Software Developer's JOURNAL
new ideas & solutions for professional programmers
NR 2 WRZESIEŃ 2013 ISSN 1734-3917

Testowanie Oprogramowania z punktu widzenia ekspertów

- SpiraTeam – ALM dla wymagających
- Wdrożenie metodyki testów w oparciu o normę ISO/IEC 29119

Software Developer's JOURNAL
new ideas & solutions for professional programmers

Miesięcznik Software Developer's Journal jest wydawany przez NPRACA Sp. z o.o.

Redakcja: *sdjpl@sdjournal.pl*
Kierownik produkcji: Andrzej Kuca

Adres korespondencyjny:
NPRACA Spółka z o.o.
ul. 3 Maja 46, 72-200 Nowogard, Polska
Oddział w Warszawie:
ul. Postępu 17 D, 02-676 Warszawa, Polska
www.sdjournal.pl
cooperation@software.com.pl
Dział reklamy: *adv@software.com.pl*

Osoby zainteresowane współpracą prosimy o kontakt:
cooperation@software.com.pl
Skład i łamanie:
Digital Concept
www.digitalconcept.pl *admin@digitalconcept.pl*

Współpraca redakcyjna:
Marcin Michalski
Adam Końca (Sagiton - implementing ideas)

ALM
Application Lifecycle Management

ALM (Application Lifecycle Management) – trzy litery, które jakiś czas temu przedefiniowały znany do tej pory kontekst narzędzi wspierających cykl życia oprogramowania. Narzędzia ALM przejęły rozproszoną dotychczas kontrolę nad zarządzaniem wymaganiami, cyklem życia oraz testowaniem oprogramowania w jednym, spójnie działającym środowisku.

Dowiesz się:

- Czym jest ALM i jak może być wspierany narzędziami.
- Jakie funkcjonalności powinno posiadać narzędzie klasy ALM na przykładzie SpiraTeam.

Powinieneś wiedzieć:

- Czym jest cykl życia oprogramowania.
- Znać podstawowe metodyki wytwarzania oprogramowania.

Współcześnie tworzone systemy informatyczne są często bardzo złożone. W trakcie ich wytwarzania pojawiają się zmiany wynikające zarówno z rewizji wymagań, ze zmian podejścia do procesu wytwórczego, jak i nierzadko ze zmian technologii na bardziej nowoczesną. Wszystkie te aspekty sprawiają, że prowadzenie projektów informatycznych bez odpowiedniego wsparcia narzędziowego staje się praktycznie niemożliwe. Wybranie odpowiedniego narzędzia jest równie dużym wyzwaniem, a sam wybór nie rozwiązuje wszystkich problemów. Dopiero wdrożenie i dostosowanie do procesów obowiązujących w organizacji, jak i późniejsze używanie narzędzia w praktyce, weryfikuje z czasem słuszność naszego wyboru.

Czym zatem jest to odpowiednie narzędzie ALM i czy musi kosztować fortunę?

Główne cechy, jakie powinny być brane pod uwagę przy wyborze, to przede wszystkim:

- Skalowalność i łatwość konfiguracji narzędzia – każde rozwiązanie musi zostać dostosowane do indywidualnych potrzeb klienta.
- Jednolitość – każdy z członków danego projektu powinien działać w ramach jednego, wspólnego procesu, co pozwala funkcjonować w sposób usystematyzowany, spójny i zrozumiały dla całego zespołu. Wszystkie artefakty procesu wytwórczego powinny być zarządzane w jednym repozytorium oraz być ze sobą powiązane i wersjonowane.
- Łatwość użytkowania – intuicyjny i prosty interfejs możliwy do uruchomienia w wielu przeglądarkach bez konieczności instalacji. Ta cecha sprawia, że użytkownicy szybko przystępują do korzystania z narzędzia w wygodnym dla siebie środowisku.
- Nieskomplikowana integracja z zewnętrznym oprogramowaniem – obecnie jest wiele narzędzi wspierających prace w procesie wytwórczym, chociażby narzędzia do automatyzacji testów, narzędzia do projektowania wymagań czy też systemy IDE. Możliwość integracji z nimi zapewnia zarządzanie wszystkimi procesami za pośrednictwem systemu ALM.
- Pełne wspieranie procesu wytwórczego – od zbierania wymagań, poprzez zadania deweloperskie, tworzenie i wykonywanie testów, tworzenie zgłoszeń incydentów, aż po monitorowanie i raportowanie postępu prac w wielu projektach jednocześnie.

Zdaniem autora świetnym wyborem, sprawdzającym się w praktyce, posiadającym wszystkie wymienione wyżej cechy jest SpiraTeam firmy Inflectra. Dodatkowymi atutami przemawiającymi za tym wyborem są przejrzysty model licencjonowania i wsparcia dla użytkowników narzędzia oraz bardzo atrakcyjna cena w porównaniu do innych sztandarowych produktów konkurencji dostępnych na rynku.

inflectra™

SOFLAB

Soflab Technology

jako Wyłączny Partner

firmy **Inflectra**

w Polsce realizuje kompleksowe projekty wdrażania rozwiązań i metodyk zarządzania testami i wytwarzania oprogramowania.

Used by over 1,500 customers worldwide...

SpiraTest® - End The Testing Chaos!

Why spend money on standalone requirements management, bug tracking and testing tools?

SpiraTest® provides a complete Quality Assurance solution that manages your Requirements, Test Cases, Test Sets, Bugs and Issues in one environment with complete traceability from inception to completion.

> **Learn More** | **Download Free Trial**

„Narzędzie SpiraTeam jest przez nas wykorzystywane obecnie w kilku dużych projektach. Integracja z Active Directory (zapewniająca dostęp dla użytkowników) oraz z TFS (umożliwiająca rejestrację incydentów z ich przekazaniem bezpośrednio do TFS) są dla nas bardzo ważne. Mozliwość pracy wielu testerów na jednym scenariuszu testowym daje komfort pracy i znacznie przyspiesza proces testowania a duży wybór raportów umożliwia bieżące śledzenie postępów testów"

Piotrek Ryder
Kierownik Wydziału Testów, Departament Rozwoju Usług IT
Centrala BGŻ S.A.

Profesjonalne narzędzia ALM pozwalają na własne definiowanie obiektów i przepływów w ramach projektu. Również SpiraTeam udostępnia użytkownikom dowolność w projektowaniu work-flow, przy czym jest to proces bardzo przejrzysty i uwzględniający wiele aspektów takich jak np. dodawanie dodatkowych pól opisujących obiekty uczestniczące w procesie czy też konfiguracja notyfikacji mailowych dla użytkowników.

Model procesu może zakładać zastosowanie typowo zwinnych metodyk jak SCRUM, Agile z charakterystycznymi dla nich elementami (iteracje, sprinty, scorecard). Narzędzie wspiera także projekty prowadzone zgodnie z metodykami klasycznymi.

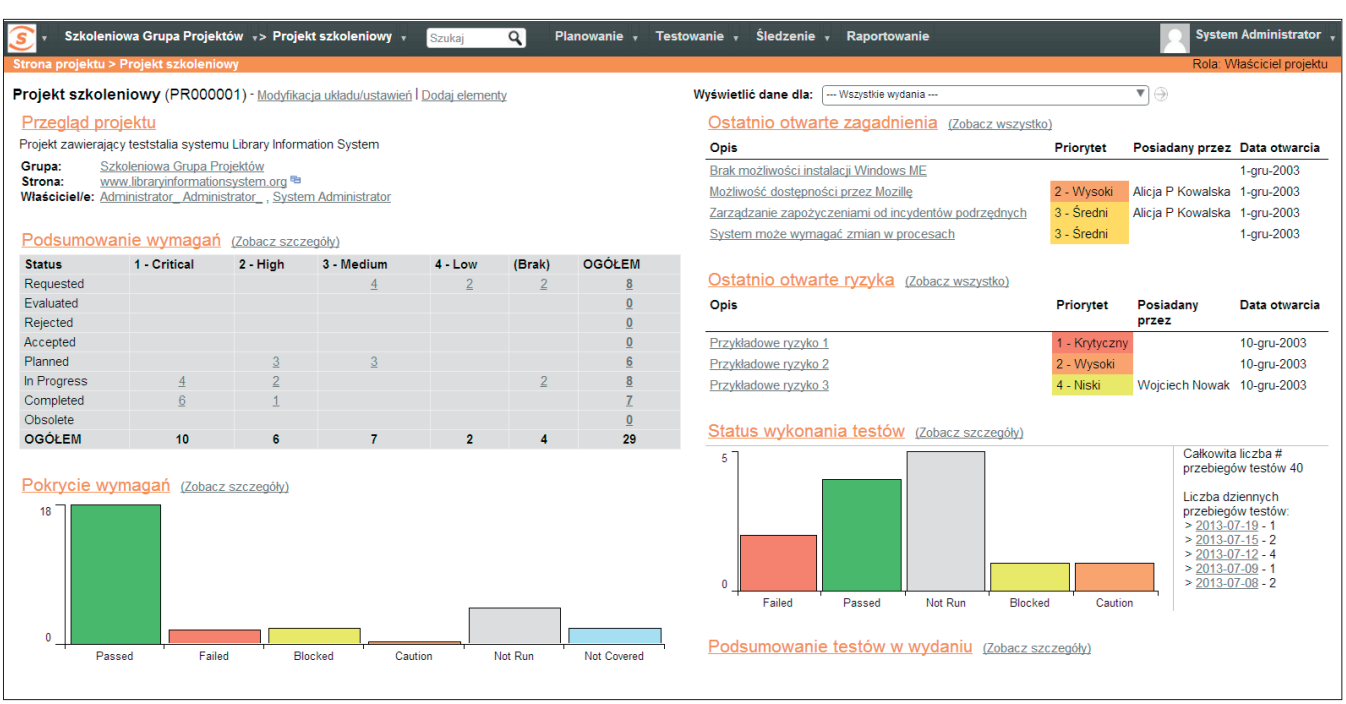
SpiraTeam pozwala na prowadzenie wielu projektów jednocześnie, w różnych metodykach i konfiguracjach.

Dodatkowo zapewnia podgląd najważniejszych metryk w postaci dashboard'u, zarówno na poziomie projektu, jak i zdefiniowanej grupy projektów. Jest to jedna z najbardziej docenianych funkcjonalności systemu. Dzięki temu Project Manager ma możliwość podglądu statusu całego portfolio swoich projektów na jednym ekranie.

Bardzo ciekawą funkcjonalnością jest możliwość zdekomponowania wymagań na powiązane z nimi niskopoziomowe zadania, które to następnie można osobno szacować oraz przypisywać do wykonania poszczególnym członkom zespołu projektowego. Dla każdego z zadań można śledzić postęp, a ich szacowany czas realizacji sumuje się do całkowitego czasu implementacji wymagania.

Wymagania

SpiraTeam jest również repozytorium wymagań. Przejrzysta, hierarchiczna struktura drzewiasta umożliwia przechowywanie wymagań w czytelnej formie „od ogółu do szczegółu" tzn. wysoko poziomowe wymaganie biznesowe jest wymaganiem grupującym, a wymagania techniczne są mu podporządkowane. Oprócz dodawania dekompozycji wymagań na pomniejsze zadania, narzędzie klasy ALM daje również powiązania wymagań z przypadkami testowymi, tworząc połączenie zwane pokryciem testami



Rysunek 1. *Indywidualny panel kontrolny projektu*

(test coverage). Dzięki temu uzyskujemy jasny wgląd w to w jakim stopniu wymaganie zostało przetestowane i z jakim skutkiem.

