak używać kontenera?

Dependency incjection container w Symfony2 może zostać skonfigurowany na kilka sposobów. Wszystkie standardowe usługi frameworka zdefiniowane są w plikach *xml*. Ze względu na łatwość użycia i przejrzystość, bardzo popularne jest wykorzystanie formatu yaml. Możliwe jest także zbudowanie kontenera bezpośrednio z kodu PHP, za pośrednictwem obiektu buildera. Ze względu na ograniczoną ilość miejsca, w poniższych przykładach przedstawię tylko notację yaml. Przykłady użycia pozostałych sposobów konfiguracji znaleźć można w dokumentacji Symfony2.

Przygotowanie definicji usługi kontenera jest bardzo proste. Wystarczy w pliku konfiguracyjnym jako element tablicy services podać nazwę usługi i nazwę klasy, która odpowiada za jej realizację:

```
services:
   my_hello_service:
      class: Acme\MyHelloServiceClass
```

Kontener zależności w Symfony2 wspiera wstrzykiwanie za pomocą konstruktora, settera oraz właściwości klasy.

```
services:
   my_mailer:
      class: Acme\Mailer

   my_logger:
      class: Acme\Logger

   my_templating:
      class: Acme\Templating

   my_hello_service:
      class: Acme\MySecondClass

      # wstrzyknięcie przez konstruktor
      arguments:
         - @my_mailer

      # wstrzyknięcie przez setLogger
      calls:
         - [ setLogger, [@my_logger] ]

      # wstrzyknięcie przez właściwość templating
      properties:
         templating: @my_templating
```

Zależności wstrzykiwane do obiektów mogą być stałymi, parametrami konfiguracyjnymi lub innymi usługami. Nazwy parametrów muszą znajdować się pomiędzy parą znaków %, natomiast nazwy usług muszą zostać poprzedzone znakiem @.

```
parameters:
   my_parameter: ”random string”

services:
   my_service:
      class: Acme\RandomClass
      arguments: [ %my_parameter%, @my_other_service ]
```

Czasami do stworzenia obiektu potrzebna jest bardziej skomplikowana logika. W takim wypadku, definiując usługę, można użyć stworzonej wcześniej fabryki. Wymagane jest podanie klasy fabryki oraz metody, która zostanie użyta. Jeżeli klasa jest już zarejestrowana w kontenerze, możliwe jest odwołanie się do niej, jak do usługi.

```
services:
    my_mailer:
        class: Acme\MyMailer
        # użycie klasy fabryki
        factory_class: Acme\MyMailerFactory
        factory_method: getInstance # metoda fabryki
    my_logger_factory:
        class: Acme\MyLoggerFactory
    my_logger:
        class: Acme\MyLogger
        # użycie usługi fabryki
        factory_service: my_logger_factory
        factory_method: get
```

W klasycznym kontrolerze, dziedziczącym po abstrakcyjnej klasie kontrolera dostarczonej przez framework, pobranie usługi z kontenera sprowadza się do wykonania metody get() z parametrem będącym nazwą usługi.

```
$myService = $this->get('my_service');
```

Możliwe jest także określenie zakresu życia usługi. Domyślnie klasy zwracane przez kontener są singletonami. Zmieniając wartość parametru scope, można ograniczyć ich zakres do podzapytania lub sprawić, że za każdym razem zwracana będzie nowa instancja.

Mechanizm kontenera zależności w Symfony2 ma jeszcze wiele możliwości i opcji konfiguracyjnych. Niestety nie sposób opisać tu wszystkich. Do tych najbardziej interesujących należy z pewnością mechanizm tagów, który wykorzystywany jest między innymi podczas obsługi zdarzeń tworzonych przez silnik aplikacji. Bardzo ciekawym sposobem automatycznej konfiguracji usług jest CompilerPass, który pozwala na modyfikację elementów kontenera, które zostały oznaczone wybranym tagiem. Zainteresowanych odsyłam do obszernej dokumentacji na stronie frameworka.

Wydajność

Jak wiadomo w php nie przechowuje się obiektów w pamięci pomiędzy poszczególnymi wywołaniami. W związku z tym konieczne jest budowanie całego środowiska od nowa, po każdym odpytaniu serwera. Parsowanie plików konfiguracyjnych lub używanie obiektu buildera jest dość kosztowne, szczególnie dla aplikacji obsługujących duży ruch. Aby usprawnić ten proces, stworzono mechanizm kompilacji kontenera zależności. Funkcjonalność ta pozwala na wygenerowanie kontenera w sztywnej, docelowej postaci. Uzyskana klasa jest zapisywana w cache'u i wykorzystywana w kolejnych wywołaniach skryptu.

Symfony2 DI jako niezależny komponent

Symfony2 ma modularną budowę, a jego elementy nie są od siebie silnie uzależnione. Jeżeli konkretny przypadek nie wymaga użycia całego frameworka, można wykorzystać poszczególne mechanizmy i włączyć je do własnej aplikacji. Aby to umożliwić, autorzy obok jednego, zbiorczego repozytorium, udostępnili osobne repozytoria z pojedynczymi funkcjonalnościami. Kontener zależności znajduje się w pakiecie o nazwie symfony/DependencyInjection. Informacje o nim można znaleźć na stronie http://symfony.com/components

JAK TO ROBIĄ INNI?

Jak już wspomniałem, społeczność php lubi czerpać sprawdzone wzorce z innych języków. *Dependency injection* w Symfony2 bazuje na rozwiązaniu przyjętym w javowym frameworku Spring. Istnieją oczywiście pewne różnice, z których najciekawsze postaram się tutaj wymienić.

Adnotacje w Spring Framework

Frameworki javowe najczęściej kojarzą się z ogromnymi plikami konfiguracyjnymi, zapisanymi w formacie *xml* i służącymi do definiowania sposobu działania aplikacji. Coraz częściej słyszy się jednak o podejściu *convention over configuration*, które zakłada, że część (lub całość) konfiguracji może odbywać się za pomocą konwencji nazewnictwa i umiejscowienia plików źródłowych. Jednym ze sposobów na pozbycie się obszernych plików konfiguracyjnych jest użycie adnotacji. Z tego właśnie rozwiązania korzysta Spring. Klasy, które są kandydatami do wstrzyknięcia, oznaczane są adnotacją @Service. Jeżeli metoda zostanie oznaczona jako @Autowired, framework próbuje automatycznie dopasować wymagane obiekty na podstawie typów przyjmowanych argumentów. Programista może jednak pomóc mechanizmowi, dodając odpowiednie atrybuty i metaadnotacje.

Type-hinting i refleksje w Zend Framework

Kontener zależności w Zend Framework obsługuje kilka sposobów definiowania usług. Domyślny, o nazwie RuntimeDefinition, używa type-hintingu oraz mechanizmu refleksji do ustalenia, który obiekt powinien zostać wstrzyknięty. Metoda ta działa zarówno w przypadku wstrzykiwania przez konstruktor, jak i przez setter. Podejście to nie jest może zbyt wydajne, ale może zostać połączone z innymi definicjami i ostatecznie skompilowane do statycznej definicji kontenera.

DOBRE PRAKTYKI

Czasy, kiedy teoretycy spierali się, czy loose coupling, inversion of control i dependency injection są lepsze, niż podejście tradycyjne, dawno już odeszły. Dziś wszyscy zgadzają się, że IoC jest dobrym pomysłem, a dyskusje toczą się na temat jego realizacji. Dependency injection jest przez wielu uważane jako najlepsze z dostępnych rozwiązań i jest dobrą praktyką samą w sobie. Istnieją jednak sposoby, aby je usprawnić.

Programowanie do interfejsu (ang. *Interface-based programming*) szeroko rozpowszechnione jest wśród programistów Javy. Przed stworzeniem konkretnej implementacji wstrzykiwanej zależności, tworzymy interfejs, który będzie ją reprezentował. Powinien on składać się z minimalnej liczby metod, które są wymagane w ramach udostępnianej funkcjonalności. Jeżeli implementacja już istnieje, przyjmujemy interfejs zgodny z nią. W miejscach, gdzie tak zdefiniowana zależność jest używana, posługujemy się tylko interfejsem. Dzięki temu możliwa jest łatwa zamiana implementacji bez konieczności zmiany istniejącego kodu.

Inną dobrą praktyką jest używanie adnotacji. Pozwalają one na uniknięcie definiowania zależności poza ciałem klasy, co poprawia czytelność kodu. Dodatkową zaletą jest zmniejszenie rozmiaru plików konfiguracyjnych. Niestety, aby obsługa adnotacji była dostępna w Symfony2, należy zainstalować pakiet dostarczany przez zewnętrzną firmę. Pozostaje mieć nadzieję, że taka funkcjonalność zostanie dodana w kolejnych wersjach frameworka.

ANTYWZORCE

Złe zrozumienie koncepcji DI może prowadzić do błędnych implementacji. Jednym z najczęściej spotykanych antywzorców jest constructor overinjection. Występuje on wtedy, gdy przez

konstruktor wstrzykiwane są nadmiarowe zależności. Jeżeli wykonanie pewnych operacji jest determinowane przez wartość wyrażenia logicznego, mogą istnieć przebiegi, które pomijają te operacje. Zależności, które związane są tylko z opcjonalnymi fragmentami kodu, mogą zostać wstrzyknięte niepotrzebnie, ponieważ nigdy nie zostaną wykorzystane. Podejście to może spowodować także rozrastanie się list argumentów konstruktorów do sporych rozmiarów.

Rozwiązań tego problemu jest kilka. Po pierwsze, zależności opcjonalne, a w szczególności te związane z konkretnym środowiskiem uruchomieniowym, można wstrzykiwać przez setter, co spowoduje skrócenie konstruktorów.

Aby zmniejszyć ilość zużywanej pamięci, warto skonfigurować obiekt kontenera tak, aby zwracał zawsze tę samą instancję obiektu. Nie zawsze jest to możliwe, ale większość funkcjonalności można napisać tak, żeby nie przechowywały stanu.

Opcjonalne zależności, dla których podniesienie instancji obiektu jest złożone obliczeniowo, lub wymaga użycia limitowanych zasobów (np. połączeń z bazą danych), mogą zostać opakowane w inną klasę, która powiela interfejs i wykorzystuje lazy-loading. Do docelowego obiektu wstrzykiwana jest klasa opakowująca, posiadająca zmienną reprezentującą prawdziwą zależność. Zmienna ta jest inicjowana obiektem dopiero w momencie wywołania jednej ze zdefiniowanych metod.

Użycie kontenera jako service locatora przez wielu programistów uważane jest także jako antywzorzec. W zasadzie pozwala on na osiągnięcie tego samego efektu, co bezpośrednie wstrzykiwanie zależności, lecz wiąże implementacje funkcjonalności z konkretną implementacją kontenera. Z pozoru wydaje się, że aktywne pobieranie klas jest wydajniejsze, bo nic nie jest tworzone nadmiarowo. Jednak większość problemów wydajnościowych w przypadku DI udaje się rozwiązać, stosując się do wytycznych przedstawionych powyżej.

Service locator utrudnia testowanie mechanizmów za pomocą testów jednostkowych, ponieważ konieczne jest przygotowanie obiektu mock samego kontenera. Nie do końca jasne jest także to, jakie zależności faktycznie są wymagane.

Zarówno Symfony2, jak i Zend Framework 2 wykorzystują ten wzorzec w kontrolerach. Jednak, jak twierdzi Fabien Potencier (jeden z twórców Symfony), w większości przypadków kontener nie powinien być przekazywany poza klasy kontrolerów.

PODSUMOWANIE

Loose coupling niewątpliwie posiada wiele zalet. Należą do nich reużywalność kodu, łatwe testowanie i modyfikowanie funkcjonalności poprzez podmianę zależności. Do minusów można zaliczyć pogorszenie czytelności przepływów pomiędzy funkcjonalnościami. Czy warto budować aplikację w oparciu o ten wzorzec, pozostaje decyzją, którą każdy musi podjąć samodzielnie. Niemniej jednak idea ta zyskuje coraz więcej zwolenników, niezależnie od języka programowania. Miejmy nadzieję, że na stałe zagości w świecie PHP.

Uwagi:

Powyższy tekst powstał w oparciu o wersję Symfony 2.0, Zend Framework 2.0.0rc5 oraz Spring Framework 3.0.

W sieci

- http://en.wikipedia.org/wiki/Dependency_injection
- http://en.wikipedia.org/wiki/Loose_coupling
- http://www.martinfowler.com/articles/injection.html
- http://ralphschindler.com/2011/05/18/learning-about-dependency-injection-and-php
- http://symfony.com/doc/current/components/dependency_injection/index.html
- http://symfony2tips.blogspot.com/2011/02/dependencyinjection-for-controllers.html
- http://fabien.potencier.org/article/12/do-you-need-a-dependency-injection-container
- http://devzone.zend.com/1775/quick-start-symfony-di-dependency-injection-tutorial/
- http://framework.zend.com/wiki/display/ZFDEV2/Zend+DI+QuickStart
- http://mwop.net/blog/260-Dependency-Injection-An-analogy.html
- http://blog.ploeh.dk/2010/01/20/RebuttalConstructorOverinjectionAntipattern.aspx
- http://www.loosecouplings.com/2011/01/dependency-injection-using-di-container.html
- http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html
- http://www.vogella.com/articles/SpringDependencyInjection/article.html

Marek Mizier

Autor programuje w języku PHP od dziesięciu lat. Obecnie zajmuje się rozwojem platformy transakcyjnej Allegro. W swojej pracy wykorzystuje framework Symfony2 do budowania aplikacji w architekturze SOA.

SZCZEGÓŁOWY RAPORT

POLSKIEGO RYNKU **INFORMATYCZNEGO I TELEKOMUNIKACYJNEGO**

- Ponad 160 stron o branży ICT
- Kto jest prawdziwym liderem polskiego rynku?
- Kto utrzymał pozycję, a kto stracił?
- Tabele porównujące pozycje firm
- Rankingi dostawców rozwiązań i usług w najważniejszych sektorach polskiej gospodarki
- Kierunki rozwoju technologii IT

Cena: 100 zł

ZAMÓW

www.top200.computerworld.pl tel. (+4822) 321-77-77 fax (+4822) 321-78-88

Łukasz Klejnberg

Windows Phone 7.5 – XNA Game Studio 4.0

Sposób na XML. Poznaj dwie sprawdzone metody parsowania dokumentów XML

Gdy pisałem swoją pierwszą grę na urządzenia z systemem Windows Phone 7.5, spotkałem się z problemem szybkiego wczytywania plików XML w XNA Game Studio 4.0. Istnieje bardzo dobra metoda, dzięki której wczytana zawartość od razu jest listą obiektów określonej klasy. Pokażę Ci również możliwość parsowania pobranych dokumentów XML z Internetu – przekonaj się, jakie to jest proste.

TWORZYMY PROJEKT GRY

W Microsoft Visual Studio 2010 Express dla Windows Phone tworzymy nowy projekt gry Windows Phone Game (4.0). Następnie do zasobu WindowsPhoneGameContent dodajemy plik *Base.xml* o zawartości jak na Listingu 1.

Listing 1. Zawartość pliku XML

```
<?xml version="1.0" encoding="utf-8" ?>
<XnaContent>
   <Asset Type="SharedContent.PossibleMoves">
      <moves>
         <Item>
            <name>robot 1</name>
            <x>50</x>
            <y>50</y>
         </Item>
         <Item>
            <name>robot 2</name>
            <x>200</x>
            <y>50</y>
         </Item>
         <Item>
            <name>robot 3</name>
            <x>350</x>
            <y>50</y>
         </Item>
         <Item>
            <name>robot 4</name>
            <x>500</x>
            <y>50</y>
         </Item>
         <Item>
            <name>robot 5</name>
            <x>500</x>
            <y>150</y>
         </Item>
         <Item>
            <name>robot 6</name>
            <x>350</x>
            <y>150</y>
         </Item>
         <Item>
            <name>robot 7</name>
            <x>200</x>
            <y>150</y>
         </Item>
         <Item>
            <name>robot 8</name>
            <x>50</x>
            <y>150</y>
         </Item>
      </moves>
   </Asset>
</XnaContent>
```

Następnie dodajemy nowy projekt "Windows Phone Game Library (4.0)" o nazwie **SharedContent**. Zmieniamy nazwę pliku **Class1.cs** na **PossibleMoves.cs**, jednocześnie zmieniając nazwę klasy **Class1** na **PossibleMoves**. Następnie dodajemy nową publiczną klasę o nazwie **Move**.

W klasie **PossibleMoves** dodajemy listę obiektów klasy **Move** jak poniżej:

```
public List<Move> moves;
```

Następnie uzupełniamy klasę Move jak na Listingu 2.

Listing 2. Klasa Move

```
public class Move
{
   public string name { get; set; }
   public int x { get; set; }
   public int y { get; set; }
}
```

W projekcie **WindowsPhoneGame** oraz **WindowsPhoneGameContent** dodajemy referencję do projektu **SharedContent** – klikamy *Add reference* na zasobie *References* i z zakładki *Projects* wybieramy właściwy projekt.