Jeśli podczas testów został zgłoszony defekt, developer, tester oraz inni interesariusze procesu mogą prześledzić jego wpływ na wymagania. Dzięki temu łatwo jest przeprowadzić analizę wpływu. SpiraTeam automatyzuje proces raportowania pokrycia wymagań. Jeśli wszystkie przypadki testowe powiązane z danym wymaganiem zostały wykonane z wynikiem pozytywnym, to wymaganie zostanie odznaczone jako zweryfikowane pozytywnie a dodatkowo jego status zmieni się na „Zaakceptowane". Pozwala to w łatwy sposób, w każdym momencie trwania testów ocenić poziom zaufania dla testowanego oprogramowania.

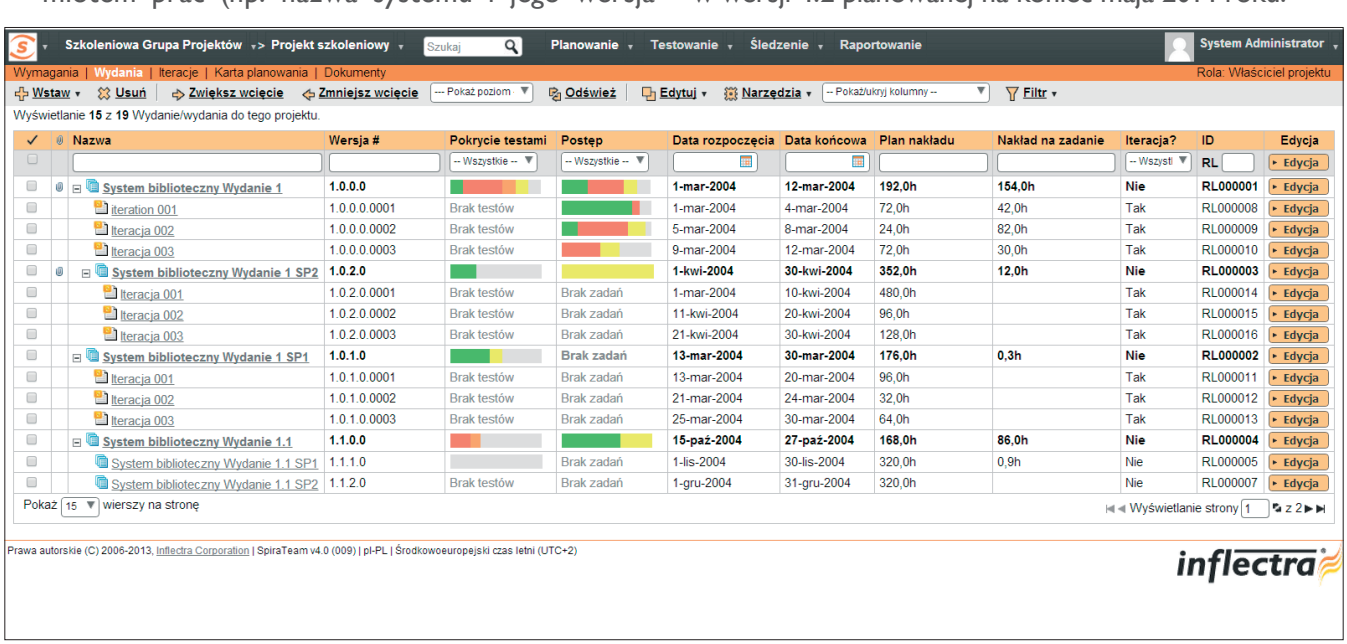
"If you fail to plan, you are planning to fail!" - Benjamin Franklin

Planowanie i ustalanie zakresu dla bieżących oraz przyszłych przedsięwzięć jest kluczowym założeniem zarządzania cyklem życia aplikacji. Narzędzie klasy ALM powinno wspierać planowanie zakresu pracy w jednostkach czasowych (w zależności od metodyki i techniki mogą być to sprinty, sesje, iteracje…) tak, aby cały zespół i wszyscy interesariusze wiedzieli w każdym momencie pracy, czy aktualne przedsięwzięcie zostanie zrealizowane, a może jest ryzyko niepowodzenia. Moduł planistyczny jest jednym z filarów narzędzia SpiraTeam. Został on przejęty w całości ze SpiraPlan.

W module użytkownik może zdefiniować wydania, które może podzielić na iteracje. Główne atrybuty wydania to:

- Nazwa – powinna jasno określać co jest przedmiotem prac (np. nazwa systemu i jego wersja rozwojowa) tak, aby interesariusze mogli łatwo zidentyfikować czego dotyczy pokazywany na spotkaniu projektowym raport.
- Daty graniczne – określenie daty początku i daty końca wydania umożliwia zaplanowanie realizacji zakresu wdrożenia w określonym czasie.
- Zakres – w jego skład wchodzą:
  - Incydenty – zgłoszone podczas bieżącego wydania a także incydenty, które nie zostały rozwiązane w poprzednim wydaniu a osoba odpowiedzialna za zakres ustaliła, że na przykład w tym wydaniu powinny zostać rozwiązane.
  - Wymagania i Zadania – zadania deweloperskie, które mogą mieć pokrycie w wymaganiach.
  - Przypadki testowe – wyznaczone przypadki, które muszą zostać wykonane by zweryfikować czy wydanie może zostać przekazane do produkcji. Należy pamiętać o tym, że przypadki testowe są w ścisłej relacji z wymaganiami (pokrycie wymagań).
  - Pojemność (capacity) – umożliwia managerowi ustalenie ilu pracowników dostępnych jest do realizacji zadań w wydaniu i iteracjach. W korelacji z datami granicznymi umożliwia to planowanie operacyjne i unikanie przeładowania pracą (dni wolne od pracy i dostępność godzinowa jest konfigurowalna).

Znacznie łatwiejsze byłoby planowanie kalendarza wydań przy pomocy wykresu GANT'a znanego choćby z MS Projecta. Podobnie jak autor, pomyśleli również użytkownicy masowo wysyłając zapotrzebowanie na taką funkcjonalność. Według Road Mapy dostarczonej przez producenta, takie usprawnienie będzie dostępne w wersji 4.2 planowanej na koniec maja 2014 roku.



Rysunek 2. *Moduł wydań*

Z doświadczenia autora wynika, iż planowanie wydań i iteracji wymaga pewnego poziomu samodyscypliny od managera, ale w późniejszych etapach projektu ten wysiłek się opłaca. Interesariusze widzą postępy prac oraz ich status bezpośrednio w narzędziu, na indywidualnych panelach kontrolnych.

Dobrze zdefiniowane wydania i iteracje w dużym stopniu ułatwiają planowanie operacyjne. Dzięki modułowi opartemu o Kanban Board, manager może szybko przydzielać i przeplanować zadania pomiędzy wydaniami i iteracjami. Dostępność zasobów podczas planowania jest cały czas monitorowana.

Moduł ten pozwala zespołowi na przejrzystą wizualizację wymagań, zadań, incydentów przypisanych do konkretnej iteracji.

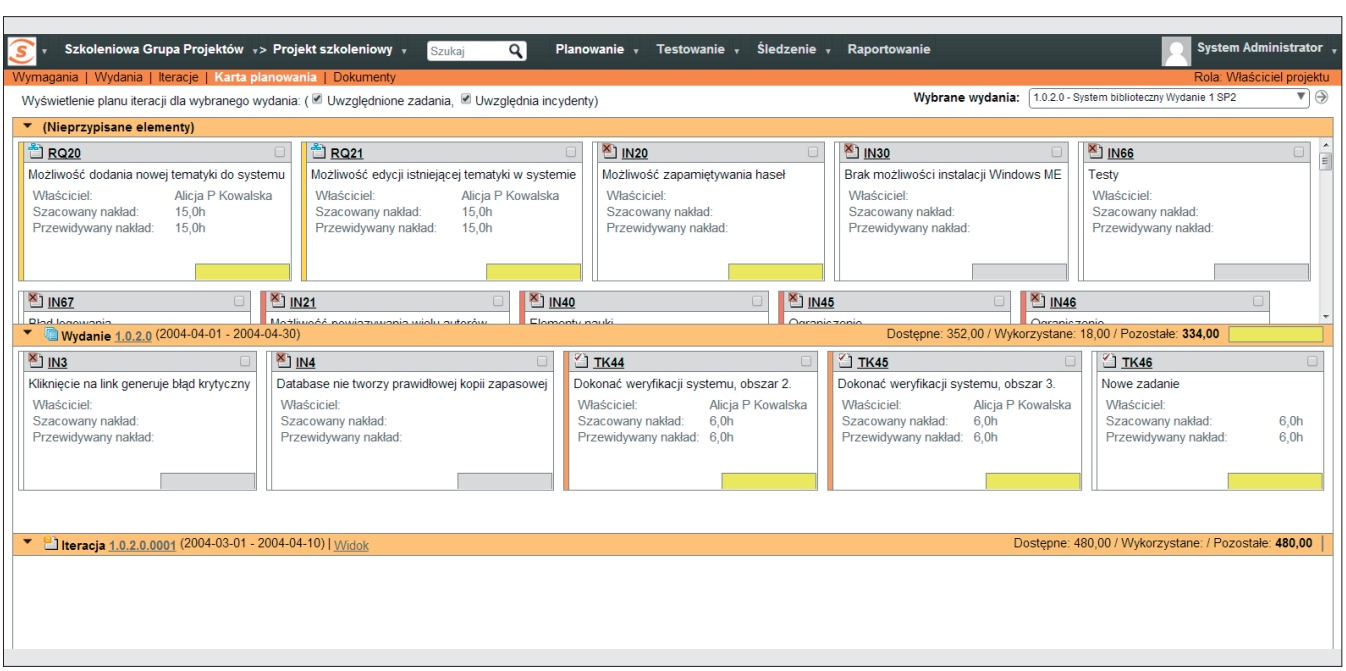
Dzięki temu szybko i precyzyjnie można dotrzeć do artefaktów przypisanych do np. konkretnego dewelopera w danej iteracji i w razie potrzeby przypisać kolejne zdania lub też w przypadku nadmiaru zadań – przypisać je innej osobie. Dzięki metodzie drag-and-drop oraz kolorowym kodom, efekt przejrzystości i jednoznaczności został zmaksymalizowany. To rozwiązanie znakomicie sprawdza się w zespołach pracujących w zwinnej metodyce, przy dużej liczbie zmian oraz wysokim tempie pracy.

SpiraTeam posiada także funkcjonalność zarządzania zasobami. Dzięki temu rozwiązaniu planowanie pracy członków zespołu odbywa się również w narzędziu przy wykorzystaniu danych z modułu planowania. Każda osoba ma własną kartę pracy, w której raportuje czas pracy/poświęcony na realizację zadań.

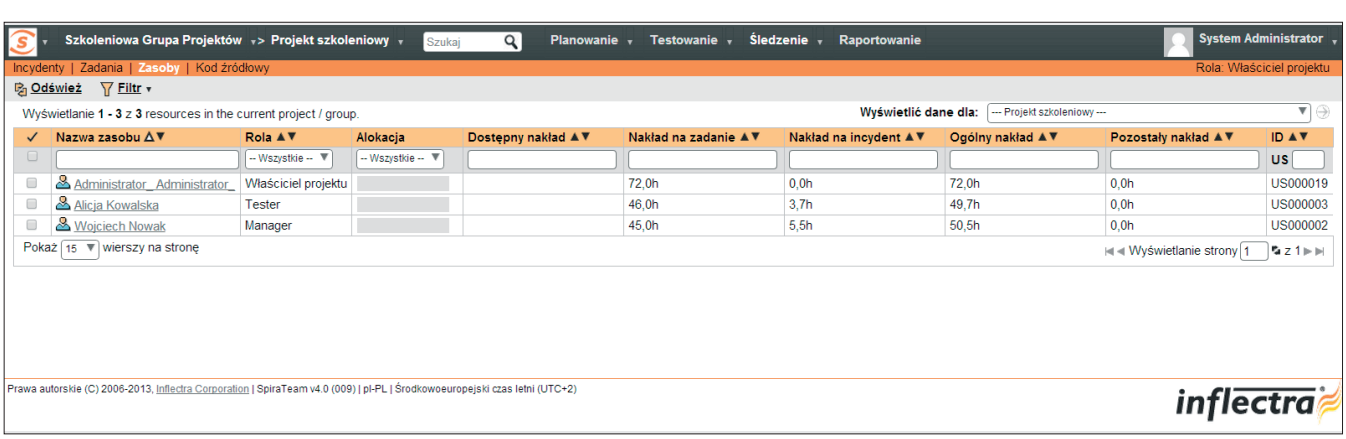
SpiraTeam posiada także możliwość dodawania załączników plikowych, linków wewnętrznych lub zewnętrznych, do każdego artefaktu

Wbudowana funkcjonalność „screen capture" umożliwia szybkie dodanie zrzutu ekranu do tworzonego zgłoszenia incydentu.

Umożliwia to wzbogacanie artefaktów o dodatkowe, bardziej szczegółowe i użyteczne informacje.



Rysunek 3. *Karta planowania*



Rysunek 4. *Karta zasobów projektowych*

Testowanie

Organizacje, które nie przykładają dużej wagi do zapewnienia jakości, najczęściej używają następujących narzędzi do wspierania procesu testowego:

- Zarządzanie wymaganiami na współdzielonych zasobach dyskowych.
- Planowanie testów, katalogowanie i utrzymanie przypadków testowych, raportowanie wykonania testów manualnych oraz automatycznych przy użyciu MS-Word oraz MS-Excel.
- Zarządzanie defektami i ich zgłaszanie przy pomocy poczty elektronicznej lub prostego bug trackera.

W konsekwencji:

- Procesy są procesami ad-hoc i nie są powtarzane w innych projektach.
- Brakuje powiązań pomiędzy Przypadkami Testowymi, Wymaganiami, Defektami i innymi artefaktami, co uniemożliwia ocenę kiedy rzeczywiście możemy zakończyć testowanie.
- Mierzenie postępu prac i produktywności w czasie wykonywania testów pochłania mnóstwo czasu i jest bardzo skomplikowane.
- Ciężko jest przygotować raport, dzielić się informacjami w projekcie oraz zbierać metryki w czasie rzeczywistym.
- Brakuje jednego spójnego repozytorium dla wszystkich poziomów i typów testów.
- Wykorzystanie danych historycznych jest problematyczne i czasochłonne.

Przy projektowaniu SpiraTeam wszystkie powyższe aspekty zostały uwzględnione, stąd rozwiązanie to stanowi kompletny system ALM, wspierający w szerokim zakresie prace zespołu testowego. Od tworzenia przypadków testowych na podstawie wymagań, poprzez grupowanie ich w scenariusze testowe, przebiegi testowe, aż po zarządzanie zgłoszeniami incydentów. Wszystkie te elementy można ze sobą wiązać i śledzić stworzone w ten sposób zależności w postaci pełnego raportu powiązań (tracebility matrix).

„SpiraTeam okazał się dla nas optymalnym rozwiązaniem wspierającym zespół projektowy. Skutecznie wspiera rozproszone środowisko wytwórcze, w którym pracujemy. Użytkownicy docenili także przejrzystość interfejsu aplikacji i dostępność polskiej wersji językowej. Narzędzie znacząco usprawniło komunikację z dostawcami. Dodatkowo nawet przy zakupie licencji Enterprise udało się osiągnąć bardzo korzystny poziom zwrotu z inwestycji."

Bartłomiej Wnuk
Kierownik Projektu
PKP Energetyka S.A. Centrala

Incydenty

System posiada wbudowany moduł do zarządzania zgłoszeniami incydentów. Pozwala na ich tworzenie w procesie wykonywania testów lub, niezależnie, edycję, przypisywanie incydentów do poszczególnych członków zespołu, obsługę w pełni konfigurowalnym obiegu zadań, raportowanie w różnym wymiarze, śledzenie powiązań z innymi artefaktami systemu, aż do zamknięcia zgłoszenia. Incydenty to znacznie szersza kategoria niż defekty. Narzędzie umożliwia definiowanie różnych rodzajów zgłoszeń z oddzielnymi regułami tranzycji pomiędzy stanami.

Zarządzanie zmianą

W cyklu wytwórczym znacznym wyzwaniem jest poradzenie sobie z obsłużeniem dużej liczby zmian pojawiających się w różnych etapach cyklu życia oprogramowania. SpiraTeam posiada możliwość konfiguracji workflow dla artefaktów oraz możliwość śledzenia powiązań (traceability), co znacząco usprawnia poradzenie sobie z wyzwaniem dużej zmienności elementów procesu wytwarzania. Tworząc dodatkowy byt „Żądanie Zmiany" w module zarządzania incydentami, z odpowiednim workflow w postaci zamkniętej pętli, można sprawnie obsługiwać pojawiające się zmiany. Dla stworzonych w ten sposób „Żądań Zmian" możemy tworzyć powiązania z innymi artefaktami systemu. Zapewniamy w ten sposób pełną kontrolę nad implementacją. Dodatkowo – przy użyciu wbudowanych mechanizmów raportowych SpiraTeam ułatwia weryfikację zakresu prac do wykonania oraz ich postęp.

Wsparcie urządzeń mobilnych

Producent wraz z rozwiązaniem dostarcza dodatkowo interfejs dla urządzeń mobilnych umożliwiając tym samym korzystanie z narzędzia przy użyciu smartphone'ów i tablet'ów. Dzięki temu można na nich przeglądać dashboardy projektowe a nawet wykonywać testy. Przy zgłaszaniu problemu testowego/incydentu można dołączyć zdjęcie wykonane przy użyciu wbudowanej w urządzenie kamery.

Podsumowanie

Z naszego doświadczenia wynika, że SpiraTeam bardzo dobrze wspiera zarządzanie pracą zespołów zaangażowanych w procesy wytwórcze i wspierające rozwój oprogramowania. Narzędzie znacząco usprawniło komunikację i pracę nawet przy rozproszeniu geograficznym zespołu.

Usprawniło to komunikację i współpracę zespołu klienta oraz pozostałych dostawców. Producent umożliwił użytkownikom dostosowanie produktu na każdej płaszczyźnie – od ustawienia indywidulanie własnej strefy czasowej po integrację z innymi narzędziami.

Dzięki funkcjonalności konfigurowania ról, dostępów i widoków istnieje możliwość takiego nadania dostępów, aby dostawcy nie widzieli nawzajem swoich incydentów natomiast Project Manager miał możliwość podglądu wszystkich incydentów.

Dzięki temu Project Manager w jednym widoku ma dostęp do wszystkiego, a jednocześnie każdemu z dostawców udostępnia tylko niezbędny dla niego zakres informacji.

Wracając do Road Mapy wydań SpiraTeam warte podkreślenia jest, że w wersji 4.1, która ma ukazać się pod koniec września bieżącego roku, producent doda szereg intersujących usprawnień:

- Tworzenie przypadków użycia i zarządzanie nimi.
- Chat pomiędzy członkami projektu.
- Możliwość tworzenia podzadań (podobnie jak Jira).

Odpowiednim narzędziem ALM według oceny autora jest właśnie SpiraTeam, która została zweryfikowana w praktyce przez setki klientów na całym świecie. Interfejs narzędzia jest dostępny w wielu językach, tym również w języku polskim.

Przez półtora roku ustawicznego korzystania, utwierdziłem się w przekonaniu, że w pełni funkcjonalne i konfigurowalne narzędzie, posiadające ciekawe rozwiązania wcale nie musi kosztować wiele, a może pozytywnie wpłynąć na usprawnienie pracy zespołów IT. SpiraTeam z rozwiązania niszowego zaczyna wyrastać na czołowy produkt klasy ALM.

Maciej Zbrzeźniak
Test Manager
Soflab Technology Sp. z o.o.

MACIEJ ZBRZEŹNIAK
Test Manager w Departamencie Usług Wspierających Testy Soflab Technology
Posiada bogate doświadczenie we wdrażaniu narzędzi i rozwiązań usprawniających proces testowy. Prowadzi projekty automatyzacji testów i realizacji testów wydajnościowych. Inżynier i konsultant w jednej osobie. W wolnych chwilach motocyklista i snowbordzista.

SQAM™

Zaufało nam ponad 3 tys. osób z prawie 1 tys. firm i instytucji państwowych.

Sprawdź:

www.sqam.org

ISTQB® International Software Testing Qualifications Board

Requirements Engineering Qualifications Board

IBAQB INTERNATIONAL BUSINESS ANALYSIS QUALIFICATIONS BOARD

ITIL®

ACCREDITED EXAM CENTER gasq®

Zarządzanie Działem Testów

SQAM™

SQAM Sp. z o.o. Sp. komandytowa
ul. Bokserska 1
02-682 Warszawa
tel.: 22 427 36 83 | fax: 22 244 24 59
e-mail: sqam@sqam.org

Zespół SQAM
Joanna Konopka
joanna.konopka@sqam.org

Wdrożenie metodyki testów w oparciu o normę ISO/IEC 29119

Mając okazję współpracy z wieloma firmami IT obserwuję, że podejścia do organizacji obszaru testów i zapewnienia jakości są bardzo zróżnicowane.

Powinieneś wiedzieć:

- Czym charakteryzują się popularne metodyki testów.
- Jak zbudowane są procesy liniowe, kaskadowe.

Dowiesz się:

- Jak jest zbudowany model warstwowy w normie ISO/IEC 29119 Software and systems engineering — Software testing.
- Jaka jest specyfika użycia procesów wywoływanych zdarzeniami.
- Jak zmapować procesy z normy na kaskadowe modele wytwarzania oprogramowania.
- Na jakie problemy można się natknąć w czasie wdrożenia metodyki opartej o normę ISO/IEC 29119.

W pewnym stopniu wynika to z różnych profili działalności firm – inne są potrzeby firmy dostarczającej autorskie rozwiązania IT, inne integratora, a jeszcze inne firmy zamawiającej systemy u zewnętrznych dostawców.

Wśród pomysłów na funkcjonowanie działów jakości i testów mamy model zakładający kontrolę jakości w ramach komórek developerskich, przez rozproszenie odpowiedzialności w części IT organizacji zależne od rodzaju czy poziomu testów. Są też organizacje, w których za jakość odpowiadają komórki biznesowe. Oczywiście te przykłady nie wyczerpują złożoności modeli organizacji testów, a jedynie pokazują istniejące różnice w podejściach.