Jak użyć takiego zasobu? Bardzo prosto i niezwykle szybko. Wystarczy dodać w klasie **Game1** deklarację pola **PossibleMoves** o nazwie **possibleMoves**, może być prywatna. Aby klasa **PossibleMoves** była widoczna w **Game1**, należy dodać **using SharedContent**. W metodzie **LoadContent()** ładujemy nasz xml i wypisujemy przy pomocy **Debug.WriteLine()** jej zawartość jak na Listingu 3. Do tego celu użyjemy cudownej metody **Load<T>** z klasy **ContentManager** frameworka XNA. Należy również dodać w nagłówku klasy **Game1 using System.Diagnostics;**

Listing 3. Zawartość metody LoadContent()

```
protected override void LoadContent()
{
   spriteBatch = new SpriteBatch(GraphicsDevice);
   possibleMoves = this.Content.
Load<PossibleMoves>("Base");
   foreach(Move m in possibleMoves.moves)
   {
      Debug.WriteLine("name: {0}, x: {1}, y: {2}",
m.name, m.x, m.y);
   }
}
```

Przy okazji tą metodą możesz wczytywać takie obiekty jak **Texture2D**, **SoundEffect**, **SpriteFont** itp.

Uruchamiamy projekt w trybie Debug i obserwujemy wynik w oknie output. Jeśli nie widzisz okna output, możesz je dodać poprzez menu główne Visual Studio, czyli Debug/Windows/Output. Rezultat operacji **Debug.WriteLine()** powinien być taki sam jak poniżej:

```
name: robot 1, x: 50, y: 50
name: robot 2, x: 200, y: 50
name: robot 3, x: 350, y: 50
name: robot 4, x: 500, y: 50
name: robot 5, x: 500, y: 150
name: robot 6, x: 350, y: 150
name: robot 7, x: 200, y: 150
name: robot 8, x: 50, y: 150
```

Jest również inny sposób parsowania dokumentów xml. Przedstawię go na innym przykładzie. Będziemy pobierać poprzez zapytanie restowe dokument xml z serwera. W tym celu do naszego projektu musimy zainstalować pakiet **restsharp**. W tym celu przechodzimy do Tools/Library Package Manager/Package Manager Console i w oknie wpisujemy **Install-Package restsharp**. Jak poniżej:

```
PM> Install-Package restsharp
Successfully installed 'RestSharp 103.4'.
Successfully added 'RestSharp 103.4' to WindowsPhoneGame.
```

Po instalacji domyślnie dodawana jest referencja RestSharp, jednakże nie zadziała ona w projekcie dla Windows Phone, dlatego też należy ją usunąć i dodać inną referencję przeznaczoną dla Windows Phone 7.1. Dodajemy ją podobnie jak w przypadku projektu SharedContent, jednakże wybieramy zakładkę Browser, a następnie packages/RestSharp.103.4/lib/sl4-wp71/RestSharp.WindowsPhone.dll. W nagłówku klasy **Game1** dodajemy:

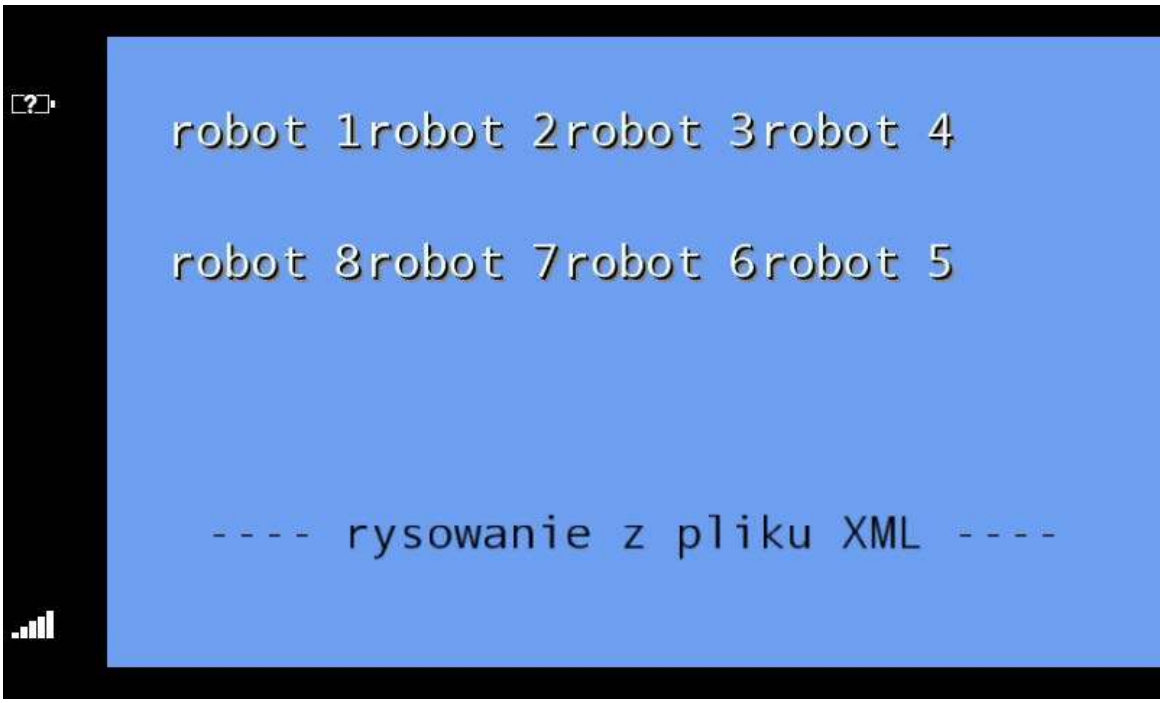
```
using RestSharp;
using using System.Xml.Linq;
```

Dla testów stwórzmy sobie nową prywatną metodę o nazwie **GetRequest()**. Jej wywołanie dodajemy na końcu metody **LoadContent()**, czyli **this.GetRequest();**

Zakładamy, że pod adresem http://xyz3456789xyz.pl/xml.php znajduje się dokument xml. Może być ten sam co w poprzednim przykładzie, więc musimy stworzyć podstawowy request poprzez użycie metody **RestClient()** i **RestRequest()**. Następnie wykonamy request i asynchronicznie pobierzemy wynik. Na końcu wyświetlimy wynik tak samo jak poprzednio poprzez użycie **Debug.WriteLine()**. Uzupełniamy metodę **GetRequest()** tak jak na Listingu 4.

Listing 4. Metoda GetRequest()

```
private void GetRequest()
{
   var client = new RestClient("http://xyz3456789xyz.pl/xml.php");
   var request = new RestRequest();
      client.ExecuteAsync(request, (response) =>
      {
         var content = response.Content.ToString();
         XDocument doc = XDocument.Parse(content);
         var moves = doc.Descendants("Item").Select(Item => new
      {
         name = Item.Element("name").Value,
         x = Item.Element("x").Value,
         y = Item.Element("y").Value
      }).ToList();
```



Rysunek 1. Wynik działania aplikacji

```
    foreach (var m in moves)
        {
        Debug.WriteLine("name: {0}, x: {1}, y: {2}",
m.name, m.x, m.y);
        }
    });
}
```

W AKCJI..

Spróbujemy narysować dane z pliku *XML*. W tym celu dodajemy do zasobu spritefont, następnie w metodzie LoadContent() ładujemy spritefont do pola gameFont jak poniżej

```
gameFont = this.Content.Load<SpriteFont>("gamefont");
```

Rozszerzamy metodę Draw(GameTime gameTime) jak na Listingu 5. Po uruchomieniu aplikacji wynik powinien być podobny jak na Rysunku 1.

Listing 5. Metoda Draw

```
protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);

        spriteBatch.Begin();
        DrawPossibleMoves();
        DrawTitle();
        spriteBatch.End();

        base.Draw(gameTime);
    }

    private void DrawTitle()
    {
        string str = "---- rysowanie z pliku XML ----";
        Viewport viewport = GraphicsDevice.Viewport;
        Vector2 vStr = gameFont.MeasureString(str) / 2;
        vStr.X = viewport.Width / 2 - vStr.X;
        vStr.Y = viewport.Height - vStr.Y * 5;
        spriteBatch.DrawString(gameFont, str, vStr, Color.Black);
    }

    private void DrawPossibleMoves()
    {
        Vector2 v = Vector2.Zero;
        foreach (Move m in possibleMoves.moves)
        {
            v.X = m.x;
            v.Y = m.y;
            spriteBatch.DrawString(gameFont, m.name, v, Color.Gray);
            v.X -= 1;
            v.Y -= 1;
            spriteBatch.DrawString(gameFont, m.name, v, Color.Gray);
            v.X -= 1;
            v.Y -= 1;
            spriteBatch.DrawString(gameFont, m.name, v, Color.Black);
            v.X -= 1;
            v.Y -= 1;
            spriteBatch.DrawString(gameFont, m.name, v, Color.Black);
            v.X -= 1;
            v.Y -= 1;
            spriteBatch.DrawString(gameFont, m.name, v, Color.Azure);
        }
    }
```

PODSUMOWANIE

W ten oto sposób dowiedziałeś się, jak można parsować dokumenty XML na dwa całkiem różne sposoby. Jeśli zamierzasz pracować z plikami *XML*, to na pewno zawarte informacje w tym artykule będą bardzo przydatne. Dobrej zabawy :)

Łukasz Klejnberg lukasz.klejnberg@gmail.com

Pracuje w firmie GG Network S.A. jako Programista Aplikacji Mobilnych w dziale Qt, w którym to m. in. zajmuje się rozwojem wersji mobilnej komunikatora GG na telefony z systemem Symbian. Pasjonuje się programowaniem na platformy mobilne, udziela się w społecznościowym projekcie Metroone.pl, jak i też eZ Publish Community.

Marek Sawerwain

OpenCL – standard nie tylko dla kart graficznych

Technologia CUDA zdobyła bardzo dużą popularność w dziedzinie uniwersalnych obliczeń wykonywanych z pomocą kart graficznych. Jednakże pomimo popularności CUDY, standardowym rozwiązaniem w dziedzinie obliczeń na kartach graficznych jest OpenCL. Warto więc poznać standard OpenCL, gdyż jak się wydaje, będzie on stosowany coraz szerzej, nie tylko w kontekście kart graficznych, ale również w przypadku tradycyjnych procesorów czy innych rozwiązań o wysokiej wydajności.

Wykorzystanie kart graficznych firmy AMD oraz NVIDIA do realizacji obliczeń związanych z kartami graficznymi to obecnie bardzo popularny temat, i wiele aplikacji np. związanych z przetwarzaniem grafiki, multimediów, a nawet programy do kompresji danych pozwalają na wykorzystanie mocy obliczeniowej kart graficznych.

Obecnie istnieją dwa standardy tworzenia programów dla kart graficznych: technologia CUDA firmy NVIDIA oraz standard OpenCL (lub krótko OCL), który jest odpowiednikiem OpenGL, ale tym razem do realizacji obliczeń. Obydwa rozwiązania mają swoje zalety i wady. CUDA obecnie jest bardzo popularna, jednakże ta technologia działa tylko z kartami firmy NIVIDA, natomiast OpenCL to standard ustanowiony przez organizację Khronos, co oznacza, iż aplikacje opracowane za pomocą OCL działają na kartach graficznych obydwu wymienionych przed chwilą producentów.

Obydwie technologie posiadają podobny model programowania, zaletą CUDY jest również pełna integracja z językiem C/C++, właściwie to należy mówić o CUDA C i C++. Natomiast OpenCL oferuje oddzielny język podobny do C, ale nie jest to C++. W CUDA można np. tworzyć klasy języka C++, których metody są realizowane przez kartę graficzną. W przypadku OpenCL tworzone są tzw. kernele (jądra obliczeniowe albo procedury obliczeniowe) stanowiące oddzielny program, choć np. istnieje pakiet Aparapi dla języka Java, który funkcjonuje w sposób identyczny jak CUDA, to znaczy program przeznaczony do realizacji na karcie graficznej opracowuje się w języku Java, dalej jest tłumaczony na OpenCL, aby następnie został wykonany przez kartę graficzną (w międzyczasie jest tłumaczony z OpenCL na odpowiedni assembler, ale to już przemilczymy).

Należy także dodać, iż OpenCL można implementować na zwykłych procesorach, przykładem jest pakiet Intel OpenCL, gdzie dana procedura obliczeniowa, choć opracowana w standardzie OCL jest wykonywana przez procesor przy wykorzystaniu wektorowych rozkazów SSE czy najnowszych AVX.

PRZYKŁAD PIERWSZY – SUMA LICZB

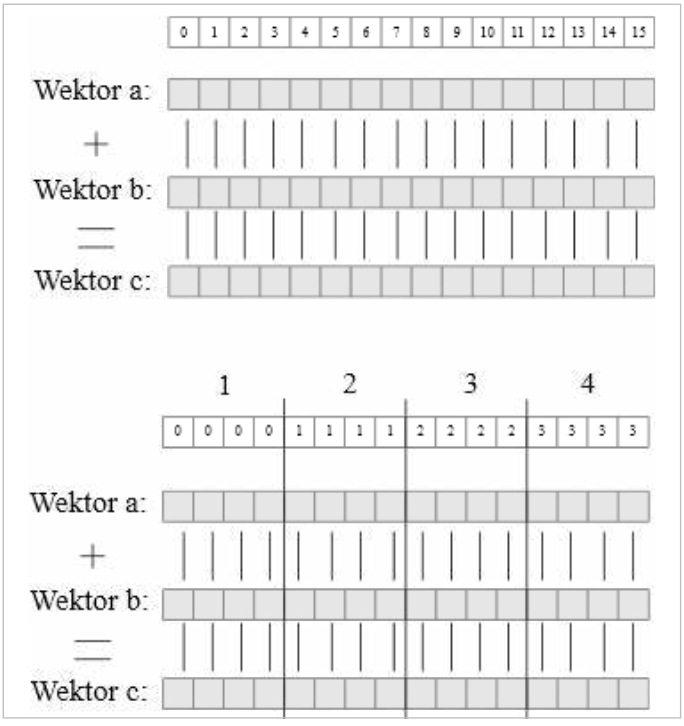
Jako pierwszy przykład jak zawsze warto wybrać nieskomplikowane zadanie, a takim będzie dodanie do siebie dwóch ciągów liczby, czyli wektorów, choć z punktu widzenia OpenCL będą to jednowymiarowe tablice. Taką operację naturalnie zapisujemy za pomocą jednej pętli **while**, np. w języku C przedstawia się to następująco:

```
int i;, a[N],b[N],c[N]
for( i=0 ; i<N ; i++ ) {
   a[i]=b[i] + c[i];
}
```

Wszystkie trzy tablice naturalnie muszą być tych samych wymiarów. Zamiana powyższego kodu na równoważny kod w OpenCL także nie będzie trudna. Tym bardziej iż łatwo dla tego zadania poddać program równoległy, ponieważ poszczególne elementy tablicy mogą być przetwarzane w sposób niezależny. Posiadając wiele jednostek obliczeniowych, można dodawać różne fragmenty wektorów, nie martwiąc się o spójność danych.

Jądro obliczeniowe OpenCL zostało przedstawione na Listingu 1. Przekazujemy trzy argumenty, pierwszy argument bez słowa **const** będzie zawierać wektor wynikowy, natomiast dwa pozostałe parametry o nazwach **b** oraz **c** stanowią wektory wejściowe. Jak widać, znikła pętla **while**, bowiem jej rolę przejęły poszczególne jednostki obliczeniowe, dlatego w pierwszej kolejności odczytujemy, jakim indeksem będziemy się zajmować:

```
int i = get_global_id(0);
```



Rysunek 1. Idea przetwarzania równoległego wektora za pomocą karty graficznej

Wartości funkcji **get_global_id** są związane z rozmiarem tzw. siatki obliczeniowej (w naszym przypadku będzie miała kształt i wielkość wektora), krótko mówiąc będzie to liczba z zakresu od zera do **n-1**, gdzie **n** to ilość elementów, co oznacza, iż **get_global_id** będzie zwracać numer indeksów dla naszych tablic reprezentujących wektory. Rysunek 1 przedstawia ideę takiego sposobu przetwarzania, przy czym możemy to zrealizować na dwa sposoby: można przydzielać poszczególne jednostki przetwarzające oddzielnie dla każdego punktu lub nieco inaczej, gdyż kilka punktów może być przetwarzanych przez jedną jednostkę obliczeniową.

Można także usunąć instrukcję warunkową, odpowiedzialną za sprawdzanie, czy został przekroczony zakres wymiaru wektora, jednakże w takim przypadku należy określić rozmiar siatki obliczeniowej, na równy co do ilości elementów zawartych w wektorze. Obecność instrukcji **if** pozwala bowiem na zastosowanie większej siatki obliczeniowej, o wielkości dopasowanej do ilości dostępnych rdzeni obliczeniowych, co zazwyczaj przyczynia się do większej wydajności obliczeń.

Listing 1. Suma dwóch wektorów w OpenCL

```
__kernel void sum(const unsigned int n,
   __global float* a, __global const float* b, __global
float* c) {
   int i = get_global_id(0);
   if(i < n)
      a[i] = b[i] + c[i];
}
```

PYOPENCL NA DOBRY POCZĄTEK

Uruchomienie jądra obliczeniowego będzie najłatwiejsze w języku Python z wykorzystaniem pakietu PyOpenCL. Listing 2 zawiera prawie kompletny kod źródłowy, usunięto tylko pewne fragmenty procedury obliczeniowej, które znajdują się na Listingu 1.

Zadania, jakie nalezy wykonać, dzielimy na kilka etapów. Na początek należy włączyć dwa pakiety: NumPy – a jest to biblioteka do operacji na wektorach i macierzach oraz naturalnie PyOpenCL ułatwiający korzystanie z OpenCL i współpracujący z pakietem NumPy.