Generalnie można stwierdzić jednak, że w wielu firmach wykorzystujących w swojej działalności rozbudowane systemy informatyczne zauważalny jest problem, niezależny od ich specyfiki czy branży, braku odpowiednio zdefiniowanych ram organizacyjnych dla działań związanych z kontrolą i zapewnieniem jakości oprogramowania. Celem artykułu jest przybliżenie normy ISO/IEC 29119, jako wartościowego fundamentu, na którym można budować zasady testowania i kontroli jakości w organizacjach.

Dlaczego ISO/IEC 29119?

Wybór metodyk i norm mamy szeroki, co nie oznacza, że jest to wybór prosty. Jeżeli chcemy skorzystać z rozwiązania popartego uznanym autorytetem, to warto pomyśleć o międzynarodowych komitetach standaryzacyjnych, co kieruje nas bezpośrednio w obszar skodyfikowanych norm. Po krótkim przeglądzie norm okazuje się, że większość z nich nie obejmuje całościowo procesu testowego. Normy opisują wybrane aspekty wytwarzania oprogramowania, albo są archaiczne, kompletnie niedostosowane do nowoczesnych technik wytwarzania oprogramowania. Jednym słowem mamy problem. Sytuacja ta na szczęście została już dostrzeżona i od kilku lat trwa praca nad normą ISO/IEC 29119, która w założeniach ma objąć kompletny proces testowy i pozwalać na wykorzystanie nowoczesnych podejść do testów. Norma ta nie jest jeszcze oficjalnie opublikowana, ale weszła już w taką fazę dojrzałości, że warto z niej skorzystać, do czego zachęcają społeczność testerską jej twórcy.

W związku z tym, że rozpoczynaliśmy duży projekt wdrożenia metodyki u dużego klienta postanowiliśmy skorzystać z tego dobrodziejstwa.

Model warstwowy

Normy z rodziny ISO/IEC 29119 opisują podejście do poszczególnych aspektów testowania, a część 2 normy skupia się na opisie procesu testowego. Norma ta opisuje standardowy proces testowy, który należy dostosować do potrzeb organizacji.

Zgodnie z normą proces testowy jest opisany na trzech warstwach: organizacyjnej, zarządczej i operacyjnej. Na każdym z tych poziomów są zaproponowane odpowiednie zestawy procesów.

ORGANIZACYJNY PROCES TESTOWY – ma za

zadanie definiować proces budowania i utrzymania organizacyjnych specyfikacji testowych (Polityka czy Strategia), procesów, procedur i innych aktywów testowych.

PROCES ZARZĄDZANIA TESTAMI – definiuje procesy dotyczące zarządzania testami dla całych projektów testowych, poszczególnych faz czy typów testów (jak testy systemowe czy wydajnościowe)

OPERACYJNY PROCES TESTOWY – definiuje uniwersalne procesy i procedury prowadzenia testów, które mogą być stosowane zarówno do poszczególnych poziomów testów (np. sprawdzania wstępnego czy akceptacyjnych) lub rodzajów testów (np. ciągłości biznesowej czy migracji danych) w ramach projektu testowego.

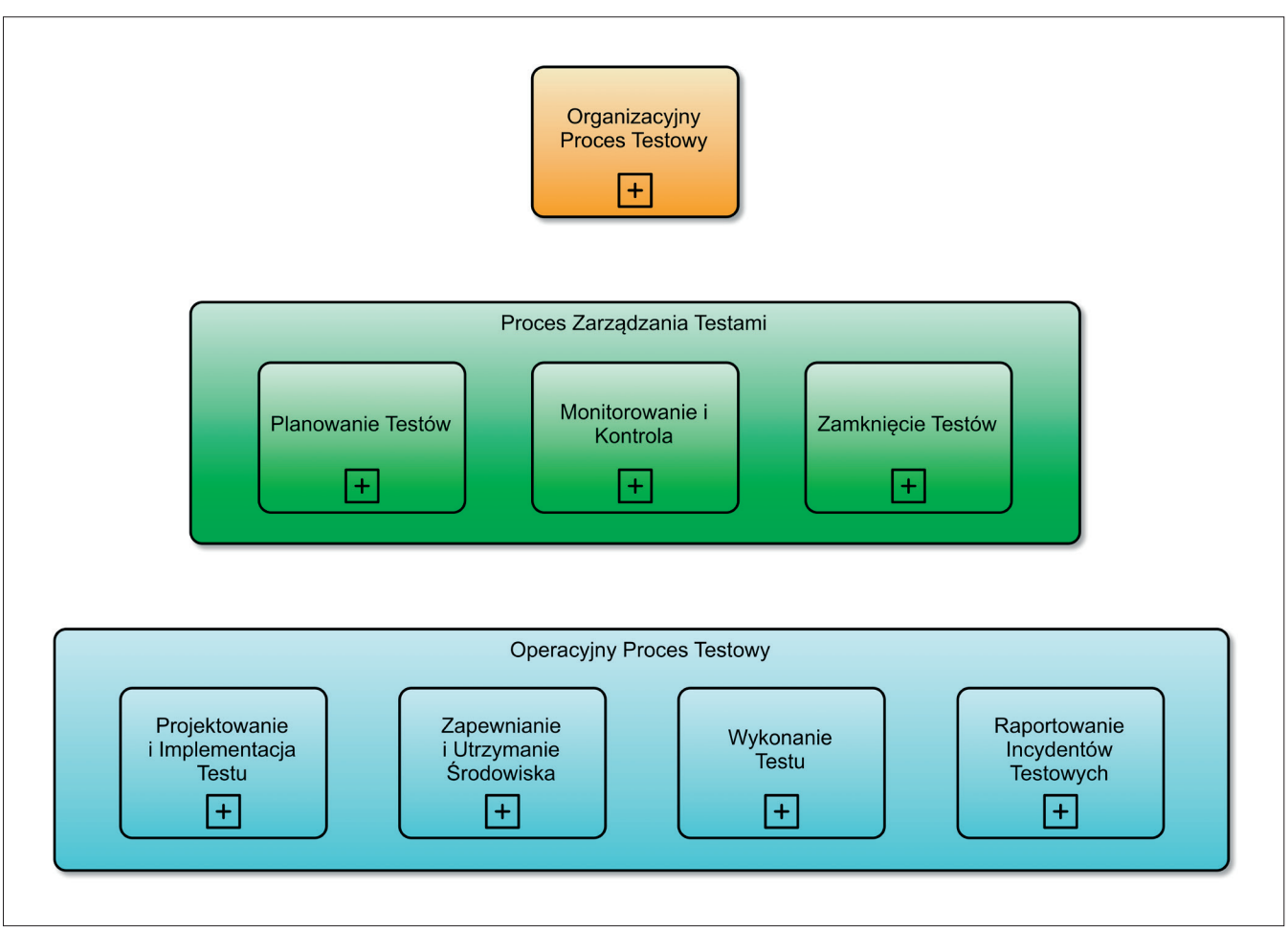
Procesy te można dostosować do dowolnego modelu wytwarzania oprogramowania. A co za tym idzie można je dostosować do już obowiązującego w organizacji modelu wytwórczego.

Sposób użycia procesów

Niezwykle istotne jest zrozumienie przedstawionych w normie zasad i założeń funkcjonowania procesu testowego. Największą zmianą w stosunku do standardowych metodyk jest to, że procesy opisane w normie nie są ułożone liniowo, kaskadowo. Jest to największy problem przed jakim stają zarówno osoby implementujące metodykę, jak też te, które metodykę tę mają stosować na co dzień.

Warto pamiętać o kilku zasadach:

- Procesy te nie są wywoływane liniowo, jeden po drugim, ale są wyzwalane zdarzeniami.
- Jednocześnie może być używana nadrzędna i podrzędna instancja procesu. Dobrym przykładem jest tu proces Monitorowania i Kontroli, który jednocześnie może działać dla całego Programu (zestawu Projektów), jak też w tym samym czasie jako instancja procesu dla aktualnie realizowanego Projektu. W pierwszym przypadku za proces odpowiedzialny będzie np. Kierownik Testów Programu, a w drugim Lider Testów Projektu.
- Kolejne wywołania procesu mogą tworzyć bardziej szczegółowe dokumenty lub je aktualizować. Przykładem jest proces Planowania Testów, którego zadaniem raz jest wytworzenie Głównego Planu Testów (dla całego Programu), a w ramach kolejnej iteracji – procesu Planów Testów Projektu/Rodzaju (np. Planu Testów Regresji). Procesy mogą być wywoływane rekurencyjnie.
- Mogą być wielokrotne wywołania tego samego procesu. Przykładem jest proces Wykonania Testów,



Rysunek 1. *Model warstwowy Procesów Testowych*

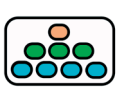
który jest wywoływany iteracyjne dla każdego Scenariusza i Przypadku Testowego.
- Przy wykorzystaniu czynności opisanych w procesach zakłada się możliwość nawrotów do wcześniejszych czynności, co nie jest dodatkowo oznaczane na diagramach.

Wdrażając metodykę testów opartą o normę trzeba dopasować się do istniejących w organizacji procesów, które często mają właśnie charakter kaskadowy, a norma w żaden sposób nie podpowiada jak to wykonać. Jako rozwiązanie dla tego problemu została przyjęta metoda „piramidek" (opisana dalej w artykule).

Odbiorca metodyki, dopóki nie zinternalizuje opisanych tu zasad czuje się nieco zagubiony pośród procesów, które nie następują kolejno po sobie. W pewnym momencie następuje „efekt Aha!" i wszystko staje się oczywiste i proste.

Metoda piramidek

Metoda piramidek polega na pokazaniu modelu warstwowego w postaci ikony „piramidki":



Na ikonie tej zaznacza się, które procesy mogą być wywołane na danym etapie kaskadowego modelu wytwarzania oprogramowania. Przykładowo ikona:

oznacza, że na danym etapie mogą być użyte procesy:

- Planowanie Testów;
- Monitorowanie i Kontrola Testów;
- Projektowanie i Implementacja Testów.

Na potrzeby tego artykułu został przyjęty uproszczony, przykładowy model wdrożenia oprogramowania:

- PLAN – planowanie projektu;
- ANALIZA – zbieranie i analiza wymagań;
- PROJEKT – projektowanie rozwiązania, architektura;
- BUDOWA – budowa rozwiązania w tym prototypowanie;
- TESTY – testy różnego rodzaju, w tym testy akceptacyjne;
- WDROŻENIE – w tym testy gotowości operacyjnej organizacji;
- PRZEGLĄD – kończące projekt podsumowania, w tym ocena jakości pracy testów.

Rysunek 2 prezentuje przykładowe mapowanie procesów z ISO/IEC 29119 dla powyższego modelu.

Opis procesów

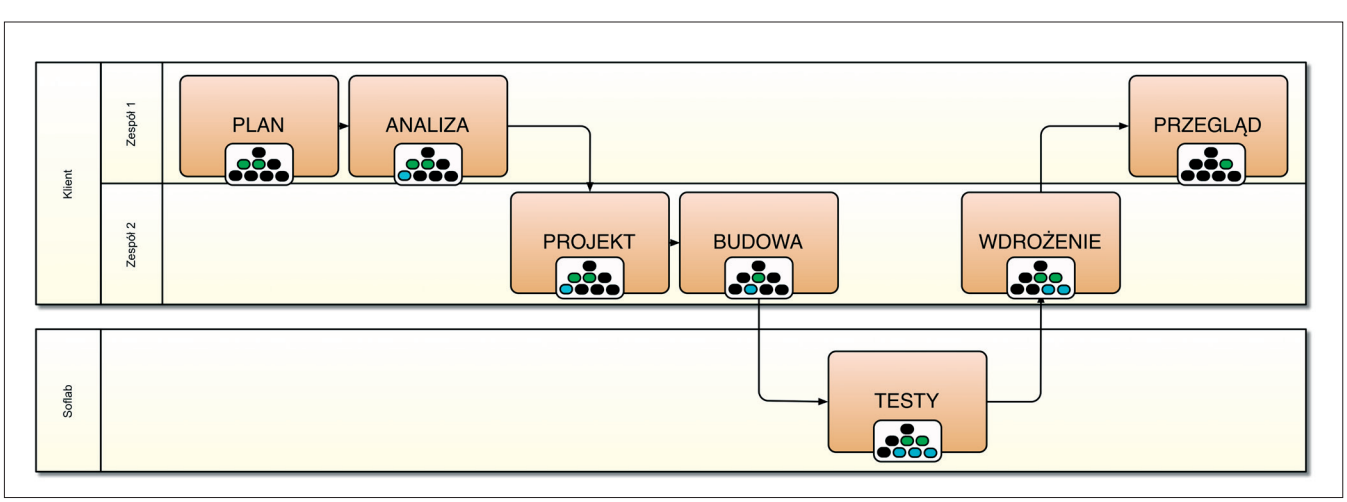
Do opisu każdego z procesów skorzystaliśmy ze standardowej notacji dla modeli procesów, czyli BPMN. Norma dostarcza uproszczone diagramy procesów, które należy dostosować do danej organizacji. Procesy również zostały zdefiniowane opisowo w dokumentach tekstowych.

Rysunek 3 przedstawia przykładowy diagram dla Procesu Planowania Testów w wersji z normy. Wersja uproszczona i zanonimizowana z implementacji metodyki u klienta, przedstawiona została na Rysunku 4.

Czego nam zabrakło w normie ISO/IEC 29119

Norma jest kompletna w zakresie procesu testowego, natomiast w każdej organizacji proces testowy jest zanurzony w innym otoczeniu. W związku z tym, żeby metodyka testów była kompletna może wymagać uzupełnienia o procesy wspierające, procesy około testowe i w każdej organizacji może to być inny zestaw.

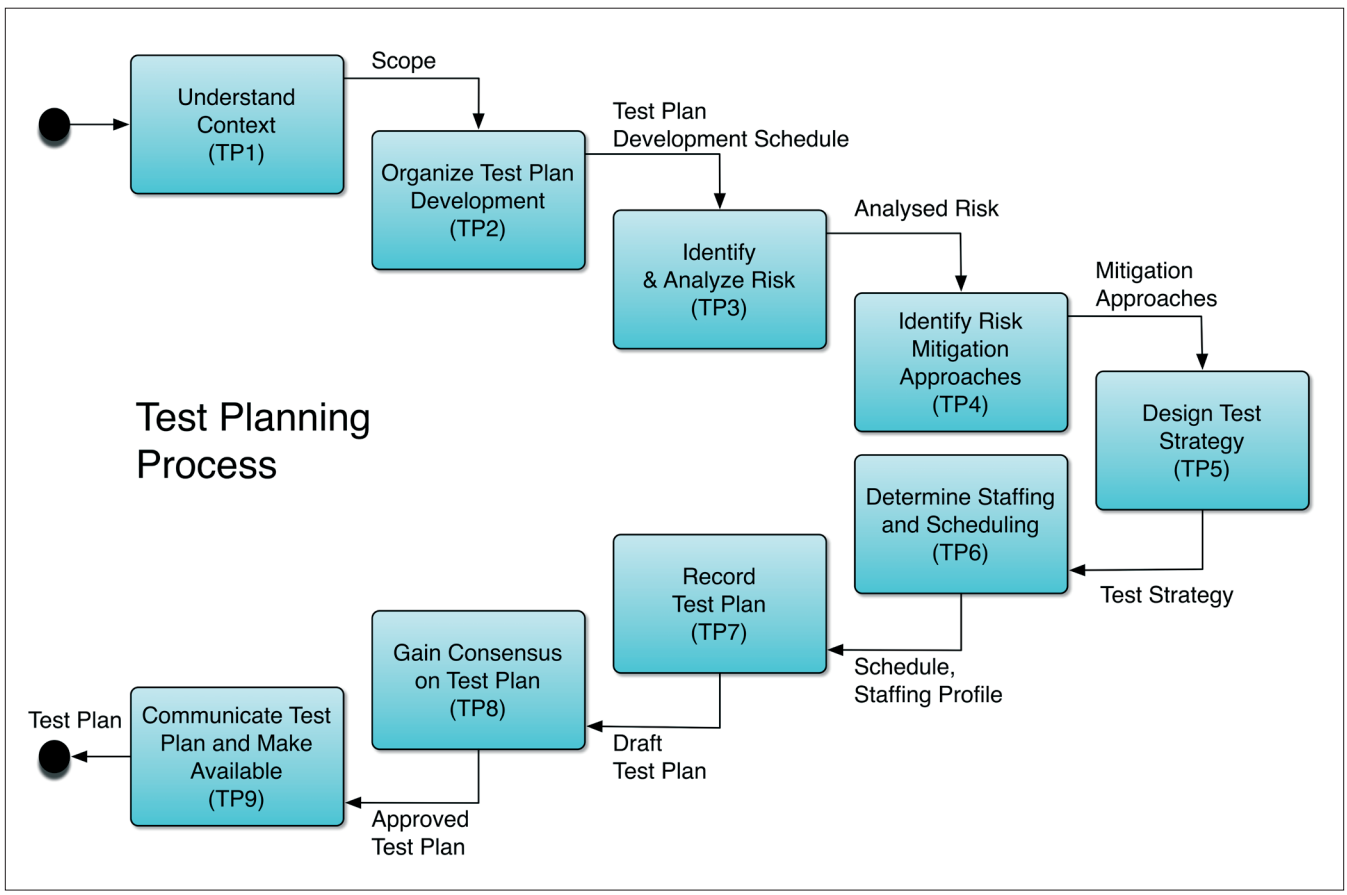
Do kompletnego procesu testowego, tak jak jest on rozumiany w naszej firmie, zabrakło jeszcze procesów:



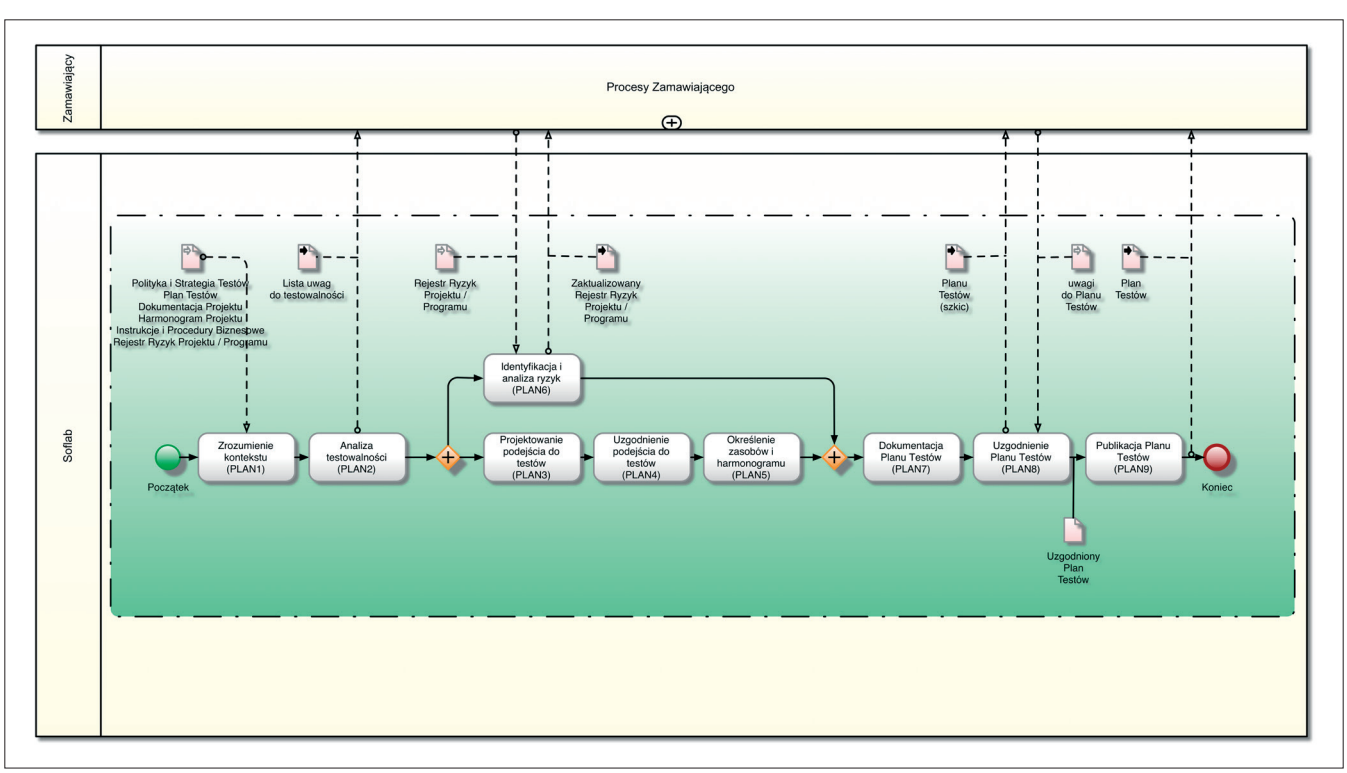
Rysunek 2. *Przykładowe mapowanie procesów z normy na model wdrażania oprogramowania*

- Zarządzania Jakością;
- Zarządzania Incydentami i Problemami Testowymi.

Zarządzanie Jakością jest szerszym pojęciem niż testowanie, w związku z tym dla kompletności metodyki



Rysunek 3. *Diagram Procesu Planowania Testów z normy*



Rysunek 4. *Diagram Procesu Planowania Testów z implementacji metodyki u klienta (wersja uproszczona)*

proces ten został dopisany. Stanowi on niejako otoczkę procesu testowego, dzięki temu dostarcza do procesu testowego metryki i wytyczne, a jednocześnie jest odbiorcą produktów Procesu Zamykania Testów takich jak uzupełnione wartości metryk i Wnioski na Przyszłość (Lessons Learned). Jego podstawowym celem jest ustanowienie i utrzymywanie strategicznej orientacji na zarządzenie jakością w procesach rozwojowych.

Zarządzanie Incydentami i Problemami Testowymi w zależności od organizacji może być umieszczane w różnych miejscach organizacji. Zarządzanie Incydentami może być umieszczone tak w projekcie jak i w centrach kompetencyjnych. Zarządzanie Problemami bywa czasem umieszczane w strukturach utrzymaniowych (wtedy kiedy w firmie obowiązuje jeden proces zarządzania problemami dla testów i dla produkcji). Jednakże życzeniem klienta w tym konkretnym wdrożeniu było, żeby Zarządzanie Incydentami i Problemami leżało w gestii zespołu testów. Z uwagi na fakt, że zespół Soflab ma kompetencje pozwalające na wykonanie takich zadań, również w tym wypadku został dopisany odpowiedni proces wspierający. W procesach testowych według ISO/IEC 29119 znajduje się jedynie początek tego procesu, czyli Raportowanie Incydentów Testowych.