Pierwsza ważna czynność w kontekście OpenCL, jaką należy wykonać, to wybór urządzenia, jednakże my na razie będziemy korzystać z pakietu PyOpenCL, więc ten krok jest wykonywany automatycznie i możemy go pominąć. Trzeba jednak przygotować dane, w pierwszej kolejności określamy ich wielkość zmienną **VECTOR_SIZE** oraz tworzymy tablice o nazwach **a** oraz **b**, które są tablicami liczb zmiennoprzecinkowych o typie **float**. Zaletą pakietu NumPy jest zgodność typów z typami OpenCL, co znacząco ułatwia tworzenie procedur obliczeniowych.

Drugie zadanie to utworzenie kontekstu obliczeniowego (czyli w pewnym sensie uzyskanie połączenia z urządzeniem), reprezentowanego przez zmienną **ctx**, oraz utworzenie kolejki poleceń – zmienna **q**. Ponieważ trzeba przenieść dane z wektorów **a** i **b** do pamięci karty graficznej, robimy to w następujący sposób:

```
a_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY
| pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=a)
```

Powyższa linia kodu utworzy w pamięci urządzenia OCL nowy bufor w trybie tylko do odczytu oraz przeniesie do niego zawartość zmiennej **a**. Identycznie postąpimy w przypadku zmiennej **b**, ale trzeci bufor zawierać będzie rezultat naszego dodawania, więc postępujemy nieco inaczej:

```
dest_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.WRITE_
ONLY, b.nbytes)
```

Utworzony bufor jest przeznaczony wyłącznie do zapisu, a jego wielkość jest określona bezpośrednio w bajtach, wykorzystując zmienną **b** i pole **nbytes**, w którym znajduje rozmiar danych zawartych w zmiennej **b**.

Po przygotowaniu bufora możemy przystąpić do utworzenia obiektu reprezentującego jądro obliczeniowe, a zrobimy to w dwóch etapach: w pierwszym tworzony jest obiekt programu (zmienna **prgobj**), a w drugim budowany obiekt (zmienna **prg**), który reprezentuje procedurę obliczeniową:

```
prgobj = pyopencl.Program(ctx, "treść jądra
obliczeniowego")
prg = prgobj.build()
```

Używając zmiennej **prg**, można uruchomić program poprzez wywołanie metody o takiej samej nazwie jak funkcja jądra obliczeniowego:

```
prg.sum( q, a.shape, None, a_buf, b_buf, dest_buf )
```

Pierwszy argument to kolejka, ale w drugim określamy kształt siatki obliczeniowej, zgodnie z naszymi wcześniejszymi ustaleniami będzie to wektor indeksowany wartościami od zera do **n-1**. W trzecim parametrze, gdzie podano wartość **None**, i jest to określenie tzw. lokalnej siatki obliczeniowej, której w naszym przykładzie nie wykorzystujemy, gdyż ograniczamy się do siatki

Instalacja pakietu PyOpenCL

Sposób instalacji zależy od systemu operacyjnego, jakiego używamy, pomocna jest także strona dokumentacji: http://wiki.tiker.net/PyOpenCL/Installation. W przypadku Windows, najwygodniej będzie skorzystać z bardzo bogatej dystrybucji o nazwie PythonXY, dystrybuowanej jako plik instalacyjny. Zawiera ona naturalnie Pythona w wersji 2.7, bardzo dobre środowisko pracy Spyder oraz wiele dodatkowych pakietów, w tym także pakiet NumPy. Nie ma natomiast pakietu PyOpenCL. Pakiet ten znajduje się na stronie projektu PythonXY i jest to również plik instalacyjny. Instalując pakiet PythonXY, a później PyOpenCL, uzyskujemy kompletne środowisko do pracy z językiem Python z obsługą OpenCL.

W przypadku systemu Linux warto przeprowadzić samodzielną kompilację, ale również nie jest to trudny proces, po ściągnięciu źródeł należy rozpakować archiwum:

```
tar xvfz pyopencl-2012.1.tar.gz
```

Następnie przechodzimy do katalogu z rozpakowanym PyOpenCL i można zainstalować pomocniczy pakiet **setuptools** dla Pythona oraz pakiet **NumPy**:

```
su -c "python ez_setup.py"
su -c "easy_install numpy"
```

Następnie konfigurujemy oraz instalujemy pakiet PyOpenCL:

```
python configure.py
su -c "make install"
```

Podczas konfiguracji może się okazać, iż nie można odszukać plików nagłówkowych oraz bibliotek OpenCL: należy w takim przypadku podczas konfiguracji wskazać katalog, gdzie zostały zainstalowane, np.:

```
python configure.py \
   --cl-inc-dir=/opt/cuda/include \
   --cl-lib-dir=/opt/cuda/lib \
   --cl-libname=OpenCL
```

globalnej. Ostatnie trzy argumenty to bufory zawierające dane naszych wektorów.

Pozostałe czynności w programie polegają na sprawdzeniu poprawności obliczeń. Wcześniej odczytujemy rezultat za pomocą copy:

```
pyopencl.enqueue_copy( … )
```

Sprowadza się to do kopiowania danych z bufora karty graficznej do zmiennej znajdującej się w pamięci RAM systemu operacyjnego.

Listing 2. Skrypt do obliczania sumy dwóch wektorów w PyOpenCL

```
#! /usr/bin/python

import pyopencl
import numpy

VECTOR_SIZE = 2 ** 16

a = numpy.random.rand(VECTOR_SIZE).astype(numpy.float32)
b = numpy.random.rand(VECTOR_SIZE).astype(numpy.float32)

ctx = pyopencl.create_some_context()
q = pyopencl.CommandQueue(ctx)

a_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY
|
   pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=a)
b_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY
|
   pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=b)
dest_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.WRITE_
ONLY, b.nbytes)

prgobj = pyopencl.Program(ctx, """
   __kernel void sum( …. ) {
…... # usunięte fragmenty
   }
   """)
prg = prgobj.build()

prg.sum(q, a.shape, None, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
pyopencl.enqueue_copy(q, a_plus_b, dest_buf)

print numpy.linalg.norm( a_plus_b – (a+b) )
```

OBSŁUGA LICZB O PODWÓJNEJ PRECYZJI

Pierwsze wersje kart graficznych, które można było stosować do realizacji obliczeń uniwersalnych, nie obsługiwały liczb o podwójnej precyzji, jednakże ostatnie serie kart graficznych oferują wsparcie dla tego typu liczb. W naszym przykładzie z Listingu 1, aby uzyskać obsługę tego typu liczb, musimy do procedury obliczeniowej dodać tylko odpowiednie wyrażenie preprocesora:

```
#pragma OPENCL EXTENSION cl_khr_fp64: enable
A także, co jest oczywiste poprawić nagłówek funkcji:
   __kernel void sum(__global const double *a,
      __global const double *b,
      __global double *c)
   { … }
```

W samym skrypcie wykonujemy tylko jedną zmianę, podczas tworzenia zamiennych, np.:

```
a = numpy.random.rand(VECTOR_SIZE).astype(numpy.
float64)
```

Stosujemy typ **float64** z pakietu NumPy, który jest bezpośrednim odpowiednikiem typu **double**.

VEXCL – UŁATWIENIA DLA C++

Zanim przedstawimy nasz główny przykład dotyczący wykrywania krawędzi za pomocą filtru Sobela, warto spojrzeć na niedawno wydaną bibliotekę VexCL. Jest to pakiet wzorców dla języka C++, współpracujący z OpenCL i ułatwiający w wielu aspektach korzystanie z OCL. Stanowi on odpowiednik pakietu PyOpenCL, ale dla języka C++. Swoistą wadą tego rozwiązania jest jego nowoczesność, bowiem wymaga on kompilatora zgodnego ze standardem C++11, więc przykłady i testy najwygodniej przeprowadzać za pomocą kompilatora GCC przynajmniej w wersji 4.6.x i systemu Linux, aby zapoznać się z tą biblioteką.

Nie ma potrzeby instalacji tego pakietu, bowiem są to tylko pliki nagłówkowe; po ściągnięciu aktualnej wersji z repozytorium GIT wystarczy skopiować zawartość katalogów CL oraz vexcl do katalogu, gdzie znajdują się źródła naszego projektu.

Listing 3 zawiera bardzo prosty przykład, gdzie sprawdzamy, jakie urządzenia OpenCL są dostępne w systemie. Uzyskanie informacji o urządzeniach sprowadza się tylko do jednej linii kodu:

```
auto dev = device_list(Filter::All);
```

Obiekt **dev** zawierać będzie listę wszystkich urządzeń, które za pomocą pętli **for** możemy przejrzeć oraz wyświetlić informacje o urządzeniach. Możliwe jest też wskazanie opisowe, jakie urządzenie nas interesuje, np. utworzenie kontekstu dla kart Radeon z obsługą liczb o podwójnej precyzji również sprowadza się do jednej linii kodu:

```
vex::Context ctx(Filter::Name("Radeon") &&
Filter::DoublePrecision);
```

Listing 3. Lista dostępnych urządzeń OpenCL

```
#include <iostream>
#include <vexcl/vexcl.hpp>

using namespace vex;

int main() {
   std::cout << "OpenCL devices:" << std::endl <<
std::endl;
   auto dev = device_list(Filter::All);
   for (auto d = dev.begin(); d != dev.end(); d++) {
   std:cout << "  " << d->getInfo<CL_DEVICE_NAME>() <<
std::endl
   << "  CL_PLATFORM_NAME = "
   << cl::Platform(d->getInfo<CL_DEVICE_PLATFORM>()).
         getInfo<CL_PLATFORM_NAME>() << std::endl
      << "  CL_DEVICE_OPENCL_C_VERSION   = "
      << d->getInfo<CL_DEVICE_OPENCL_C_VERSION>() << std::endl
      << "  CL_DEVICE_MAX_COMPUTE_UNITS   = "
      << d->getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>() << std::endl
      << "  CL_DEVICE_HOST_UNIFIED_MEMORY = "
      << d->getInfo<CL_DEVICE_HOST_UNIFIED_MEMORY>() << std::endl
      << "  CL_DEVICE_GLOBAL_MEM_SIZE     = "
      << d->getInfo<CL_DEVICE_GLOBAL_MEM_SIZE>() << std::endl
      << "  CL_DEVICE_LOCAL_MEM_SIZE      = "
      << d->getInfo<CL_DEVICE_LOCAL_MEM_SIZE>() << std::endl
      << "  CL_DEVICE_MAX_CLOCK_FREQUENCY = "
      << d->getInfo<CL_DEVICE_MAX_CLOCK_FREQUENCY>() << std::endl
      << std::endl;
   }
}
```

Łatwo także zrealizować zadanie, jakie wykonaliśmy na początku tego artykułu. W pierwszej kolejności tworzony jest kontekst, zawierający m.in. kolejkę do realizacji obliczeń:

```
vex::Context ctx(Filter::All);
```

Warto sprawdzić, czy istotnie odnaleziono odpowiednie urządzenia obliczeniowe, poprzez sprawdzenie, czy kolejka jest pusta:

```
if (ctx.queue().empty()) {
   // brak urządzeń OpenCL
}
```

Utworzenie zmiennej o N elementach typu double w pamięci operacyjnej CPU oraz wypełnienia jej wartościami losowymi realizujemy w następujący sposób (wykorzystujemy wyrażenie lambda):

```
std::vector<double> host_vec1(N);
std::generate(host_vec1.begin(), host_vec1.end(), [ ]( )
{
   return double(rand()) / RAND_MAX;
});
```

Umieszczenie nowo powołanej zmiennej w pamięci urządzenia obliczeniowego OpenCL to także jedna linia kodu:

```
vex::vector<double> x ( ctx.queue(), host_vec1 );
```

Po przygotowaniu wszystkich dodatkowych zmiennych, realizacja dodawania dwóch wektorów sprowadza się do następującego wyrażenia:

```
x = y + z;
```

Pozostaje tylko skopiować zawartość zmiennej z urządzenia OCL do pamięci operacyjnej CPU:

```
copy(x, host_vec1);
```

OpenCL dla CPU, GPU, a nawet FPGA

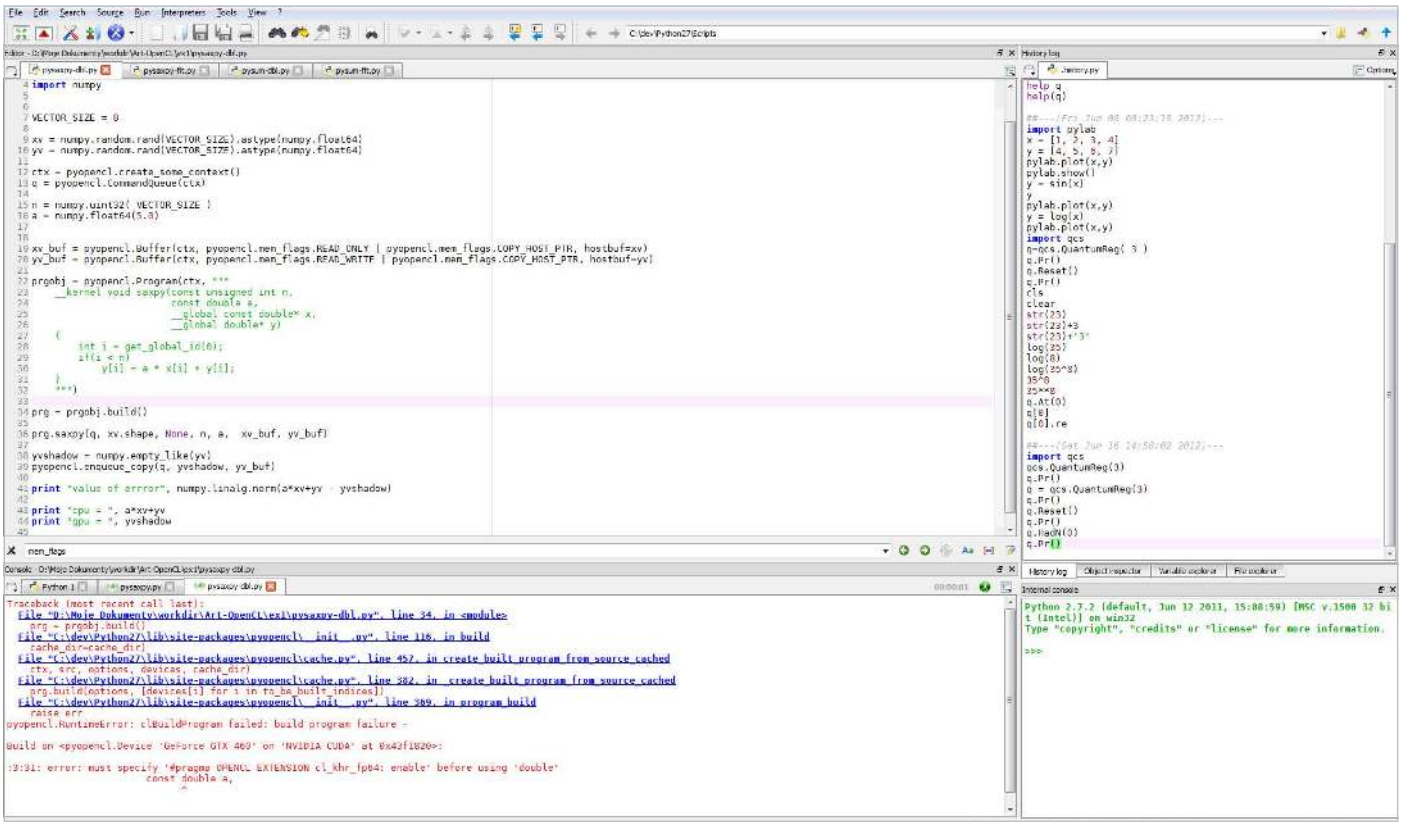
Technologia CUDA to dziś najbardziej popularna technika programowania kart graficznych, jej główną zaletą jest daleko posunięta integracja z językiem C/C++, co powoduje, iż czytając program CUDY, łatwo przeoczyć, iż dany fragment programu jest wykonywany przez kartę graficzną. Obecnie jednak na popularności zyskuje OpenCL, który ma pewną przewagę nad CUDA, gdyż może być stosowany nie tylko dla kart graficznych, ale również w kontekście tradycyjnych procesorów wyposażonych w wektorowe jednostki oferujące instrukcje typu SSE, AltiVec, Neon czy AVX. Nowością jest też wykorzystywanie standardu OpenCL dla układów FPGA, które po odpowiednim zaprogramowaniu również mogą stanowić wydajną jednostkę do realizacji obliczeń.

Istnieje wiele bibliotek wykorzystujących OpenCL, takich jak ViennaCL oraz Magma, jednak są to biblioteki do realizacji obliczeń numerycznych. Jednym z ogólnie dostępnych programów o otwartym źródle, w którym będzie stosowany OpenCL, jest GIMP, a dokładnie GEGL, gdzie w następnej wersji OCL będzie szeroko stosowane. Innym przykładem jest program WinZIP, bowiem podczas kompresji oraz dekompresji wykorzystuje się OpenCL do przyspieszenia tych operacji.

FILTR SOBELA DO WYKRYWANIA KRAWĘDZI

Poprzednie przykłady wykorzystywały dodatkowe pomocnicze biblioteki, których zadaniem było ułatwienie korzystania z OpenCL, teraz możemy już podać nieco bardziej skomplikowany przykład, tworząc program w podstawowym API standardu OCL. Będzie to filtr Sobela do wykrywania krawędzi w obrazie.

Realizację takiego filtru można dość łatwo opisać, bowiem polega to na tym, iż do każdego piksela obrazu stosowana jest odpowiednia macierz, co sprowadza się do tego, iż poszczególne sąsiadujące piksele są przemnażane przez elementy tej macierzy i w ten sposób powstaje piksel wynikowy umieszczany w obrazie



Rysunek 2. Środowisko Spyder oraz skrypt do sumy dwóch wektorów

wyjściowym. Jednak natychmiast pojawia się problem, w jaki sposób potraktować piksele leżące np. w pierwszej kolumnie albo w pierwszym wierszu. Ten problem obchodzi się bezpośrednio, analizując piksele z pominięciem kłopotliwego pierwszego i ostatniego wiersza oraz pierwszej i ostatniej kolumny. Ogólna idea takiego przetwarzania obrazu oraz postać macierzy reprezentującego filtr Sobela została przedstawiona na Rysunku 3. Trzeba także otrzymane wartości danego piksela dla współrzędnej x oraz y odpowiednio dodać, w przykładach wykorzystujemy w tym celu wartość bezwzględną, wcześniej ograniczając wartość sumy do zakresu od 0 do 255.

Rezultat działania filtru Sobela możemy zaobserwować na Rysunku 3, dla przykładowego obrazu pt. Lena, bardzo często wykorzystywanego w prezentacji różnego rodzaju algorytmów graficznych. W przypadku trybu szarości, filtr Sobela funkcjonuje tak, jak opisano to w poprzednim akapicie. Dla kolorów RGB możemy zastosować identyczne podejście, ale oddzielnie dla każdej składowej.

Listing 4 przedstawia przykładową procedurę obliczeniową, która realizuje filtr Sobela w trybie szarości. Tryb RGB realizuje się podobnie, jednak kompletne jądro obliczeniowe jest zbyt duże i zostało umieszczone w materiałach dodatkowych umieszczonych na stronie WWW magazynu. Idea procedury obliczeniowej jest podobna do naszego pierwszego przykładu, ale tym razem przetwarzana struktura danych to nie wektor, ale macierz, bo możemy potraktować obraz 2D jako macierz. Nie ma pętli **for**, gdyż podobnie jak w przykładzie z wektorem, indeksy poszczególnych elementów są odczytywane za pomocą funkcji **get_global_id** z wartościami zero oraz jeden, gdyż siatka obliczeniowa jest dwuwymiarowa.

W jądrze obliczeniowym zanim przystąpimy do obliczenia wartości wskazanego punktu, pierwsze dwie instrukcje warunkowe zajmują się ramką obrazu, bowiem jak wcześniej powiedzieliśmy, nie możemy obliczać wartości punktów leżących np. w pierwszym wierszu. Zajmuje się tym nasza główna instrukcja warunkowa sprawdzająca, czy podany punkt należy do wnętrza obrazu, jeśli tak, to przeprowadzamy odpowiednie obliczenia.

Nie będziemy opisywać wszystkich szczegółów, w jaki sposób tworzymy poszczególne obiekty, zajmiemy się tylko bezpośrednio tworzeniem jądra obliczeniowego. Rozpoczynamy ten proces od kompilacji kodu źródłowego:

```
program = clCreateProgramWithSource(context, 1, (const
char **) &KernelSource, NULL, &err);
```

Następnie można utworzyć obiekt programu:

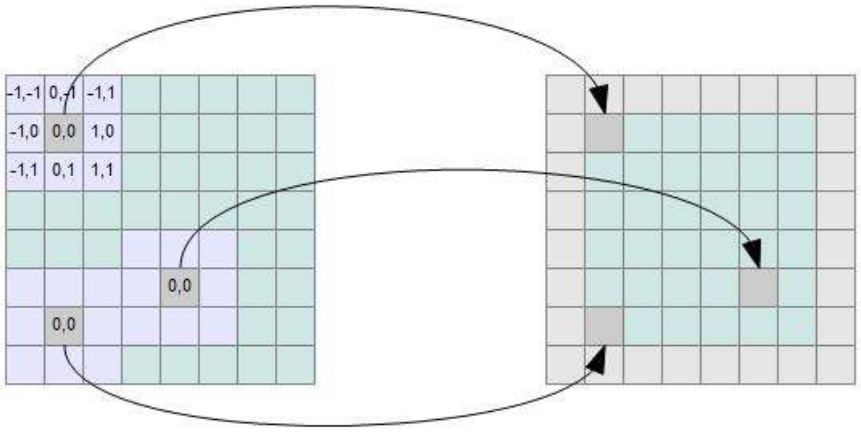
```
err = clBuildProgram(program, 1, &device_id, NULL, NULL,
NULL);
```

Dysponując obiektem, możemy przygotować kernel, gdzie podajemy nazwę procedury obliczeniowej:

```
kernel = clCreateKernel(program, "sobel_grey", &err);
```

Po utworzeniu jądra obliczeniowego, należy ustalić parametry wywołania, warto do zmiennej **err** za pomocą alternatywy logicznej wpisywać kody powrotne poleceń **clSetKernelArg**. Poszczególne parametry: obraz wejściowy i obraz wyjściowy, a także wymiary obrazu ustalamy w następujący sposób:

Rysunek 3. Obliczanie wartości poszczególnych pikseli dla filtru Sobela



$$px = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad py = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

$$p = \sqrt{px^2 + py^2} \quad \text{albo} \quad p = |px| + |py|$$

```
err = 0;
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem),
(void*)&input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem),
(void*)&output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned int),
&hsx);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int),
&hsy);
```

Jeśli nie uda się nam uzyskać wartości CL_SUCCESS, to naturalnie nie udało się poprawnie ustalić parametrów jądra obliczeniowego:

```
if (err != CL_SUCCESS) { …. }
```

W tym momencie można już uruchomić naszą procedurę obliczeniową, musimy jednak określić rozmiary siatki obliczeniowej globalnej oraz lokalnej, choć nasz przykład nie korzysta z lokalnej siatki.

```
global[0] = hsx; global[1] = hsy;
local[0]=16; local[1]=16;
```

Samo uruchomienie procedury to zadanie dla polecenia **clEnqueueNDRangeKernel**, które działa z wielowymiarowymi strukturami danych, w naszym przypadku dwuwymiarowymi:

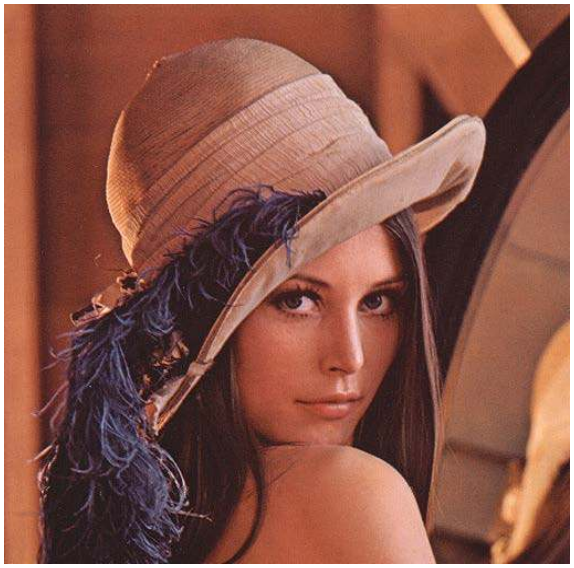
```
err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL,
&global, &local, 0, NULL, NULL);
if (err) { …. }
```

Określenie wymiarów jest dokonywane w trzecim parametrze.

Listing 4. Jądro obliczeniowe dla filtru Sobela w trybie szarości

```
__kernel void sobel_grey( __global uchar* input,
   __global uchar* output,
   unsigned int sx, unsigned int sy) {
   int ix = get_global_id(0);
   int iy = get_global_id(1);

   if ( ix == 0 || ix == (sx - 1) ) {
      output[ iy * sx ] = 255;
      output[ (( iy + 1) * sx) - 1 ] = 255;
   }

   if ( iy == 1 || iy == (sy - 2) ) {
      output[ ix ] = 255;
      output[ ix + ((sy-1) * sx) ] = 255;
   }
   if ( (iy > 0 && iy < ( sy-1)) && (ix>0 && ix<(sx-1)) )
{
      int sum_x=0, sum_y=0, sum=0;

      sum_x += (int)( input[ ix - 1 + (iy - 1)*sy] * -1);
      sum_x += (int)( input[ ix     + (iy - 1)*sy] * -2);
      sum_x += (int)( input[ ix + 1 + (iy - 1)*sy] * -1);

      sum_x += (int)( input[ ix - 1 + (iy + 1)*sy] * 1);
      sum_x += (int)( input[ ix     + (iy + 1)*sy] * 2);
      sum_x += (int)( input[ ix + 1 + (iy + 1)*sy] * 1);

      sum_x = min(255, max( 0, sum_x ));

      sum_y += (int)( input[ ix - 1 + (iy - 1)*sy] * 1);
      sum_y += (int)( input[ ix + 1 + (iy - 1)*sy] * -1);

      sum_y += (int)( input[ ix - 1 + (iy)*sy] * 2);
      sum_y += (int)( input[ ix + 1 + (iy)*sy] * -2);

      sum_y+= (int)( input[ ix - 1 + (iy + 1)*sy] * 1);
      sum_y+= (int)( input[ ix + 1 + (iy + 1)*sy] * -1);

      sum_y = min(255, max( 0, sum_y ));

      sum = abs(sum_x) + abs(sum_y);

      output[ ix + iy * sx] = 255 - (unsigned char)(sum);
   }
}
```

Rysunek 4. Wykryte krawędzie dla obrazu Lena, w trybie szarości oraz w pełnym trybie RGB



Oryginalny obraz

Krawędzie w trybie RGB

Krawędzie w trybie szarości

PODSUMOWANIE

Zaprezentowaną implementację filtru Sobela, a także pierwszy przykład, można jeszcze przyspieszyć poprzez wykorzystanie tzw. lokalnych indeksów, co w efekcie przyczyni się do większej wydajności obliczeń. Warto w tym momencie spróbować zaimplementować tzw. filtr konwolucyjny (ang. *convolution filter*), gdyż sposób realizacji jest podobny do filtru Sobela. Nie jest to zadanie zbyt trudne, a dodatkowo pozwoli na lepsze poznanie techniki tworzenia procedur obliczeniowych w standardzie OpenCL. Można także pokusić się o testy za pomocą pakietu OpenCL dla procesorów firmy Intel, i samodzielnie przekonać się ile zyskamy na wydajności np. względem zwykłej szeregowej implementacji filtru do wykrywania krawędzi.

W sieci:

- Podstawowa strona dotycząca standardu OpenCL:
  http://www.khronos.org/opencl/
- Pakiet Intel OpenCL oferujący implementację OpenCL dla CPU oraz wbudowanych kart HD 4000/2500:
  http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/
- Przykłady OpenCL od Apple dla systemu MacOS:
  https://developer.apple.com/library/mac/navigation/#section=Frameworks&topic=OpenCL
- Pakiet oparty o wzorce dla języka C++ ułatwiający programowanie w OpenCL:
  http://ddemidov.github.com/vexcl
- Przykłady do OpenCL ze strony AMD:
  http://developer.amd.com/sdks/AMDAPPSDK/samples/Pages/default.aspx
- Pakiet Aparapi do obsługi OpenCL w języku Java:
  http://code.google.com/p/aparapi/
- Technologia C++ AMP firmy Microsoft również jest niezależna od producenta karty graficznej:
  http://msdn.microsoft.com/en-us/library/hh265137%28v=vs.110%29.aspx

Marek Sawerwain redakcja@programistamag.pl

Autor, pracownik naukowy Uniwersytetu Zielonogórskiego, na co dzień zajmuje się teorią kwantowych języków programowania, ale także tworzeniem oprogramowania dla systemów Windows oraz Linux. Zainteresowania: teoria języków programowania oraz dobra literatura.

Krystian Kaczor

Wprowadzenie Agile w firmie

Rozważyłeś wszystkie za i przeciw i postanowiłeś wprowadzić Agile w firmie. Być może przeczytałeś artykuł „01 Agile w firmie. Wprowadzać czy nie?" w poprzednim numerze magazynu. Prawdopodobnie wybór narzędzia padł na framework Scrum. To naturalne, że pojawią się w Twojej głowie pytania.

TE PYTANIA TO:

- Jak powiadomić pracowników o zmianach i jak będzie wyglądał proces ich wprowadzenia w całej organizacji?
- Wprowadzamy zmianę z góry do dołu czy z dołu do góry?
- A może wprowadzamy Agile wszędzie, na raz, na zasadzie wielkiego wybuchu?
- Czy potrzebne są szkolenia, coaching, consulting?
- Jeżeli będziemy wprowadzali zmianę, zaczynając od jednego projektu, jaka będzie zależność pomiędzy tym zespołem a resztą organizacji?
- Czy można wprowadzić wybrane elementy, czy trzeba wprowadzać całe narzędzie?

Proponuję, żebyśmy zajęli się tymi zagadnieniami i odpowiedzieli sobie na te pytania w kolejności: komunikacja, wprowadzenie nowego narzędzia, kroki wprowadzania zmiany.

ZMIANA POTRZEBUJE LIDERA

Wprowadzanie Agile w firmie to nic innego jak proces zarządzania zmianą[1].

Pamiętaj, że zmiany w organizacji nie wprowadzasz sam i potrzebujesz wsparcia wielu osób. Właśnie z tych powodów decyzja o wprowadzeniu Agile powinna być jasno i precyzyjnie zakomunikowana. Najlepszym sposobem jest, gdy lider organizacji ogłosi decyzję na spotkaniu z cała firmą. Umiejętności Wystąpień Publicznych grają tutaj dużą rolę i każdy dobry lider powinien je posiadać. Chodzi o to, żeby nie pokazywać kolejnego Power Pointa i czytać ze slajdów, ale poruszyć pracowników firmy, wzbudzić zapał i silną motywację do wsparcia wprowadzenia Agile w firmie. Po czym poznać, czy wystąpienie było dobre? W starożytności było dwóch bardzo znanych mówców: Demostenes z Grecji i Cycero z Italii. Kiedy Cycero przemawiał, tłum krzyczał: „Świetne przemówienie!". Natomiast kiedy przemawiał Demostenes, tłum skandował: „Pozwól nam maszerować!".

Oczywiście nie wszyscy będą zadowoleni ze zmiany i będą jednostki stawiające opór. Zgodnie z NLP, większość ludzi posiada metaprogram Podobieństwo ze zmianą, co oznacza, że większość Twoich pracowników nie lubi drastycznych zmian i dużych różnic. Zatem powinieneś wyjaśnić, jakie rzeczy pozostaną w ich pracy takie same, a jakie zmienią się na lepsze.

Co jest potrzebne, żeby przełamać opór wprowadzenia zmian?

Richard Beckhard[2], ekspert w rozwoju organizacji, zdefiniował formułę, która określa elementy potrzebne do przełamania oporu przed zmianą.

Czynniki po lewej stronie wyrażenia mnożą się i ich iloczyn musi być większy niż prawa strona wyrażenia. Zgodnie z podstawami matematyki, mnożenie czynników wymusza, że wszystkie te czynniki muszą być większe od zera. W przeciwnym razie mnożenie przez zero zawsze da nam zero po lewej stronie.

Najpierw w imię odwiecznej zasady „Nie naprawiaj tego, co nie jest zepsute" potrzebujemy dobrego powodu, żeby wprowadzić zmianę. Obecna sytuacja musi powodować jakąś **dyssatysfakcję**. Co w obecnej sytuacji jest takim powodem? Dlaczego stan obecny nie jest zadawalający? Co nie działa?

Następnie trzeba rozpostrzeć przed pracownikami atrakcyjny, kolorowy obraz **wizji**, do którego organizacja będzie dążyć. Musi być jasne i oczywiste, jak będzie wyglądała firma i codzienna praca po wprowadzeniu zmiany. Każdy pracownik po-

1 http://pl.wikipedia.org/wiki/Zarz%C4%85dzanie_zmian%C4%85

2 http://en.wikipedia.org/wiki/Richard_Beckhard

winien umieć sformułować krótką wypowiedź w dwóch zdaniach na temat co, jak i dlaczego zmieniamy. Fachowo takie sformułowanie wizji nazywa się Elevator Pitch[3].

Trzecim i ostatnim elementem z lewej strony równania są **jasno określone pierwsze kroki**. Potrzebujemy określić tylko pierwsze kroki, ponieważ strategia powinna być wprowadzana w sposób adaptacyjny Plan-Do-Check-Act[4]. Zawsze należy planować pamiętając, że plany są po to, żeby je zmieniać w zależności od rozwoju sytuacji. Planujemy pierwsze kroki i sposób mierzenia rezultatów, a następnie korygujemy plan w zależności od rezultatów.

Kiedy dyssatysfakcja, wizja i pierwsze kroki są zakomunikowane, każdy recytuje wizję na jednym wdechu, pracownicy skandują „Pozwól nam maszerować!", wtedy możesz wprowadzać Agile.

JAK DOBRZE POZNAĆ NARZĘDZIE I DOJŚĆ DO POZIOMU MISTRZA?