Dodatkowo norma nie podaje jakie role powinny realizować poszczególne procesy, jest to dosyć obszerny temat do indywidualnych ustaleń w ramach organizacji. Celem jednoznacznego opisania odpowiedzialności za poszczególne aktywności w procesach testowych oraz na styku tych procesów z innymi procesami niezbędne jest wprowadzenie takiego zestawu ról do metodyki.

Podsumowanie

Metodyka testów w przedstawionym w artykule kształcie została przygotowana i wdrożona w organizacji klienta. Była poddawana ewaluacji w wielu działach mających styczność z testami, w tym w działach biznesowych. Spotkała się z dobrym przyjęciem. W trakcie przekazywania wiedzy o nowej metodyce testów otrzymywaliśmy wiele uwag, z których część wymagała jedynie wyklarowania opisów, a część wpłynęła na zmianę ustaleń. Jednakże szkielet metodyki oparty o ISO/IEC 29119 nie wymagał zmian. Zmiany dotyczyły jego uszczegółowienia i zmapowania na rzeczywistość projektową w organizacji. Aktualnie jesteśmy po pierwszych zastosowaniach metodyki w realnych projektach testowych. Kolejne iteracje wykorzystania nowego podejścia do testów wymagają coraz mniejszych poprawek. Biorąc pod uwagę założenie ciągłego usprawniania (Continuous Improvement) zmiany do metodyki są jej permanentną cechą.

Największym wyzwaniem dla osób niezaznajomionych z metodyką okazała się być konwencja opisu procesów (wielowątkowość), której zrozumienie wymaga „przestawienia się" i odmiennego (niż typowo liniowe) postrzegania procesów.

Wnioski jakie mamy po wdrożeniu:

- Nie zawsze jest sens i możliwość podążania za normą, ale na pewno jej wytyczne są dobrą bazą do budowania zasad organizacji testów.
- Wdrożenie metodyki powinno być realizowane bardziej zwinnym podejściem (uwzględniającym częste np. tygodniowe iteracje z przedstawicielami klienta) niż sekwencyjnym.

Metodyka oparta o ISO/IEC 29119 sprawdziła się w realnym zastosowaniu, w związku z tym chętnie będziemy korzystali z niej przy wdrożeniach u kolejnych klientów.

Spora część organizacji nie stosuje elementarnych zasad związanych z niezależnością testowania, nie wykorzystuje weryfikacji kryteriów przejścia poszczególnych etapów czy audytów jakościowych produktów procesu wytwórczego, stąd jest spore pole do usprawnień w tym obszarze, a norma ISO/IEC 29119 daje wytyczne jak to realizować.

ADAM SUSKIEWICZ
Ekspert testów i zapewnienia jakości
Soflab Technology
Posiada wieloletnie doświadczenie w zarządzaniu zespołami testowymi i prowadzeniu testów dużych wdrożeń w środowisku systemów zintegrowanych. Aktualnie pełni funkcję szefa Biura Zarządzania Testami w firmie Soflab Technology. Prywatnie szczęśliwy ojciec dwójki dzieci.

Jesień testowania

Powoli rozpoczyna się kalendarzowa jesień, ale w naszej branży od dłuższego czasu mamy do czynienia z jesienią testowania. Testowanie oprogramowania jakie wszyscy(?) doskonale znamy powoli odchodzi do lamusa wraz z calym podejściem kaskadowym (ang. Waterfall) do wytwarzania oprogramowania. Czy to koniec testowania w ogóle?

Dowiesz się

- Czym różni się zapewnianie jakości w podejściu Agile od Modelu Kaskadowego?
- Dlaczego tradycyjny Project Management nie zawsze działa w przypadku IT?
- Czego wymaga się od testerów we współczesnych procesach wytwarzania oprogramowania?

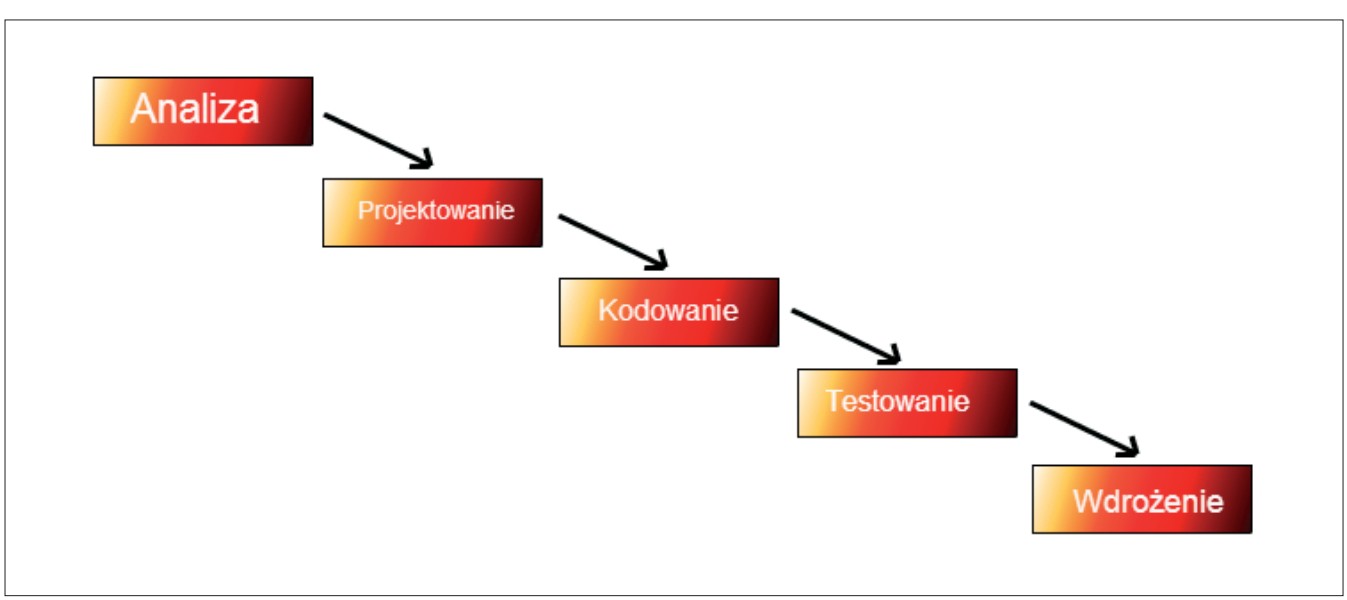
Powinieneś wiedzieć

- Czym jest Model Kaskadowy -ang. Waterfall?

Z pewnością koniec znanego nam podejścia do testowania oprogramowania, w którym testerzy stali zawsze w opozycji do programistów. Zanim zaczniemy rozważania na temat tego jak testowanie wygląda w nowoczesnych metodach wytwarzania oprgramowania takich jak Scrum czy ogolnie Agile spróbujmy przeanalizować jak wyglądało to dotychczas i co było z tym nie tak.

Przede wszystkim testowanie stanowiło dotychczas osobny etap w procesie wytwarzania oprogramowania. Całościowo proces ten wyglądał jak na Rys. 1.

To znaczy wszystko wyglądałoby jak na powyższym obrazku gdyby teoria była tożsama z praktyką. W praktyce nie istniało coś takiego jak płynne przejście pomiędzy poszczególnymi etapmi wytwarzania oprogramowania. Największe problemy wychodziły zazwyczaj na etapie testowania. Gdzie wykrywane są błędy, które powstały we wcześniejszych etapach. Dlatego też faza kodowania pomimo tego, iż oficjalnie została zakończona przeradzała się w fazę poprawiania błędów. Niestety tego na powyższym obrazku nie widać. Aby dobrze zobrazować rzeczywistość pozwoliłem



Rysunek 1. *Klasyczny model kaskadowy*

sobie wprowadzić pewne zmiany w obrazie modelu Waterfall - patrz Rys. 2.

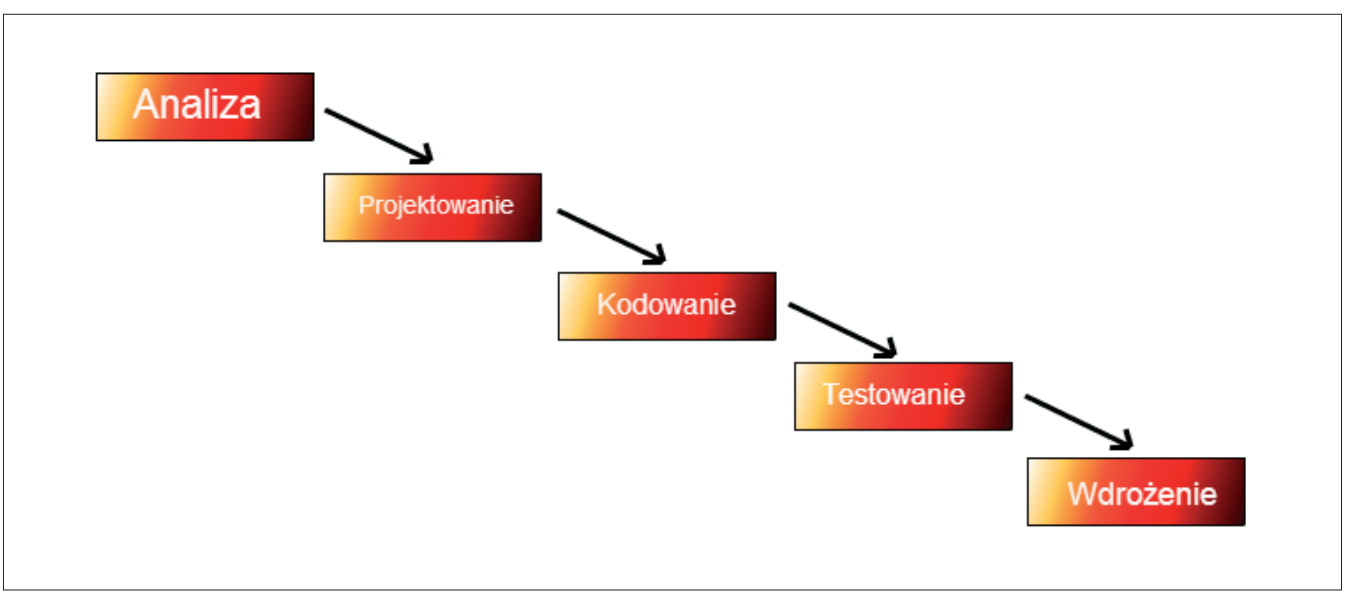
Pomiędzy testowaniem a programowaniem występowała pętla zwrotna. Znalezione przez testerów błędy wracały do programistów, którzy je następnie poprawiali, po czym oddawali do re-testów. Największym ryzykiem w takim podejściu były opóźnienia w dostarczaniu funkcjonalności do testów. Im później testerzy otrzymywali kolejną wersję oprogramowania do testów tym mniej czasu mieli na testy, a tymbardziej na re-testy po poprawkach. Takie podejście było nieefektywne.

To właśnie stąd wzięło się główne założenie Agile polegające na skracaniu pętli informacji zwrotnej. Tak, testowanie oprogramowania nie jest niczym innym jak dostarczaniem informacji zwrotnej na temat jakości wytwarzanego produktu.