Wprowadzenie Agile jest dla organizacji uczeniem się korzystania z nowego narzędzia. To jest dokładnie tak samo, jakbyś na przykład uczył się jeść pałeczkami. Może kiedyś się uczyłeś i nadal pamiętasz, jakie to było uczucie na początku. Przez pewien czas korzystanie z nowego narzędzia będzie wywierało uczucie niewygody. Podobne uczucie niewygody będzie się pojawiać, gdy mentor będzie poprawiał Twoją technikę. Pewnie nie raz byłeś w sytuacji, kiedy uczyłeś się czegoś nowego i nie osiągałeś zadowalających wyników. Wtedy możesz poprosić o radę lub mentoring kogoś, kto osiąga lepsze wyniki. Czasem wystarczy zmienić niewiele lub coś, co z pozoru nie ma wpływu na wyniki. Żeby posłać piłeczkę golfową 20 metrów w prawą stronę przy dystansie 120 metrów, należy uderzyć 2 milimetry dalej od środka. Zwykle rady udzielone przez mentora mogą wydawać się dziwne, a nowe ułożenie rąk na piłce nienaturalne, ale jeśli ćwiczysz dalej, stosując się do tych wskazówek, to uczucie zniknie i będziesz osiągał lepsze rezultaty. Zaufaj metodzie i radom osób z lepszymi wynikami, nawet jeśli nie rozumiesz ich w pełni. Z premedytacją nie używam tutaj określenia „ekspert", bo są różne ścieżki do tego, żeby zostać „ekspertem", i niekoniecznie jest to osiąganie lepszych wyników.

Możesz też pomyśleć o wprowadzaniu Agile jak o nauce jazdy samochodem. Nie można uczyć się dodawać gazu, a potem hamować, następnie zmieniać biegi i tak dalej. Musisz uczyć się tych wszystkich rzeczy na raz, aż wejdą na poziom podświadomych kompetencji i będziesz mógł przejść do doskonalenia techniki jazdy, pokonywania zakrętów, jazdy w poślizgu itd.

Bardzo ważne jest uczenie się od początku prawidłowych nawyków. Dopiero kiedy dobrze znasz teorię, potrafisz doskonale korzystać z narzędzia i odpowiedzieć na pytania: dlaczego? Wtedy jesteś mistrzem i możesz wprowadzać zmiany do metody. Czy będziesz mógł powiedzieć, że posługujesz się pałeczkami, jeżeli w prawej ręce ciągle trzymasz widelec, albo nabijasz duże kawałki jedzenia?

Według różnych badań wyrobienie nowego nawyku zajmuje przeciętnie 21 dni, więc bądź cierpliwy, bo tych nowych nawyków będzie sporo, a rozmiar organizacji komplikuje procesy i wydłuża czas. Rzeczy takie jak proce nauki taki jak na przykład cykl Kolba[5] oraz integracja wiedzy potrzebują czasu i nie da się tego przyspieszyć. Można jedynie ten proces zoptymalizować.

Z mojego doświadczenia wynika, że wprowadzenie Agile do zespołu to co najmniej trzy miesiące, osiągnięcie płynności sześć miesięcy, a wprowadzenie Agile w średniego rozmiaru firmie trwa co najmniej dwa lata.

Proces dążenia do mistrzostwa jest zgodny z koncepcją Shuhari[6]:

- **Shu (Przestrzegaj)** – uczeń uczy się technik i wiernie słucha nauk mistrza, przyjmuje krytycyzm. Uczeń naśladuje mistrza we wszystkim i buduje solidne podstawy do dalszej nauki. Fundamenty są najważniejsze.
- **Ha (Zejdź z drogi)** – po zdobyciu czarnego pasa uczeń zaczyna kwestionować technikę, poszukiwać źródeł, obserwować własne doświadczenia. Uczeń ma silną więź z mistrzem, ale mistrz pozwala mu kroczyć własną drogą. Uczeń kształtuje swój charakter i dokonuje odkryć.
- **Ri (Oddziel)** – uczeń staje się mistrzem. Podstawy sztuki pozostają takie same, ale zastosowanie i metody są charakterystyczne dla tego ucznia. Jeżeli uczeń przerośnie swojego mistrza, to sztuka może zostać rozwinięta i udoskonalona.

Tak więc wprowadź od razu cały framework lub nową metodę i ćwicz, dopóki nie zacznie Ci wychodzić korzystanie z narzędzia w sposób naturalny. Dopiero po tym etapie szukaj odpowiedzi na temat źródeł techniki i zmieniaj ją tak, żeby dostosować do swoich potrzeb i być może również rozwinąć metodę samą w sobie.

Wiedzę o nowych narzędziach można zdobywać samemu z książek i czasopism takich jak to, ale znacznie lepiej i szybciej jest skorzystać ze szkolenia. Pamiętaj, że szkolenia z pewnych powodów (o tym w inny artykule) są niewystarczające, żeby nauczyć się nowych umiejętności. Moim zdaniem, najważniejsze są dwie rzeczy: praktyka i mentoring. Po pierwsze, jak już pisałem wcześniej, musi nastąpić integracja wiedzy zdobytej na szkoleniu, a następnie następuje etap wdrażania wiedzy w praktyce, w kontekście środowiska. Wdrożenie nowych metod przyniesie nowe spostrzeżenia i teorie, przez co może prowadzić do niekoniecznie dobrych decyzji, jeśli nie ma mentora, którego można się doradzić. Smutne, ale prawdziwe jest to, że średnio po trzech tygodniach wiedza ze szkolenia zanika.

Skorzystanie z pomocy doświadczonego coacha lub konsultanta może zaoszczędzić miesiące pracy i pasmo frustracji. Dobry coach lub konsultant oprócz swojej wiedzy wprowadza do firmy bardzo użyteczne spojrzenie z boku i doświadczenie z innymi klientami. Pewnie widział już niejeden Zespół w podobnej sytuacji i ma przynajmniej klika pomysłów na rozwiązanie

3 http://en.wikipedia.org/wiki/Elevator_pitch
4 http://en.wikipedia.org/wiki/PDCA
5 http://pl.wikipedia.org/wiki/David_A._Kolb
6 http://en.wikipedia.org/wiki/Shu_ha_ri

problemu oraz w przypadku Coacha może on lub ona nauczyć zespół samodzielnie tworzyć najlepsze dla niego rozwiązania. Prawdziwy coach potrafi też być katalizatorem przyspieszającym cały proces i pomagającym ludziom odnaleźć się w nowej rzeczywistości. Przecież Agile to tak naprawdę sposób myślenia, który może odbiegać od lat dotychczasowego doświadczenia pracownika. Z tego punktu widzenia jest to głęboka zmiana psychiczna, w której bardzo pomocny może okazać się profesjonalny coach. Z doświadczenia zarówno mojego, jak i wielu kolegów konsultantów wynika, że im firma ma więcej tradycji, a pracownicy dłuższy staż w tej firmie, tym większy będzie opór wprowadzania Agile.

OSIEM PINGWINICH KROKÓW DO CELU

Podczas wprowadzania zmiany w organizacji będzie zachodziło wiele ciekawych procesów, wyłonią się pewne role w organizacji, sformułują się obozy za i przeciw zmianie. Cały proces i zachowania w organizacji są świetnie przedstawione za pomocą metafory w książce „Gdy góra lodowa topnieje. Wprowadzanie zmian w każdych okolicznościach”[7]. Fabuła książki jest zbudowana wokół tego, że pewnego dnia jeden z pingwinów zamieszkujących górę lodową dostrzega, że lód pęka i niedługo dojdzie do katastrofy. Trzeba to dokładnie sprawdzić, przekonać pozostałe pingwiny i powziąć kroki zapobiegawcze.

John Kotter, jeden z autorów tej pozycji, jest ekspertem w zarządzaniu zmianą, napisał kilka książek[8] na ten temat i polecam do nich sięgnąć.

Strategia John’a Kotter’a składa się z ośmiu kroków:

1. Stwórz poczucie pilnej potrzeby - pomóż innym zauważyć potrzebę zmiany i to, jak ważne jest działać natychmiast.
2. Zbierz Zespół Przewodni, który będzie miał umiejętności przywódcze, autorytet, umiejętności komunikacyjne, umiejętności analityczne i poczucie pilności.
3. Zbuduj Wizję i Strategię Zmiany – wyjaśnij, jak przyszłość będzie inna od przeszłości i jak to sprawić.
4. Zakomunikuj Wizję i Strategię – spraw, żeby jak najwięcej osób zaakceptowało i zrozumiało wizję i strategię.
5. Upełnomocnij innych do działania – usuń bariery na drodze osób, które wprowadzają wizję w życie.
6. Wygeneruj krótkoterminowe wygrane – osiągnij szybki, widoczny, ale niedwuznaczny sukces.
7. Nie pozwól odpuścić – dodaj presji po pierwszym sukcesie.
8. Stwórz nową kulturę wspierającą nowe zachowania, żeby zastąpić stare tradycje i nawyki.

Ken Schwaber i Jeff Sutherland w swojej nowej książce „Software in 30 days”[9] opisują podobne aktywności i według nich ekspansja Scrum w całej organizacji zajmuje od jednego miesiąca do pięciu lat. Z kolei utrzymanie i zagnieżdżenie nowej kultury potrwa od pięciu do sześciu lat.

PODSUMOWANIE

Mam nadzieję, że odpowiedziałem na większość pytań, jakie pojawiają się przy okazji wprowadzania Agile w firmie. Już wiesz, jak należy zakomunikować zmiany w organizacji, jak nauczyć się poprawnie korzystać z nowego narzędzia i jakie są kroki wprowadzenia zmiany w całej organizacji. Oczywiście zachęcam Cię także do przeczytania książek i artykułów wspomnianych w tym artykule oraz poszukania dobrego szkolenia dla pracowników połączonego z coachingiem i monitoringiem, żebyś miał wsparcie w całym procesie. Jeśli masz konkretne pytania do sytuacji, w której znalazłeś się Ty i Twoja firma, skontaktuj się bezpośrednio ze mną lub z redakcją i z przyjemnością Ci pomogę. Jeżeli wyrazisz zgodę, możemy opublikować odpowiedź w kolejnym magazynie, żeby skorzystała na tej wiedzy większa grupa czytelników.

Do dzieła!

7 John Kotter, Tytuł oryginału: Our Iceberg Is Melting: Changing and Succeeding Under Any Conditions, Wydawnictwo Helion, One Press, Sierpień 2008
ISBN: 978-83-246-1516-2

8 John P. Kotter „Leading Change”, John P. Kotter „The Heart of Change”

9 Software in 30 Days: How Agile Managers Beat the Odds, Delight Their Customers, And Leave Competitors In the Dust by Ken Schwaber, Jeff Sutherland, ISBN-10: 1118206665, ISBN-13: 978-1118206669

Krystian Kaczor http://kaczor.info/

Krystian Kaczor to coach, trener i konsultant IT, który na co dzień łączy ze sobą dwa światy, świat technologii oraz świat ludzkiej psychiki i umiejętności miękkich. Krystian zdobył wszechstronne doświadczenie, pracując w różnych rolach i projektach w ciągu prawie dziesięciu lat. W swojej karierze odkrył, że w osiągnięciu sukcesu ważniejsze od wiedzy technicznej są umiejętności miękkie i pełna zaangażowania współpraca zespołu i właśnie dlatego Krystian zainteresował się NLP i coachingiem. Pełen profil i CV można znaleźć na stronie www autora.

Artur Machura

Przegląd możliwości analizy w przedsięwzięciach IT

Artykuł ma na celu spojrzeć na wartość pracy analitycznej w przedsięwzięciach IT. Poruszy wartość tej pracy przez pryzmat czasu i zmieniających się możliwości w tym zakresie.

Artykuł skierowany jest do wszystkich uczestników przedsięwzięć IT, ale i do osób decyzyjnych. Tych, którzy inwestują i inicjują przedsięwzięcia oraz decydują o rolach i zakresach obowiązków poszczególnych uczestników projektów (m.in. Analityków).

W artykule zostaną przybliżone takie zagadnienia jak:

- uzasadnienie ekonomiczne i sens prac analitycznych w przedsięwzięciach IT w perspektywie czasu i możliwości, jakie niesie ze sobą nieustannie rozwijająca się Inżynieria Oprogramowania,
- skutki pominięcia prac analitycznych w przedsięwzięciach IT oraz konkretne przykłady dowodzące, że odpowienio dobrany warsztat metodyczno-narzędziowy mógłby uniemożliwić doprowadzeniu do takich problemów,
- organizacje międzynarodowe i możliwości wynikające z zastostosowania wiedzy budowanej w dziedzinie analizy biznesowej na przestrzeni kilkudziesięciu lat,
- dostosowanie prac analitycznych do celów przedsięwzięcia IT, zamawiającego, wykonawcy oraz dojrzałości procesu wytwórczego,
- usługi świadczone przez autora tekstu - analityka biznesowego, członka organizacji - Organization of Software Engineers www.software-engineers.org.

Pojęcie analizy biznesowej w przedsięwzięciach IT przybiera wiele postaci. Cel i zakres tych prac zależny jest zarówno od organizacji zamawiającego i wykonawcy, jak i kategorii projektów. Oczekiwana jakość pracy i dostarczanego rozwiązania IT leży zapewne u podstaw podejścia do analizy biznesowej. Gdzie istotne również są pewne przyjęte kryteria ekonomiczne i rynkowe wobec projektów i wykonawców. Biorąc pod uwagę powyższy akapit, trudno o jednoznaczną – inżynierską definicję analizy biznesowej, co zapewne ma niewątpliwy wpływ i na same zróżnicowane oczekiwania wobec kompetencji osoby wykonującej obowiązki tzw. Analityka biznesowego. Dlatego też w niniejszym artykule potraktuje się pewne definicje i zbiory wiedzy raczej jako punkt odniesienia, co również pozwoli na zachowanie potrzebnego dystansu pomiędzy wytycznymi przeróżnych centrów badawczych i doświadczeń w analizie a konkretną firmą realizującą projekt IT.

Uzasadnienie ekonomiczne i sens prac analitycznych w przedsięwzięciach IT w perspektywie czasu i możliwości, jakie niesie ze sobą nieustannie rozwijająca się Inżynieria Oprogramowania.

Wracając do podstaw, przywołam w tych pierwszych słowach artykułu sztandarowe uzasadnienie i sens realizacji prac analitycznych w przedsięwzięciach IT. Porządkując nomenklaturę, wymienia się często zamiennie stanowiska:

- analityk,
- analityk biznesowy,
- analityk IT,
- analityk systemowy,
- analityk procesów biznesowych.

Bardzo często również zdarza się przypisywać obowiązki i odpowiedzialność tej roli innym stanowiskom, jak np. specjalista, projektant, konsultant, konsultant biznesowy, kierownik, programista ...

Wobec powyższych różnie nazwanych stanowisk i ewentualnych rozbieżności, wydaje się zasadne skupienie na wykonywanych obowiązkach w kontekście Inżynierii Oprogramowania aniżeli nazw stanowisk wykorzystywanych przez firmy. Dlatego w pierwszym kroku należałoby rozróżnić oczekiwania osób/ podmiotów pozostających w bezpośrednim kontakcie z analitykiem. Wymienia się tu:

- zamawiającego,
- zespół wykonawcy.

Należy tu zauważyć, że analityk, o którym tu mowa, może być zatrudniony zarówno przez zamawiającego, jak i wykonawcę. Bywa również, że świadczy tylko usługi wobec jednej lub drugiej strony, tj. zamawiającego i wykonawcy. Takie rozróżnienie jest bardzo istotne, bowiem zamawiający oczekuje od analityka przede wszystkim **zdefiniowania i doprowadzenia do takiego rozwiązania, które będzie wspierało bądź realizowało cele biznesowe**. Natomiast wykonawca i zespół wykonawcy skupiają się przede wszystkim na otrzymaniu tego, **co i na kiedy ma wykonać**. Oczywiście na poziomie dużej ogólności te oczekiwania wobec analizy są identyczne, ale podczas wykonywania tych zadań okazuje się zwykle, że pogodzenie potrzeb obu stron jest wyzwaniem samym w sobie. *Kontekst pracy analitycznej związanej z realizacją potrzeb zamawiającego wydaje się najistotniejszy*. Ostatecznie bowiem projekt ma spełniać oczekiwania zamawiającego. W dalszej części artykułu zostaną scharakteryzowane konkretne prace wraz z ich uzasadnieniem ekonomicznym, które analityk mógłby wykonywać w tym zakresie. W dużym uogólnieniu ktoś mógłby stwierdzić, że efektem pracy analityka jest określenie tego - **co rozwiązanie IT powininno dostarczyć zamawiającemu**. Ale już po pierwszym

i bliższym przyjrzeniu się tej pracy okazuje się najczęściej, że wywiązanie się z tego obowiązku wiąże się z:

- dostosowaniem pracy analitycznej do specyfiki przedsięwzięcia,
- kontrolą przebiegu i postępu pracy,
- pozyskaniem i precyzowaniem wymagań,
- zarządzaniem wymaganiami w przedsięwzięciu,
- analizą organizacji zamawiającego,
- analizą wymagań,
- uczestniczeniem w realizacji przedsięwzięcia.

Powyżej scharakteryzowane praktyczne obszary pracy analityka, które związane są ze stwierdzonym wcześniej „co rozwiązanie IT powinno dostarczyć", uświadamiają rzeczowo, jak istotna jest ta rola w przedsięwzięciu. Jednak dla zupełnej jasności, w kolejnych punktach wymienię skutki zbagatelizowania powagi w/w obszarów prac.

- **Niedostosowanie pracy analitycznej do przedsięwzięcia**, tj. przyjętych metod, narzędzi i kompetencji do klienta w zasadzie z góry skazuje projekt na niepowodzenie. Mógłbym się pokusić nawet o stwierdzenie, że psuje wizerunek firmy - wykonawcy. Nie zapominajmy, że praca analityka osadzona jest na styku klienta i wykonawcy. Co oznacza również, że nie można pominąć specyfiki klienta i wykonawcy w realizowaniu prac. Przykładowym problemem może być próba zastosowania przez wykonawcę w projekcie dokumentacji, bez uzgodnienia z klientem szablonu i technik tam wykorzystywanych, co w efekcie może doprowadzić do sytuacji, gdzie klient nie będzie miał czasu na analizę dokumentacji albo zwyczajnie nie zrozumie właściwie zawartych tam informacji!

- **Pominięcie kontroli przebiegu i postępu pracy**, zważywszy na stosowanie przeróżnych modeli procesów wytwórczych (począwszy od tych z rodziny zwinnych, a kończąc na całej grupie formalnych) przekłada się na bardzo poważne konsekwencje. Bowiem prace analityczne muszą być realizowane we właściwym kierunku i podczas ich przebiegu zasilać zespół pewnymi potrzebnymi porcjami informacji. Nie zapominajmy, że na etapie prac analitycznych powstaje koncepcja całego rozwiązania (a ile w historii było takich przypadków, gdzie projekt był przerywany ze względu na przyjęte niepoprawne założenia?)! Na uwagę zasługuje tu fakt, że w taką kontrolę zwykle powinny być zaangażowane co najmniej dwie strony przedsięwzięcia, tj. zarówno zamawiający, jak i wykonawca. Akceptacja zamawiającego jest kluczowa w postępie prac analitycznych.

- **Zbagatelizowanie etapu pozyskiwania i precyzowanie wymagań** prawdopodobnie zalicza się do najbardziej znanych i oczywistych problemów w projektach IT (często analitykowi przypisuje się wyłącznie taką właśnie rolę, tj.

osoby zajmującej się pozyskiwaniem i precyzowaniem wymagań). A więc, sama czynność jest w zasadzie oczywista. Dlatego należałoby wspomnieć o sytuacji, gdzie niefektywnie pozyskuje się wymagania, a same ich precyzowanie również jest dalekie od ideału. Skutki niemalże oczywiste, projekt nie spełnia oczekiwań zamawiającego, co też, najczęściej, rozciąga projekt w czasie. A kto ponosi odpowiedzialność za dodatkową i "nieprzewidzianą" pracę ?

- **Zignorowanie czynności związanych z zarządzaniem wymaganiami**, oczywiście jak wszystkie inne obszary przekłada się na negatywne skutki dla projektu. Ale przybliżając problematykę, mamy przede wszystkim do czynienia z chaosem w projekcie. Zmiana i aktualizacja istniejących wymagań czy też nowe wymagania – bez właściwego zarządzania wprowadzają znaczące zamieszanie do projektu. W rezultacie czego np. trudno szacuje się realny wpływ tych zmian wymagań na realizację prac. A więc i koszty pracy wykonawcy.

- **Niedokonanie analizy organizacji zamawiającego**, z teoretycznego punktu widzenia w ogóle nie powinno mieć miejsca, bo nie ma w tym żadnego sensu. Jak bowiem mielibyśmy dopasować system do organizacji, nie poznawszy tej organizacji? Częste przyczyny takiego stanu rzeczy tj. dodatkowe koszty, sama możliwość analizy organizacji to jednak zagadnienia na odrębne artykuły.

- **Pominięcie takiej pracy, jaką jest analiza wymagań**, po pierwsze podważa samą rolę analityka. Zapewne u podstaw samego etapu analizy jest wyprzedzenie skutków pewnych decyzji projektowych. Pominięcie analizy wymagań, w obrębie pracy z wymaganiami, stawiałoby analityka jedynie jako pewnego przekaźnika pomiędzy klientem a resztą zespołu. Pominięcie tego kroku może doprowadzić do takiej niepożądanej sytuacji, gdzie na bazie "nieprzemyślanych" wymagań powstają koncepcje i samo rozwiązanie. A same skutki mogą być tragiczne dla projektu.

Podsumowując powyższe punkty, wydaje się, że pominięcie jakiegokolwiek obszaru (dostosowanie pracy do przedsięwzięcia, kontrola przebiegu, pozyskiwanie wymagań, zarządzanie wymaganiami, analiza organizacji, analiza wymagań) może niekorzystnie wpłynąć nie tylko na rezultaty etapu analizy, ale i całego przedsięwzięcia. Dla przykładu przywołam konkretne liczby i projekty, które zasadnie informują i uzasadniają sens prac analitycznych w przedsięwzięciach IT.

Zapewne gdzieś u źródeł poruszanej w niniejszym artykule problematyki jest powszechnie znany raport organizacji „The Standish Group" pt. „The Chaos Report", gdzie oficjalnie zauważono, że aż 31,1 % projektów informatycznych zostało anulowanych przed zakończeniem prac. Przy czym ponad połowa ogółu przedsięwzięć (52,7%) przekroczyła swój pierwotny budżet (źródło: www.standishgroup.com).

Istnieje również raport z Polskiego Badania Projektów IT 2010, który odpowiada na kilka pytań, w tym jaka część projektów IT kończy się w Polsce skucesem. Okazuje się, że pełen sukces zaliczyło tylko 21% projektów, częściowy sukces to 46%, natomiast w kategorii porażki znalazło się aż 33% (źródło: http://pmresearch.pl/sites/results).

W interpretacji raportów należy wziąć pod uwagę bardzo istotny fakt. Otóż, współczynnikiem sukcesu (według raportu PMRESEARCH) są:

- zgodność rzeczywistego budżetu projektu z planowanym,
- zgodność rzeczywistego czasu trwania z planowanym,
- zgodność zrealizowanego zakresu z planowanym,
- ocena projektu dokonana przez kierownika,
- ocena satysfakcji klienta dokonana przez kierownika projektu.

gdzie bardzo istotne jest, że u podstaw w/w miar leżą produkty analityczne. Produkty analityczne mogą stanowić punkt odniesienia w różnych aspektach przedsięwzięć, zarówno realizacji prac czy też samej oceny tych prac. Budżet projektu, zakres prac, czas realizacji, ocena projektu czy też satysfakcja klienta mogą i powinny być budowane we współpracy z pracami analitycznymi. Budżet projektu określany jest w dużej mierze w oparciu o to, co ma być rezultatem prac/ produkty. Czas i pracochłonność również jest określana w dużej mierze na podstawie tego, do czego dąży projekt. Idąc dalej, ocena zgodności zrealizowanego zakresu z planowanym kooperuje z postanowieniami zawartymi w podstawach projektu – produktach analitycznych. Ocena projektu czy też satysfakcji klienta, dokonywana przez kierownika - siłą rzeczy powinna bazować w dużej mierze na wymaganiach wobec zamawianego rozwiązania IT.

PODSUMOWANIE

Artykuł poruszył przekrój zagadnień związanych z pracą analityka w przedsięwzięciach IT. Nawiązał do roli analityka oraz jego ewentualnych możliwości w ramach realizacji przedsięwzięć. Miał na celu zasygnalizowanie, jak skrajnie różnie można wykonywać tę pracę i jakie to może mieć przełożenie na efekty zarówno pracy zespołu, jak i całego projektu. Poinformował również, że produkty organizacji międzynarodowych takich jak np. IIBA mogą być bardzo pomocne w kreowaniu warsztatu metodyczno – narzędziowego Analityków. Zauważył jednak, że wyzwaniem jest pogodzenie zastosowania pewnych metod i narzędzi w konkretnym środowisku przedsięwzięcia, gdzie istotne są cechy zamawiającego, wykonawcy, jak i realizowanego przedsięwzięcia IT.

Artur Machura artur.machura@software-engineers.org

Autor zajmuje się zawodowo Analizą Biznesową od 2005 roku. Od samego początku dąży do dostosowywania i zapewnienia odpowiedniego poziomu jakości w przedsięwzięciach IT. Wiąże się to z poznawaniem i stosowaniem praktyk Inżynierii Oprogramowania na różnych poziomach jej postrzegania tj.: jakości, procesów wytwórczych, metod, narzędzi. Jest członkiem Organization of Software Engineers. Tel. kom.: +48 501 176 256.

Sławomir Sobótka

Domain Driven Design krok po kroku

Część IVa: Skalowalne systemy w kontekście DDD - architektura Command-query Responsibility Segregation (stos Write)

Czy możliwe jest stworzenie systemu, który będzie charakteryzował się otwartym na rozbudowę modelem, eleganckim, testowalnym i utrzymywalnym kodem, a jednocześnie będzie przygotowany do skalowania? Czy narzędzia typu Object-relational mapper są panaceum na wszystkie problemy persystencji w systemach biznesowych? Czy baza relacyjna to zawsze najlepszy pomysł na przechowywanie danych? Na te i inne pytania odpowiemy sobie w kolejnej odsłonie naszej serii.

WSTĘP

Po omówieniu ogólnej idei DDD, technik zarówno taktycznego, jak i strategicznego modelowania oraz sposobów wykorzystania popularnych frameworków przyszedł czas na przyjrzenie się architekturze systemu jako całości. Zakładamy, że tworzymy system, co do którego postawiono następujące wymagania niefunkcjonalne:

- dostęp poprzez wiele technologii klienckich: web (w tym ajax), mobile, web service...,
- skalowanie poszczególnych modułów (w sensie wydzielonych jako produkty funkcjonalności) w celu zapewnienia wydajności,
- odporność na awarie pojedynczych modułów – system jako całość pozostaje stabilny,
- stworzenie platformy otwartej na rozszerzenia poprzez wpinanie pluginów.

W celu spełnienia tych wymagań:

- zrewidujemy tradycyjne podejście do architektur warstwowych, rozwijając je w kierunku Architektury Porst & Adapters (ramka „W sieci"),
- zastanowimy się nad kwestią spójności danych – kiedy jest ona krytyczna, a w jakich wypadkach możemy pozwolić sobie na chwilową niespójność, zyskując skalowalność,
- przyjrzymy się miejscu, jakie Object-relational Mapper zajmuje w systemie oraz kiedy jego wykorzystanie nie jest racjonalne,
- zastanowimy się nad odpowiednimi modelami danych i ich formami,
- powrócimy do zagadnienia zdarzeń omawianych w poprzednich częściach, aby wykorzystać ich pełen potencjał,

PROJEKT REFERENCYJNY

Wszystkich tych Czytelników, którzy już teraz chcieliby zapoznać się z kolejnymi zagadnieniami naszej serii, zapraszam do odwiedzenia strony projektu „DDD&CqRS Laven", której adres znajduje się w ramce „W sieci". Dostępne wersje:

- Java – Spring
- Java – EJB 3.1
- .NET - C#

Plan serii

Niniejszy tekst jest czwartym artykułem z serii mającej na celu szczegółowe przedstawienie kompletnego zestawu technik modelowania oraz nakreślenie kompletnej architektury aplikacji wspierającej DDD.

Część I: Podstawowe Building Blocks DDD;
Część II: Zaawansowane modelowanie DDD – techniki strategiczne: konteksty i architektura zdarzeniowa;
Część III: Szczegóły implementacji aplikacji wykorzystującej DDD na platformie Java – Spring Framework;
Część IV: Skalowalne systemy w kontekście DDD – architektura CqRS;
Część V: Kompleksowe testowanie aplikacji opartej o DDD;
Część VI: Behavior Driven Development – Agile drugiej generacji.

DWIE KLASY PROBLEMÓW

Typowy system klasy enterprise obsługuje dwa typy operacji:

- rozkazy wykonujące pewne operacje biznesowe – co prowadzi zwykle do modyfikacji pewnej (relatywnie niedużej) ilości danych, a w konsekwencji do konieczności utrwalenia tychże zmodyfikowanych danych,
- kwerendy (w tym kontekście niekoniecznie w sensie SQL) odpytujące część danych składowanych w systemie.

Statystycznie rzecz biorąc, ilość obsługiwanych rozkazów do ilości obsługiwanych kwerend (w jakimś interwale czasu) różnią się o kilka rzędów wielkości. Innymi słowy odczytów jest o kilka rzędów wielkości więcej niż zapisów – śmiało możemy przyjąć, że dla typowego systemu klasy enterprise może być 5 rzędów wielkości.

Drugim aspektem – obok statystycznego - który różni oba typy operacji jest aspekt jakościowy modelu danych. Model danych potrzebny do wykonania operacji na modelu biznesowym oraz dane potrzebne do prezentacji wizualnej lub komunikacji z innymi modułami to zwykle inne struktury.

Z uwagi na obszerność merytoryczną, czwartą część serii podzielono na dwa osobne artykuły: niniejszy poświęcony Rozkazom oraz część, która ukaże się w przyszłym miesiącu i będzie poświęcona Kwerendom.

POTRZEBA SEPARACJI

Projektując model danych, zwykle wybieramy podejście Relacyjne i skupiamy się na wsparciu dla modelu domenowego – czyli dążymy do Trzeciej Postaci Normalnej. Postać ta, ze względu na brak redundancji, jest optymalna z punktu widzenia modyfikacji danych, zatem pożyteczna dla operacji typu Rozkaz.

Natomiast postać ta w przypadku operacji typu Kwerenda skutkuje pojawianiem się iloczynów kartezjańskich (JOIN w SQL). Dodatkowo Agregaty mogą zawierać dane zupełnie nieistotne w kontekście danej Kwerendy.

Stosowalność Object-relational Mapper

Świat relacyjny na obiektowy mapujemy po to, aby:

- Pobierać w wygodny sposób obiekty biznesowe – wraz z wygodnymi mechanizmami typu Lazy Loading,
- Wykonywać na nich operacje biznesowe zmieniające ich stan – możemy tutaj tworzyć zarówno anemiczne encje modyfikowane przez serwisy, jak również projektować prawdziwe obiekty modelujące reguły i niezmienniki biznesowe (styl Domain Driven Design),
- Utrwalać stan obiektów biznesowych - stan, który zmienił się w poprzednim kroku (korzystając z wygodnych mechanizmów wykrywania "brudzenia" i mechanizmu kaskadowego zapisu całych grafów obiektów).

Jeżeli stosujemy ORM (np. Java Persistence API) do tych klas problemów, to używamy odpowiedniego „młotka" do odpowiedniej klasy problemu. Czyli pobieramy kilka obiektów biznesowych (Agregatów), zmieniamy jego stan, zapisujemy go.

Pisząc „zmieniam stan", nie mam na myśli "edytuję, podpinając pod formularz". Mam na myśli logikę aplikacji (modelującą Use Case/User Story), która modyfikuje mój obiekt biznesowy (uruchamiając jego metody biznesowe lub settery, jeżeli jest on anemiczny). Na Listingu 1 widzimy przykład z poprzedniej części serii – przypomnijmy: Order i Invoice to persystentne Agregaty:

Listing 1. Serwis Aplikacyjny `pl.com.bottega.erp.sales.application.services.Purchase Service` operujący na kilku persystentnych Agregatach

```
public class PurchaseService{
//...
public void approveOrder(Long orderId) {
      Order order = orderRepository.load(orderId);

      //sample: Specification Design Pattern
      Specification<Order> orderSpecification =
         generateSpecification(systemUser);
      if (!orderSpecification.isSatisfiedBy(order))
         throw new OrderOperationException("Order does not
            meet specification", order.getEntityId());

      order.submit();

      Invoice invoice = invoicingService.issuance(order,
                  generateTaxPolicy(systemUser));

      invoiceRepository.save(invoice);
      orderRepository.save(order);
   }
}
```

Jeżeli natomiast chcemy wyświetlić na ekranie dane, np. dane przekrojowe w postaci tabelki (wszyscy wiemy, że dla klienta biznesowego najważniejsze są tabelki, w których można przestawiać kolejność ich kolumn), to narzędzie typu ORM nie jest najlepszym rozwiązaniem tego problemu. W tym wypadku na każdym etapie postępujemy nieracjonalnie:

- Pobieramy z ORM listę obiektów (zamapowanych na całe tabelki w bazie), gdy potrzebuję na ekranie jedynie kilku kolumn z każdej tabelki (dla bazy nie robi to różnicy, ale w przypadku komunikacji sieciowej zaczniemy odczuwać skutki wydajnościowe tej decyzji),
- Mam możliwość korzystania z mechanizmu Lazy Loadingu, który nie ma sensu dla operacji typu "pobierz dane do wyświetlenia" i prowadzi do dramatycznego problemu z wydajnością „N+1 Select Problem",
- Silnik mapera wykonuje niepotrzebne operacje związane ze wsparciem dla Lazy Loading i Dirty Checking, które nie będą wykorzystywane,
- Zdradzam model biznesowy warstwie prezentacji. Być może w prostych aplikacjach z prezentacją w technologii webowej (ta sama maszyna pobiera i prezentuje dane) nie jest to problem - dodatkowo zyskujemy produktywność w pracy. Ale jeżeli klienty są zdalne (np. Android)? Zdradzanie modelu domenowego wiąże się z drastycznym spadkiem bezpieczeństwa (wsteczna inżynieria) oraz z koniecznością koordynacji prac zespołów pracujących nad "klientem" i "serwerem" i zapewnianiem kompatybilności starszych wersji klientów. Co prawda jest to kwestia oczywista, ale w „materiałach ewangelizacyjnych" popularnych platform korporacyjnych można znaleźć nawoływanie do takich „uproszczeń",
- Pracujemy na puli połączeń zestawionej w trybie READ-WRITE, gdy wystarczający jest tryb READ – większość baz danych optymalizuje działanie w tym trybie.

WARSTWY SĄ DOBRE, ALE DWA STOSY WARSTW SĄ JESZCZE LEPSZE

W dotychczasowych artykułach opieraliśmy architekturę aplikacji na stylu warstwowym, wprowadzając warstwy:

- prezentacji,
- logiki aplikacji,
- logiki domenowej (z ew. podziałem na 4 poziomy modelu: Capability, Operations, Policy, Decission Support),
- infrastruktury.