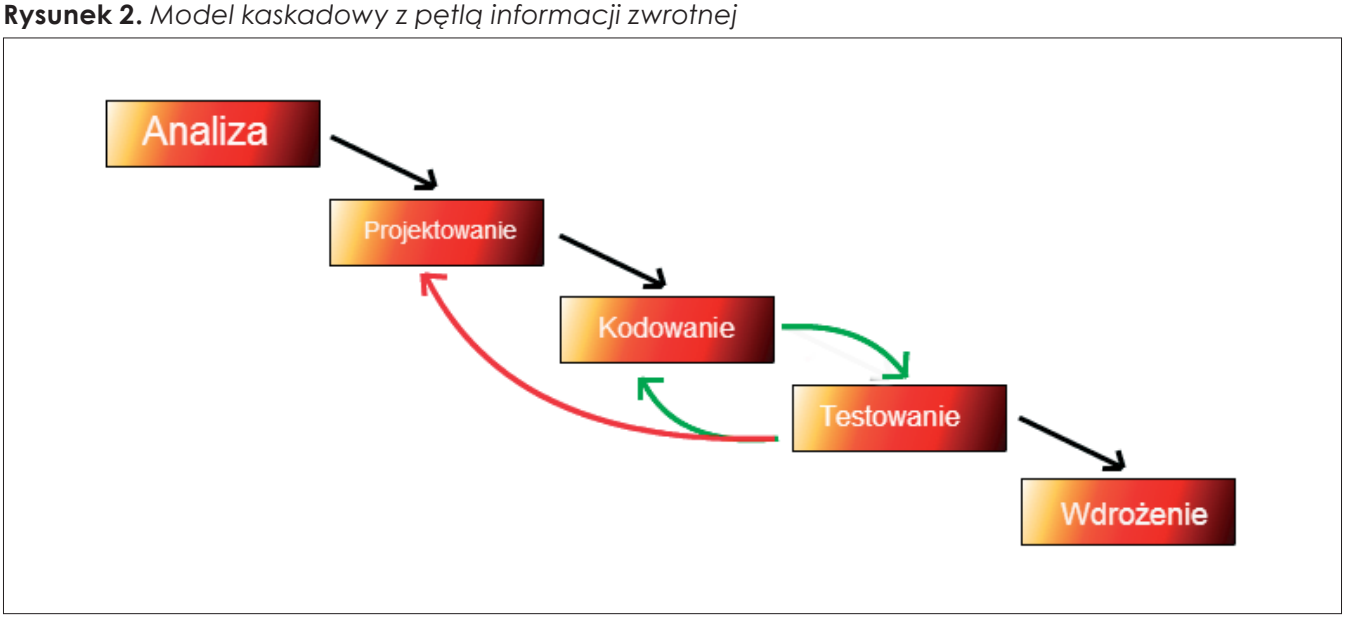
Powszechnie wiadomo (na podstawie badań przeprowadzonych już w latach 80-tych), że im później wykryje się błędy tym bardziej stają się one kosztowne. Czyli, gdy testerzy znajdą błąd a programiści stwierdzą, że błąd wynika z błędnego zaprojektowania architektury to nasz cały proces powinien wrócić do etapu projektowania. Wygląda to mniej więcej tak jak na Rys. 3.

Tą informację zwrotną też wypadało by skrócić. Stąd taki nacisk w "zwinych metodach wytwarzania oprogramowania" na wskroś-funkcjonalność zespołów. Jeśli ludzie pracują razem w zespole, najlepiej siedząc w jednym pomieszczeniu oraz mają wspólny cel jaki jest dostarczenie działającego oprogramowania, to komunikacja między nimi przebiega w sposób płynny, a informacje są dostarczane bardzo szybko.

Podstawowym problemem organizacji, które wcześniej wytwarzały oporgramowanie w sposób kaskadowy a obecnie chcą przejść na "model zwinny" jest silosowatość kompetencyjna. W takich organizacjach mamy do czynienia z osobnymi działami analizy, programowania i testów.



Rysunek 2. *Model kaskadowy z pętlą informacji zwrotnej*



Rysunek 3. *Rozszerzona pętla informacji zwrotnej*

O ile sama specjalizacja nie stanowi problemu to istnienie wyraźnych granic pomiędzy tymi działami prowadzi do poważnych komplikacji na poziomie komunikacji i zaangażowania.

Dosyć klasycznym przykładem obrazującym powyższe problemy jest powszechny brak zaangażowania poszczególnych obszarów w organizacji w wytwarzanie produktu jako czegoś kompletnego. W wielu organizacjach istnieje przekonanie, że wystarczy, iż ja wykonam swoje zadanie dobrze i w zasadzie nie muszę przejmować się całą resztą. Nie chodzi tutaj nawet o pracę zespołową i wzajemną pomoc ale raczej o wizję produktu, który tworzymy. Nawet jeśli wszyscy w organizacji bardzo się starają i wykonują swoje zadania najlepiej jak potrafią (w co nie wątpię) to w efekcie i tak nie zawsze powstają produkty wysokiej jakości - o ile w ogóle jakiekolwiek produkty powstają.

To właśnie dlatego jako firma doradczo-szkolniowa skupiamy się na dostarczaniu kompleksowych usług dla całej organizacji, tak by we wszystkich jej zakątkach wizja tego jak powinny wyglądać produkty oraz procesy ich wytwarzania była spójna. Tylko w ten sposób można zwiększyć efektywność oraz sprawić, by wytwarzane produkty były najwyższej jakości.

Wracając do naszego modelu wytwarzania oprogramowania, bardzo często okazuje się w fazie testowania, czy dopiero po wdrożeniu naszego produktu, że zostały popełnione zasadnicze błędy na etapie analizy wymagań. W takim przypadku nasz proces powoli zaczyna wyglądać mniej więcej tak jak na Rys. 4.

Jak widać na powyższym obrazku wytwarzanie oprogramowania wymaga współpracy i niezaburzonej niczym komunikacji pomiędzy osobami odpowiedzialnymi za każdy etap wytwarzania oprogramowania.

Strzałek we wszystkie strony można by narysować jeszcze więcej, niestety zaciemniło by to tylko obraz. Dlatego dobrze zobrazowany proces powinien wyglądać tak jak na Rys. 5.

To właśnie produkt - produkt jako całość jest w centrum naszego procesu. Dlatego wszyscy powinniśmy mieć wspólny cel jakim jest wytworzenie działającego, czyli jak by nie było przetestowanego produktu wysokiej jakośći, który będzie dokładnie tym czego potrzebuja nasi klienci.

Współczesne metody wytwarzania oprogramowania nawiącują również do ciągłej współpracy z biznesem oraz do testowania naszych założeń biznesowych bezpośrednio na rynku. Stąd właśnnie na powyższym obrazku znalazło się również wdrożenie.

Na przykład Scrum opiera się na założeniu, że wytwarzamy produkt w sposób inkrementalny. Czyli, że w każdej iteracji dostarczamy kompletny, przetestowany i potencjalnie gotowy do wydania inkrement produktu. Inkrementalnosc powinna występować także na poziomie kazdego wymagania z osobna. Aby móc testować nasze

Rysunek 4. *Krążące informacje*

Rysunek 5. *Produkt w centrum procesu*

założenia biznesowe powinniśmy każde zaimplementowane i przetestowane wymaganie wdrażać na produkcję i w ten sposób badać reakcje rynku na nasze pomysły.

W ten sposób kawałek po kawałku tworzymy kompletny produkt, który ma tą potężną przewagę nad innymi, że jest dokładnie tym czego potrzebują jego użytkownicy.

Jak widać we współczesnych metodach wytwarzania oprogramowania, powszechnie stosowanych na zachodzie i coraz częściej adaptowanych w przez nasze krajowe organizacje rola testera jest znacznie szersza niż w przypadku modelu kaskadowego. To właśnie dlatego od testerów wymaga się coraz więcej. Niemniej jednak w parze za rosnącymi wymaganiami rośnie również znaczenie testera w projektach, co oczywiście, jeśli tylko ktoś te wymagania spełnia ma swoje odzwierciedlenie w jego zarobkach.

Same umiejętności testerskie, techniki testowania, projektowania testów itp. to domena głównie testerów i obszary, których poprawę należy właśnie do nich adresować. Ale warto zaznaczyć, że już na przykład automatyzacja testów i testy automatyczne to zagadnienia dużo szersze niż samo testowanie oprogramowania i dlatego też powinny dotyczyć całej organizacji a nie tylko testerów.

Skupianie się na testowaniu jako odrębnym etapie procesu wytwarzania oprogramowania to popełnianie błędu już na etapie założenia, że testowanie jest odrębnym etapem wytwarzania oprogramowania.

Więcej na ten (i wiele innych ciekawych tematów) możecie przeczytać w mojej książce "*Mity i problemy w Agile*" (*https://leanpub.com/problemywagile*).

O AUTORZE

Wiktor Żołnowski - Trener i Agile Coach specjalizujący się w transformacjach Lean i Agile w organizacjach IT i nie tylko. Współwłaściciel firmy doradczoszkoleniowej Code SpriNters. Jak sam mówi obecnie jest bezrobotny bo praca, którą wykonuje jest jego hobby. Programista, QA i pragmatyk. Software Craftsmen. Autor książki "Mity i problemy w Agile" którą możecie znaleźć tutaj: https://leanpub.com/problemywagile. Bloger autor jednego z pierwszych w Polsce blogów o jakości oprogramowania blog.testowka.pl

Code Sprinters jest liderem na rynku polskim w doradztwie i szkoleniach w zakresie nowoczesnych metodyk tworzenia oprogramowania oraz zarządzania IT.

CodeSprinters
Experts in Agile

Naszą specjalnością są metody zwinne (ang. Agile), zarówno na poziomie organizacji procesu jak i praktyk technicznych. Posiadamy również doświadczenie w procesach zmiany organizacyjnej, a także tradycyjnych metodach zarządzania projektami.

Więcej informacji o naszych usługach można znaleźć na stronie
http://www.agileszkolenia.pl

Koncepcja tworzenia logów diagnostycznych

z punktu widzenia testera oraz działu wsparcia technicznego

Temat tworzenia logów traktowany jest przeważnie w sposób dość pobieżny. Najczęściej logi służą głównie deweloperom tworzącym kod, zwykle programiści mają również możliwość reprodukcji znalezionego błędu we własnym środowisku. Zdarzają się jednak projekty, w których testowanie wykonywane jest przez zupełnie oddzielny dział przy użyciu wyspecjalizowanego sprzętu i nie jest możliwe wielokrotne powtarzanie niezaliczonego testu z użyciem debuggera. Poszukując błędu deweloperzy mogą bazować jedynie na logach zebranych przez testera. W artykule omówione zostaną najczęściej stosowane rozwiązania problemu logowania w złożonych systemach. Oceniona zostanie także przydatność każdego z tych rozwiązań na etapie testowania.

Dowiesz się:

- jaki system logowania wybrać,
- o czym pamiętać aby tworzone logi były przydatne nie tylko dla deweloperów,
- jak ułatwić testerom pracę przy testach typu „black-box".

Powinieneś wiedzieć:

- czym się charakteryzują podstawowe poziomy testowania,
- co to są testy modułowe, komponentowe, integracyjne, akceptacyjne.