Architektura Command-query Responsibility Segregation przedstawiona na Rysunku 1 rozdziela odpowiedzialność kodu na dwa wyraźnie stosy warstw. Stos „Write" zawiera opisane powyżej warstwy i obsługuje omawiane wcześniej polecenia typu Command – operacja biznesowa i zmiana stanu systemu. Drugi stos „Read" jest mniej złożony i zawiera serwisy wyszukujące dane.

Separacja stosów nie dotyczy jedynie kodu. Problemy wydajnościowe opisane w sekcji „Potrzeba separacji" mogą (ale nie muszą – o czym w dalszej części) wymóc decyzję o rozdziale modelu danych. Model odpowiedni do operacji biznesowych zwykle nie jest odpowiedni do szybkiego serwowania danych np. na potrzeby prezentacji.

STOS WRITE (COMMAND)

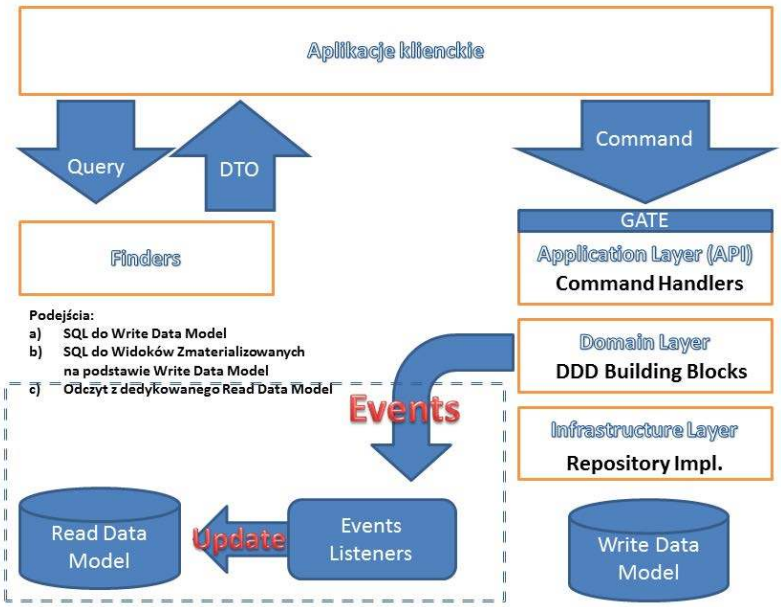
Stos „Write" niejawnie omawialiśmy w poprzednich częściach – były to nasze serwisy aplikacyjne. Natomiast w niniejszym artykule – bazując na założeniach i wiedzy z poprzednich części – wprowadzimy pewne rozszerzenia.

Stos „Write" możemy tworzyć w dwóch stylach:

- „klasycznym" w postaci serwisów,
- opartym na dostosowanym do architektury klient-serwer Wzorcu Projektowym Command.

Dla projektantów posiłkujących się frameworkiem Spring (lub innym kontenerem wspierającym Aspect Oriented Programming) wybór stylu ma wymiar głównie estetyczny, ponieważ przy pomocy AOP można osiągnąć w klasycznym podejściu wszystko to, co oferuje podejście oparte o styl Command.

Jeżeli natomiast nie mamy dostępu do AOP, wówczas styl Commad daje nam znaczną przewagę dzięki możliwości silnego odwracania kontroli – co zobaczymy już niebawem...



Rysunek 1. Architektura Command-query Responsibility Segregation – dwa stosy warstw. Komponent Gate został w szczegółach przedstawiony na Rysunku 2

Styl Serwisowy

Styl Serwisowy omawialiśmy w poprzednich częściach serii, dla przypomnienia odsyłam do Listingu 1. Transakcyjność i bezpieczeństwo zostały wprowadzone poprzez mechanizmy AOP dzięki zastosowaniu Adnotacji interpretowanych przez Spring Framework.

Styl Command & Command Handler

Alternatywą dla każdej z metod Serwisowych jest para: Command i Command Handler. Jak widać na Listingach 2 i 3 styl ten polega na rozdzieleniu klasycznego Wzorca Projektowego Command na 2 klasy. Command – przesyłany przez Tier Kliencki zawiera jedynie parametry żądania. Natomiast odpowiadający mu Command Handler zawiera logikę obsługującą Command. Dzięki temu w odróżnieniu od klasycznego Wzorce Command, aplikacje klienckie nie mają dostępu do kodu logiki aplikacyjnej – ma to oczywisty wpływ na bezpieczeństwo (dekompilacja) i ogólny coupling modułów.

Ogólne zasady tworzenia Command Handlerów pozostają takie same jak tworzenia metod Serwisów Aplikacyjnych omawianych w poprzednich częściach. Przedstawiony na Listingu 3 Command Handler jest odpowiednikiem Serwisu Aplikacyjnego z Listingu 1.

Listing 2. Command `pl.com.bottega.erp.sales.application.commands.SubmitOrderCommand` – polecenie zatwierdzenia zamówienia niosące parametry z warstwy prezentacji

```
@SuppressWarnings("serial")
@Command(unique=true)
public class SubmitOrderCommand implements Serializable{

    private final Long orderId;

    public SubmitOrderCommand(Long orderId) {
        this.orderId = orderId;
    }

    public Long getOrderId() {
        return orderId;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof SubmitOrderCommand) {
            SubmitOrderCommand command = (SubmitOrderCommand) obj;
            return orderId.equals(command.orderId);
        }

        return false;
    }

    @Override
    public int hashCode() {
        return orderId.hashCode();
    }

}
```

Wspólny punkt – miejsce na odwracanie kontroli

Aplikacje klienckie komunikując się z Serwerem, wysyłają Command na wspólny punktu dostępowy, przedstawiony na Listingu 4 oraz zilustrowany na Rysunku 2.

Wspólny punkt dostępowy pozwala na wprowadzenie interesujących i pożytecznych operacji dodatkowych wykonywalnych podczas obsługi polecenia. Przykładowo:

- Sprawdzenie, czy dany Command jest duplikatem (przykładowo atak DOS) – jeżeli tak, to następuje jego odrzucenie. Jako duplikat traktujemy Command oznaczony odpowiednią adnotacją oraz przechowywany w rejestrze ostatnio obsługiwanych poleceń (ostatnio, czyli w czasie x milisekund lub w obszarze x MB pamięci lub w ilości x). Przykładowo dodanie kilkakrotnie tego samego produktu do zamówienia nie będzie traktowane jako duplikat (pozwólmy klientom wydawać pieniądze), ale zatwierdzenie tego samego zamówienia więcej niż raz jest niepożądane. Dzięki mechanizmowi historii poleceń możemy odrzucać niepożądane duplikaty bez potrzeby sięgania do bazy po dane biznesowe.
- Sprawdzenie, czy dany Command jest oznaczony jako asynchroniczny – jeżeli tak, to wówczas odkładamy jego wykonanie np. do Kolejki.

Listing 3. Command `pl.com.bottega.erp.sales.application.commands.handlers.SubmitOrderCommandHandler` – polecenie zatwierdzenia zamówienia niosące parametry z warstwy prezentacji

```
@CommandHandlerAnnotation
public class SubmitOrderCommandHandler implements CommandHandler<SubmitOrderCommand, Void> {

   @Inject
   private OrderRepository orderRepository;

   @Inject
   private InvoiceRepository invoiceRepository;

   @Inject
   private InvoicingService invoicingService;

   @Inject
   private SystemUser systemUser;

   @Override
   public Void handle(SubmitOrderCommand command) {
      Order order = orderRepository.load(command.getOrderId());

      Specification<Order> orderSpecification = generateSpecification(systemUser);
      if (! orderSpecification.isSatisfiedBy(order))
         throw new OrderOperationException("Order does not meet specification", order.getEntityId());

      //Domain logic
      order.submit();
      //Domain service
      Invoice invoice = invoicingService.issuance(order, generateTaxPolicy(systemUser));

      orderRepository.save(order);
      invoiceRepository.save(invoice);

      return null;
   }
```

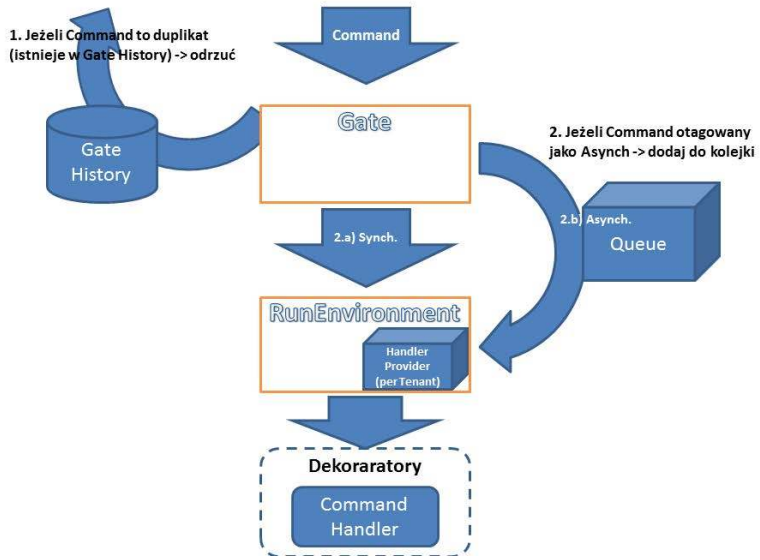
Listing 4. Command pl.com.bottega.cqrs.command.impl.StandardGate – wspólny punkt dostępowy

```
@Component
public class StandardGate implements Gate {

   @Inject
   private RunEnvironment runEnvironment;

   private GateHistory gateHistory = new GateHistory();

   @Override
   public Object dispatch(Object command){
      if (! gateHistory.register(command)){
         //TODO log.info(duplicate)
         return null;//skip duplicate
      }

      if (isAsynchronous(command)){
         //TODO add to the queue. Queue should send this command to the RunEnvironment
         return null;
      }

      return runEnvironment.run(command);
   }

   private boolean isAsynchronous(Object command) {
      if (! command.getClass().isAnnotationPresent(Command.class))
         return false;

      Command commandAnnotation = command.getClass().getAnnotation(Command.class);
      return commandAnnotation.asynchronous();
   }
}
```

Listing 5. Środowisko uruchomieniowe pl.com.bottega.cqrs.command.impl.RunEnvironment pozwalające na wprowadzenie dodatkowych mechanizmów Odwracania Kontroli

```
@Component
public class RunEnvironment {

   public interface HandlersProvider{
      CommandHandler<Object, Object> getHandler(Object command);
   }

   @Inject
   private HandlersProvider handlersProfiver;

   public Object run(Object command) {
      CommandHandler<Object, Object> handler = handlersProfiver.getHandler(command);
      //You can add Your own capabilities here: dependency injection, security, transaction management, logging,
profiling, spying, storing commands, etc
      Object result = handler.handle(command);
      //You can add Your own capabilities here
      return result;
   }
}
```

Rysunek 2. Komponent Gate stanowiący wspólny punkt dostępowy – obsługa Command wysyłanych z aplikacji klienckich



Przyjrzyjmy się teraz Listingowi 5, który przedstawia środowisko uruchomienia polecenia.

Odpowiedzialność Środowiska Uruchomieniowego jest prosta: dopasować do Command odpowiedni ComamndHandler i wykonać go.

Dzięki scentralizowanemu punktowi uruchamiania logiki aplikacji mamy możliwość dokonania całkowitego odwrócenia kontroli, bez użycia kontenera typu Spring:

- Rozpoczęcie transakcji przed uruchomianiem Hadlera oraz jej zatwierdzenie albo wycofanie w zależności od powodzenia działania Handlera,
- Sprawdzenie uprawnień do wykonania Handlera,
- Wstrzyknięcie zależności do Handlera,
- Logowanie poczynań klientów,
- Profilowanie działania systematu,
- Zwrócenie Handlera odpowiedniego dla pracującego w systemie klienta – zagadnienie wersjonowania API i systemy wspierające multi-tenant,
- Dekorowanie Handlerów – wzorzec Dekoratora.

Transakcyjność vs wydajność

Jedną z głównych zasad projektowania skalowalnych systemów z wykorzystaniem DDD jest zapisywanie podczas transakcji jednego Agregatu. Odczytać (w celu wykonania operacji biznesowej) możemy oczywiście kilka Agregatów, ale zapisać – tylko jeden.

Jest to oczywiście jedynie wskazówka, którą możemy złamać w uzasadnionych przypadkach, ale zastanówmy się nad konsekwencją jej stosowania.

Jeżeli zapisujemy w jednej transakcji kilka Agregatów, to nie możemy sobie pozwolić na ich utrwalanie w osobnych bazach danych. Zakładając oczywiście, że transakcje rozproszone (Double Phase Commit) nie są tym, czego potrzebuje architekt projektujący wysokowydajny i skalowalny system.

Zdarzenia

Co zatem zrobić, jeżeli musimy zapisać więcej niż jeden Agregat podczas obsługi Command? Jeżeli wynika to po prostu z wymagań, to rozwiązaniem mogą być Zdarzenia Domenowe, które omawialiśmy w poprzedniej części naszej serii. Przypomnijmy, że Agregaty emitują zdarzenia niosące informacje o istotnych w cyklu życia Agregatów „wydarzeniach biznesowych". Zdarzenia do tej pory stosowaliśmy w celu:

- komunikacji pomiędzy osobnymi Bounded Context,
- odłożeniu wykonania do kolejki operacji, których natychmiastowe wykonanie nie jest krytyczne,
- otwarciu systemu na Pluginy (którymi są Listenery Zdarzeń, jak również Polityki DDD).

Zdarzenia będą nam również potrzebne, aby odświeżać dane w dane do odczytu w drugim stosie naszej Architektury CqRS. Zdarzeń możemy również używać jako Behawioralnego Modelu Danych i zastąpić nimi Relacyjną Bazę w Stosie Write. Zdarzenia mogą być analizowane przez systemy CEP (Complex Events Processing) oraz służyć jako wektory uczące dla Sztucznych Sieci Neuronowych. Ale o tym w kolejnej odsłonie serii.

W sieci

- oficjalna strona DDD: http://domaindrivendesign.org
- wstępny artykuł poświęcony DDD: http://bottega.com.pl/pdf/materialy/sdj-ddd.pdf
- przykładowy projekt – Java (Spring i EJB): http://code.google.com/p/ddd-cqrs-sample/
- przykładowy projekt .NET: http://cqrssample.codeplex.com
- Architektura CqRS http://martinfowler.com/bliki/CQRS.html
- Architektura Porst and Adapters: http://c2.com/cgi/wiki?PortsAndAdaptersArchitecture

Sławomir Sobótka slawomir.sobotka@bottega.com.pl

Programujący architekt aplikacji specjalizujący się w technologiach Java i efektywnym wykorzystaniu zdobyczy inżynierii oprogramowania. Trener i doradca w firmie Bottega IT Solutions. W wolnych chwilach działa w community jako: prezes Stowarzyszenia Software Engineering Professionals Polska (http://ssepp.pl), publicysta w prasie branżowej i blogger (http://art-of-software.blogspot.com).

Michał Bartyzel, Mariusz Sieraczkiewicz

Jak pisać prosty kod?

Jednym z zadań, które sami przed sobą stawiamy, jest wyodrębnianie różnic, które robią różnicę, oraz tworzenie łatwych do zastosowania technik, dzięki którym programiści mogą podnieść swoją efektywność. W tym artykule zaprezentujemy technikę prowadzącą do tworzenia prostego zrozumiałego kodu. Wydaje się ona tak oczywista, że łatwo ją zignorować. Nie daj się zwieść! Już za chwilę dowiesz się, jak kilka banalnych kroków może wprowadzić ogromną zmianę w kodzie.

Dużo się mówi, pisze i czyta o tworzeniu czystego i prostego kodu. Programiści w Twoim zespole, patrząc na kod, potrafią objawowo ocenić, czy jest on wystarczająco prosty, czy nie. Istnieją zdefiniowane *code smells*, po których poznajemy, że z kodem jest coś nie tak. Sęk w tym, że wszystkie te rzeczy dzieją się retrospektywnie. Możemy je orzec dopiero wtedy, gdy kod jest już napisany. Czy nie korzystnie by było od razu tworzyć prosty kod? W miarę nabierania doświadczenia, programiści (choć nie wszyscy :)) wykształcają umiejętność pisania coraz lepszego, czyli prostego, czytelnego, testowalnego kodu. Z umiejętności tej zazwyczaj korzystają intuicyjnie. Okazuje się, że stoi za nią kilka bardzo prostych i łatwych do opanowania kroków, które będziesz mógł szybko wdrożyć w swoim zespole.

PRZYPADKOWA ZŁOŻONOŚĆ

Przez jakiś czas w trakcie szkoleń przeprowadzaliśmy z programistami pewien test. Poprosiliśmy ich o napisanie programu rozwiązującego równanie kwadratowe. W trakcie gdy uczestnicy pracowali, robiliśmy wszystko, co w naszej mocy, aby w tym doświadczeniu zasymulować sytuacje wydarzające się w rzeczywistych projektach. Modyfikowaliśmy więc nieznacznie wymagania, przerywaliśmy prace programistyczne, aranżowaliśmy spontaniczne spotkania itp. Kilka przykładowych rozwiązań zamieszczamy w Listingu 1.

Listing 1. Przypadkowa złożoność pojawiająca się w kodzie

```
//Rozwiązanie 1
public class Calculator {

   public static void main(String[] args) {
      Calculator calc = new Calculator();
      double a = calc.getA();
      double b = calc.getB();
      double c = calc.getC();

      calc.calc(a, b);
      calc.calc2(a, b, c);
   }

   public void calc(double a, double b){
      double x =  -b/a;
      System.out.println("x = " + x);
   }

   public void calc2(double a, double b, double c) {
      double delta = delta(a,b,c);
      if (delta == 0) {
         System.out.println("x = " + -b/2*a);
      } else if (delta > 0) {
         double x1 = (-b - Math.sqrt(delta)) / (2*a);
         double x2 = (-b + Math.sqrt(delta)) / (2*a);
         System.out.println("x1 = " + x1);
         System.out.println("x2 = " + x2);
      } else {
         System.out.println("aaaa");
      }
   }
}

//Rozwiązanie 2
public class Rownanie {

   public static void main(String[] args) {

      final Scanner scanner = new Scanner(System.in);

      double a = scanner.nextDouble();
      double b = scanner.nextDouble();
      double c = scanner.nextDouble();

      double czescUrojona = scanner.nextDouble();
      double czescRzeczywista = scanner.nextDouble();

      double delta;
      double pierwiastek;

      delta = (b * b) - (4 * c * a);
      pierwiastek = Math.sqrt(delta);

      if (a == 0) {
         System.out.println("Równanie liniowe");
         System.out.println("y = " + b + "*x+" + c);
         System.out.println("x = " + ((-c) / b));
      }

      if (delta < 0) {
         System.out.println("delta <0");
      }

      if (delta == 0) {
         System.out.println("Jeden pierwiastek x = " + ((-
b) / (2 * a)));
      }

      double x1 = 0;
      double x2 = 0;

      if (delta > 0) {
         x1 = (-b + pierwiastek) / (2 * a);
         x2 = (-b - pierwiastek) / (2 * a);

         System.out.println("Dwa pierwiastki");
         System.out.println(+x1);
         System.out.println(+x2);
      }
   }
}

//Rozwiązanie 3
public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      Complex[] calculate = calculate(scanner.
nextDouble(), scanner.nextDouble(), scanner.
nextDouble());
      for (int i = 0; i < calculate.length; i++) {
         System.out.println(calculate[i]);
      }
   }

   public static Complex[] calculate(double a, double b,
double c){
      if(a == 0){
         return new Complex[] {new Complex(calculate(b,
c))};
      }else{
         return calculateQuatr(a, b, c);
```

```
        }
    }
    public static double calculate(double a, double b) {
        return -b/a;
    }

    public static Complex[] calculateQuatr(double a,
double b, double c) {
        double delta = calculateDelta(a, b, c);

        if(delta < 0){
            return calculateComplex(a,b,delta);
        }else {
            double x1 = (-b - Math.sqrt(delta))/(2*a);
            double x2 = (-b + Math.sqrt(delta))/(2*a);
            return new Complex[] {new Complex(x1) , new
Complex(x2)};
        }

    }


    private static Complex[] calculateComplex(double a,
double b, double delta) {
        Double re = -b/2*a;
        Double im = Math.sqrt(-delta)/(2*a);
        return new Complex[] {new Complex(re, im) , new
Complex(re, -im)};
    }

    public static double calculateDelta(double a, double
b, double c) {
        return b*b - 4*a*c;
    }

}
```

Dlaczego rozwiązania są tak różne? Czy nie wydaje Ci się, że niektóre z nich są „udziwnione" ponad konieczność? Czy rozwiązania z Listingu 1 można nazwać prostymi? Nasze pierwsze spostrzeżenie było takie, że stworzony kod jest nieproporcjonalnie skomplikowany w stosunku do problemu, który miał rozwiązywać. O tego typu złożoności w kodzie często mówi się, że jest przypadkowa. Powstaje nie w sposób planowy, lecz w losowy. Jej przeciwieństwem jest złożoność intencjonalna, wynikająca z natury zagadnienia, którym się zajmujemy. Złożoność intencjonalna jest wpisana w naszą pracę i dzięki niej tworzymy oprogramowanie, którego ludzie chcą używać. Złożoność przypadkowa jest jak chwast, który może zdominować wartościowy kod.

W jaki sposób przypadkowa złożoność powstaje? Naszym zdaniem jest kilka kluczowych powodów. Być może to niewiarygodne, lecz można napisać dużą ilość kodu bez zastanawiania się, do czego konkretnie będzie on używany i jaką realną wartość ma zapewnić klientowi. Zdarza się, że gdy programista otrzymuje *ticket* do zrealizowania, to po prostu go realizuje, bez chociażby krótkiej analizy. W ten sposób **rozwiązywane są lokalne problemy**, bez odniesienia do szerszego kontekstu. Skutkiem takiego postępowania jest kod, w którym brakuje przemyślanego algorytmu realizującego daną funkcjonalność. W jego miejsce powstaje kod, który koncentruje się na **obsłudze kolejnych przypadków szczególnych** tej funkcjonalności (*if-else programming*). Każdy kolejny wiersz kodu tylko coraz bardziej komplikuje zagadnienie, a komentarze //FIXME, //XXX, //workaround są na porządku dziennym.

Inną powszechnie występującą przyczyną powstawania przypadkowej złożoności jest **zbyt wczesne abstrahowanie**. Można to utożsamiać z terminem *overengineering*, lecz my lubimy to nazywać „przekombinowanymi" rozwiązaniami. Gdy przypatrzysz się Listingowi 1, to zauważysz, że rozwiązanie trzecie jest właśnie przekombinowane. Powodem może być nieco literalne stosowanie obiektowości. Mając w głowie uproszczoną zasadę „rzeczownik = klasa, czasownik = metoda", programiści upychają klasy wszędzie, gdzie się da, dla zasady opakowując nawet pojedyncze pola. Z pewnością domyślasz się, że chodzi o klasę Complex. To, że akurat programista pisze kod, który oblicza rozwiązania zespolone równania, nie oznacza, że od razu powinna powstać klasa reprezentująca liczbę zespoloną. Nie twierdzimy, że klasa ta nie powinna powstać *w ogóle*, lecz *jest jeszcze za wcześnie*, aby powstała. Jeszcze nie wiadomo, czy jest potrzebna.

MAŁE DECYZJE Z WIELKIMI KONSEKWENCJAMI

Wielkie problemy z kodem biorą się z małych codziennych zaniedbań. Pisząc kod, w każdej chwili programista podejmuje drobne decyzje: *Jaką nazwę nadać metodzie? Czy wyodrębnić klasę? Czy podzielić metodę na części?* Tych decyzji jest tysiące, miliony. Na bazie tych decyzji podejmowane są kolejne decyzje, również przez innych członków zespołu. Kolejni programiści analizują kod z repozytorium i dochodzą do wniosku, że jeśli ktoś zastosował dane podejście, to najwyraźniej jest ono dobre, a następie powielają zaobserwowany schemat w innych miejscach systemu. Czy możesz zatem przewidzieć, jakie konsekwencje za miesiąc spowoduje „if", który właśnie napisałeś? Oczywiście, że nie. Zbyt wiele możliwości byłoby do przeanalizowania.

Skłaniaj ludzi do podejmowania lepszych decyzji

Najbardziej skuteczną rzeczą, którą możesz w takiej sytuacji zrobić, jest skłanianie ludzi do podejmowania lepszych decyzji odnośnie kodu, który tworzą. Do nadawania lepszych nazw, do pisania czytelniejszych metod, do projektowania bardziej zwartych metod. Prawdą jest, że wielkie problemy z kodem biorą się z małych codziennych zaniedbań. Jednocześnie prawdą jest, że dobrej jakości kod powstaje w wyniku małych codziennych ulepszeń. Stałe egzekwowanie elementarnych zasad higieny programistycznej zazwyczaj daje lepsze rezultaty niż spektakularne refaktoryzacje raz do roku.

Złożoność w kodzie można albo ignorować, albo adresować – czyli podejmować aktywności zmierzające do jej opanowania. W Tabeli 1 zestawiliśmy te dwie możliwości. Znajdziesz tam działania, które możesz podjąć, aby opanować powstającą złożoność oraz konsekwencje, które będą skutkiem jej ignorowania.

	Złożoność intencjonalna	Złożoność przypadkowa
Adresując wykorzystujemy...	• Modelowanie • Techniki pracy z kodem • Wzorce projektowe	• Refaktoryzację • Przeglądy kodu
Ignorowanie prowadzi do...	• Trudności komunikacyjnych z biznesem • Architektury nieprzygotowanej na zmiany w procesie biznesowym • Zmiany złożoności intencjonalnej w przypadkową	• Zniechęcenia programistów • Problemów z wydajnością • Dużej ilości zadań „na wczoraj" • Rosnących kosztów utrzymania

Tabela 1. Aktywności wykorzystywane do adresowania złożoności oraz konsekwencje jej ignorowania

PROSTA TECHNIKA

W kolejnym kroku doświadczenia z równaniem kwadratowym prosiliśmy uczestników szkolenia, aby zapisali na kartce kroki algorytmu rozwiązującego równanie kwadratowe, które jest im tak dobrze znane. Okazało się, że po kilku minutach każdy z programistów otrzymał niemal dokładnie ten sam efekt (Listing 2).

Listing 2. Kroki algorytmu w pseudokodzie

```
//1. Wczytaj a
//2. Wczytaj b
//3. Wczytaj c
//4. Jeśli a == 0 to jedno rozwiązanie -c/b
//5. Oblicz deltę
//6. Jeśli delta == 0 to pierwiastek podwójny -b/(2*a)
//7. Jeśli delta > 0 to dwa pierwiastki rzeczywiste
   // a. x1 = -b/( 2*a ) - Math.sqrt( delta )/( 2*a )
   // b. x2 = -b/( 2*a ) + Math.sqrt( delta )/( 2*a )
//8. Jeśli delta < 0 to dwa pierwiastki zespolone
   // a. -b/( 2*a ) - iMath.sqrt( Math.abs( delta ) )/ (
2*a )
   // b. -b/( 2*a ) + iMath.sqrt( Math.abs( delta ) )/ (
2*a )
```

Ciekawe, że kilka chwil z kartką, zamiast natychmiastowego rzucania się do programowania, sprawia, że luźne pomysły nabierają porządku. Najciekawszy jednak efekt ujawnił się, gdy poprosiliśmy uczestników, aby zapisali w języku programowania **dosłownie** to, co przed chwilą napisali w pseudokodzie.

Listing 3. Kod, który powstał (u każdego z uczestników) na podstawie pseudokodu

```
public class EquationSimpleSolution {

   public static void main(String[] args) {

      System.out.println("az^2 + bz + c = 0");
      System.out.println();

      Scanner scanner = new Scanner(System.in);

      double a = readParamter(scanner, "a");
      double b = readParamter(scanner, "b");
      double c = readParamter(scanner, "c");

      if (a == 0) {
         double x = -c / b;
         System.out.println("[Równanie liniowe] x = " + x);

         return;
      }

      double delta = b * b - 4 * a * c;
      if (delta == 0) {
         double x0 = -b / (2 * a);
         System.out.println("[Pierwiastek podwójny] x1 =
x2 = " + x0);

         return;
      }

      if (delta > 0) {
         double x1 = -b / (2 * a) - Math.sqrt(delta) / (2 * a);
         double x2 = -b / (2 * a) + Math.sqrt(delta) / (2 * a);

         System.out.print("[Dwa pierwiastki rzeczywiste] ");
         System.out.print("x1 = " + x1 + " x2 = " + x2);

         return;
      }

      if (delta < 0) {
         double re = -b / (2 * a);
         double im = Math.sqrt(Math.abs(delta)) / (2 * a);

         System.out.print("[Dwa pierwiastki zespolone] ");
         System.out.print("x1 = " + re + " + " + im + "i");
         System.out.print("x2 = " + re + " - " + im + "i");
      }
   }

   private static double readParamter(final Scanner
scanner, String param) {
      System.out.println(param + " = ");
      double a = Double.parseDouble(scanner.nextLine());
      return a;
   }
}
```

Zwróć uwagę, że kod na Listingu 3 jest minimalistyczny. Nie zawiera na razie żadnych obiektów, gdyż nie są tam one jeszcze potrzebne. Kod ten jest czytelny i zrozumiały dla każdego, kto zna równania kwadratowe. Złożoność tego kodu jest adekwatna do złożoności problemu, który on rozwiązuje. Ten kod jest prosty.

Hipotezy

Na podstawie wyżej opisanego i wielokrotnie powtarzanego doświadczenia postawiliśmy następujące hipotezy:

- Rozpoczynanie programowania od pisania kodu sprawia, że nasza perspektywa zawęża się do konkretnego zadania i gubimy szerszy kontekst całego zagadnienia;
- Napisanie w pierwszej kolejności kroków algorytmu w postaci zdań sprawia, że programista łatwiej potrafi przeanalizować dalsze konsekwencje rozwiązania, które tworzy;
- Kod tworzony na podstawie napisanych kroków algorytmu jest prostszy niż kod tworzony „z głowy” i rzadziej powstaje w nim przypadkowa złożoność.

Na podstawie wymienionych hipotez sformułowaliśmy prostą technikę (zgodnie z obietnicą złożoną na wstępie). Jest ona tak oczywista, że bardzo często bywa ignorowana i uznawana za niegodną doświadczonego programisty. Technika ta ma dwa kroki:

- Napisz zadaniami kolejne kroki algorytmu
- Zapisz w kodzie **dosłownie** to, co zostało napisane w pseudokodzie

Mimo swojej prostoty siłą tej techniki jest skutek jej stosowania. Konsekwentne jej używanie zapobiega powstawaniu przypadkowej złożoności w kodzie i pomaga tworzyć prosty i czytelny kod.

Michał Bartyzel,
Mariusz Sieraczkiewicz
m.bartyzel@bnsit.pl,
m.sieraczkiewicz@bnsit.pl

Trenerzy i konsultanci w firmie BNS IT. Badają i rozwijają metody psychologii programowania, pomagające programistom lepiej wykonywać ich pracę. Na co dzień Autorzy zajmują się zwiększaniem efektywności programistów poprzez szkolenia, warsztaty oraz coaching i trening.

bottega
IT SOLUTIONS

Praktycy dla Praktyków
Szkolenia i doradztwo

www.bottega.com.pl

Sławomir Sobótka

Główny trener, programujący architekt specjalizujący się w Java EE, prezes SSEPP

SZKOLENIA ›
Java EE, Spring, Seam, EJB, Architektura, Testowanie, .NET

DORADZTWO ›
Architektura, DDD, Audyty, Dobór technologii.

MENTORING ›
Asysta HR, Rekrutacja, Coaching teamu.

DEVELOPMENT ›
Kickoff projektów, wsparcie projektowe i architektoniczne.

SZKOLENIA

Kiedy potrzebujesz rozwinąć nowe umiejętności.

KATALOG SZKOLEŃ

Java SE, Java EE
Podstawy Java, EJB, JPA, JSF, Architektura. ›

Spring – aplikacje webowe.
Kompleksowy stos oraz architektura. ›

.NET: C#, LINQ, web.
Wzorce, Arch., Profilowanie, Testowanie ›

Inżynieria oprogramowania.
Architektura, Wzorce, DDD, Testowanie. ›

SZKOLENIA ++

Szkolenia++ to połączenie **rzetelnych** szkoleń merytorycznych z team buildingiem.

Oferujemy szkolenia połączone z **imprezami integracyjnymi** – liczne atrakcje dla uczestników w ciekawych miejscach w Polsce.

DORADZTWO

Kiedy umyka Ci problem.

ZAKRES USŁUG

- Dobór technologii i architektury
- Optymalizcja procesu wytwórczego
- Modelowanie biznesowe, DDD
- Doradztwo IT dla biznesu
- Audyty i ekspertyzy
- Asysta w procesie rekrutacji

SŁUŻYMY WIEDZĄ I DOŚWIADCZENIEM

W odpowiedzi na typowe wyzwania w procesie tworzenia oprogramowania oferujemy szereg **dobrze zdefiniowanych** usług doradczych.

Dzięki nim skupisz się na kluczowych aspektach i ruszysz z miejsca.

MENTORING

Kiedy chcesz szybko dojść do celu.

ZAKRES USŁUG

- Coaching zespołu
- Praca nad koncepcją rozwiązania
- Pair Programming
- Najlepsze praktyki i typowe pułapki

DLACZEGO MENTORING?

Mentoring jest uzupełnieniem szkoleń. Pozwala na **dopasowanie** zakresu wiedzy do specyficznego kontekstu.

Najszybciej uczymy się przez naśladowanie a zjawisko to może zostać spotęgowane przez obecność eksperta, który na bieżąco **wyjaśnia** podejmowane decyzje.

bottega - [z j. włoskiego] *pracownia; w renesansie było to miejsce gdzie spotykał się humanizm, sztuka i rzemiosło*

www.bottega.com.pl

Jesteśmy członkiem: 10 01 SSEPP STOWARZYSZENIE SOFTWARE ENGINEERING PROFESSIONALS POLSKA

To nie sen, to JAVA
Grupa Allegro czeka na Ciebie!

Programujesz w JAVA?
Wejdź na kariera.allegro.pl i skorzystaj z jednej z naszych ofert:

- **Programista J2EE**
  (Warszawa, Poznań, Toruń)
- **Młodszy Programista J2EE**
  (Warszawa, Poznań, Toruń)
- **Programista Java w zespole**
  Utrzymania Aplikacji
- **Programista Java**
  (Warszawa, Poznań)

Znajdziesz nas również na
facebook.com/KarierawGA

allegro group

DOŁĄCZ DO NAS

allegro otomoto.pl otoWakacje.pl otodom.pl tablica.pl PLATNOSCI
CITEAM CENEO cokupić.pl markafoni iStore.pl oferia.pl mojeauto.pl Bankier.pl