W projektach programistycznych często mamy do czynienia z sytuacją, gdy powstające oprogramowanie jest docelowo przeznaczone do rozbudowanych urządzeń bądź systemów sterowania czy kontroli. O ile testowanie poszczególnych modułów czy też całych komponentów w takim projekcie można przeprowadzać zwykle w klasyczny sposób przy użyciu symulatorów bądź odpowiednio przygotowanych środowisk testowych, o tyle integracja i testowanie akceptacyjne wymagać mogą specjalistycznych laboratoriów bądź są możliwe jedynie w systemie docelowym znajdującym się już u klienta.

Sytuacja taka może mieć miejsce na przykład w złożonych systemach telekomunikacyjnych, gdzie z oczywistych względów wytwórca oprogramowania nie jest w stanie zbudować środowiska testowego o tak dużej złożoności jak sieć kliencka. Tworzy się co prawda odpowiednio wyspecjalizowane sieci testowe będące pewnym modelem tego, z czym będzie musiał sobie poradzić docelowo testowany system, jednak zwykle dostęp do takich środowisk jest ograniczony czasowo. Nie dość, że ich obsługa wymaga wykwalifikowanego personelu to koszta budowy są dla firmy tak duże, że jedno środowisko w miarę możliwości wykorzystywane jest w wielu różnych projektach. Prowadzi to do mocno niekorzystnej z punktu widzenia programistów sytuacji, gdy na pewnym poziomie zaawansowania testy wykonywane są przez osobne działy w firmie, często położone w odległej geograficznie lokalizacji. Zdarza się też, iż pomimo posiadania przez firmę sieci testowej nie ma możliwości symulowania odpowiednio dobrze nieprzewidywalnych zależności czasowych, obciążenia, dużych wahań liczby użytkowników, zmienności warunków radiowych bądź awarii sprzętu, na którą napisane przez nas oprogramowanie powinno zareagować w określony sposób. Skutkiem tego jesteśmy zmuszeni część testów wykonać dopiero w fazie wdrożenia u klienta końcowego. W efekcie w przypadku znalezienia błędu programiści nie mają możliwości jego reprodukcji w dostępnym dla nich środowisku i muszą w pełni polegać na raporcie dostarczonym przez testerów bądź wdrażających system inżynierów wsparcia technicznego.

Należy dodać, że w dużych projektach z logów korzystają nie tylko deweloperzy. Gdy pracujemy nad systemem złożonym z wielu komponentów pisanych przez

różne grupy programistów do obowiązków testera należy zwykle przynamniej wstępne wskazanie, która z grup powinna zająć się znalezionym problemem. Wymaga to oczywiście znajomości architektury systemu oraz zrozumienia zależności pomiędzy jego elementami. Niemniej jednak wiedza ta może okazać się zupełnie nieprzydatna, jeśli tester nie jest w stanie prześledzić tego, co wydarzyło się w systemie przed wystąpieniem niepożądanej reakcji. Zwykle jedyną możliwością zajrzenia choć trochę do wnętrza testowanego „black-boxa" są właśnie logi.

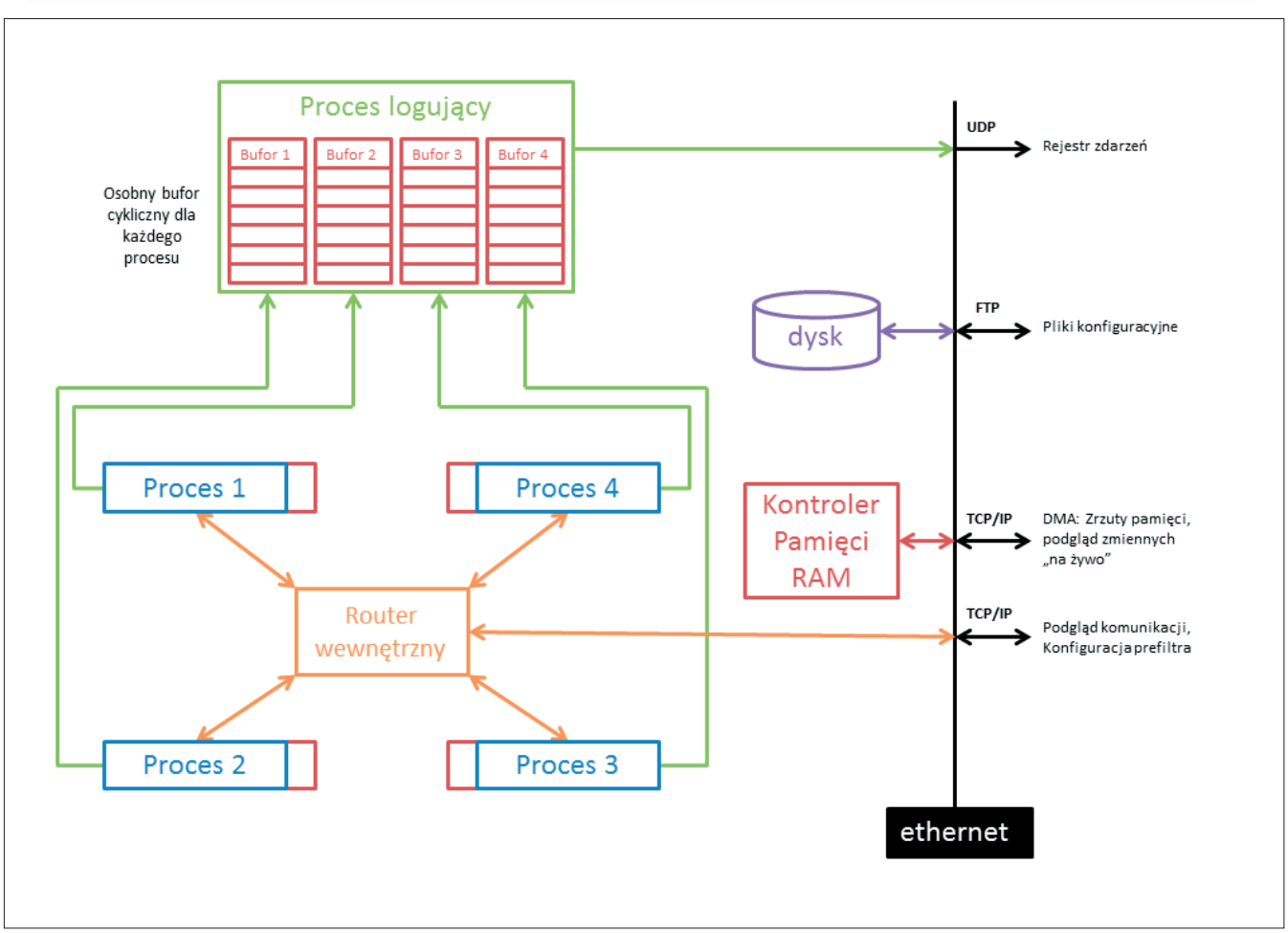
Zagadnieniem powiązanym z tematem artykułu jest też diagnostyka oprogramowania, które już zostało wypuszczone na rynek i okazało się, że zawiera błędy. Zwykle raport dostarczony przez użytkownika końcowego jest ubogi, nie zawiera opisu procedury, która doprowadziła do ujawnienia się błędu bądź opis ten jest bardzo lakoniczny. W bardzo złożonych systemach rozproszonych często jest też tak, że defekt pojawia się okazjonalnie, nie da się określić, w jakich warunkach występuje i nie da się go zreprodukować nawet u klienta. W takiej sytuacji jedyne dane, jakimi dysponuje programista, to dostarczone mu logi.

Architektura systemu logowania

W dalszej części artykułu omówione zostaną najczęściej stosowane rozwiązania problemu logowania pracy systemu i przy okazji oceniona zostanie przydatność każdego z tych rozwiązań na etapie testowania. W pierwszej części skupmy się jednak na ogólnym założeniu, w jaki sposób logi mogą być zbierane. Zwykle logowanie ograniczone jest do stworzenia pliku, do którego w określonych sytuacjach dokonuje się wpisu. W systemach bardziej rozbudowanych każdy z komponentów

Po co testerom logi?

- pozwalają na identyfikację komponentu, w którym wystąpił błąd,
- ułatwiają zrozumienie jak działa testowany system i jakie występują w nim zależności czasowe,
- ułatwiają napisanie skryptów automatycznych do wykrywania zdarzeń w systemie,
- są świetnym narzędziem do wskazania programistom, że zaistniała sytuacja jest rzeczywistym błędem a nie problemem ze środowiskiem testowym.



Rysunek 1. *Przykład kompleksowego systemu logowania*

oprogramowania posiada własny log, co wymusza na osobie analizującej takie logi konieczność ich synchronizacji. Stąd ważne, aby każdy z wpisów posiadał precyzyjnie określony czas utworzenia. W zależności od rodzaju systemu wymagania dotyczące dokładności to mili- czy mikrosekundy, albo wręcz pojedyncze takty procesora.

Ciekawym i często stosowanym rozwiązaniem może być dodatkowe stworzenie dla każdego z komponentów bufora kołowego, w którym zapisywane będą logi. Rozmiar bufora powinien być ściśle określony tak, aby mógł zawierać na przykład kilkaset ostatnich wpisów. W sytuacji wystąpienia błędu lub na specjalne żądanie użytkownika możemy zawartość wszystkich buforów zapisać do plików. Ułatwi to znacznie poszukiwanie źródła błędu w sytuacji, gdy jeden z komponentów loguje znacznie częściej niż pozostałe i zalewa logi setkami wpisów w każdej sekundzie.

Gdy systemem docelowym dla naszego oprogramowania jest samodzielne urządzenie, warto zastanowić się nad implementacją interfejsu zewnętrznego do zbierania logów. Świetnie w tej roli sprawdzi się magistrala USB bądź sieć Ethernet (Rysunek 1). Zyskujemy w ten sposób wydajne narzędzie diagnostyczne dla naszego systemu.

Operator może podłączyć się do urządzenia i przy pomocy dodatkowego oprogramowania na żywo monitorować stan poszczególnych komponentów uzyskując na przykład podgląd niektórych zmiennych.

Z możliwości takiej bardzo chętnie skorzystają działy testujące oprogramowanie, które zyskają spore ułatwienie w automatyzowaniu swoich testów. Pojawi się także możliwość stworzenia aplikacji zewnętrznych automatycznie wykrywających niepożądane sytuacje. Należy zauważyć, że stworzenie testerom sytuacji, gdy testy typu „black-box" staną się czymś, co można nazwać „gray-box" na pewno bardzo pozytywnie wpłynie na współpracę działu testowania z programistami. Po pierwsze unikniemy zapewne wielu niesłusznie zaraportowanych błędów, a po drugie w wielu przypadkach wprost z logów będzie można wskazać gdzie jest błąd.

Klasyczny log typu "rejestr zdarzeń"

Rejestr zdarzeń jest najczęściej stosowanym a w większości rozwiązań także jedynym zaimplementowanym sposobem logowania. Zwykle jest to chronologiczny zapis charakterystycznych momentów w systemie, takich jak podłączenie się nowego użytkownika do serwisu, uruchomienie jakiejś procedury, zmiana parametrów konfiguracyjnych, przeprowadzenie testu wewnętrznego czy uruchomienie procedury obsługi błędu. Logi tego typu zbierane

Listing 1. Zawartość rejestru zdarzeń dla różnych poziomów logowania

```
RejestrZdarzen.txt, poziom logowania u klienta końcowego (WRN,ERR)
2013.09.12 12:54:08:254980 WRN KOMPONENT2/MODUŁ_A userId: 043 Unsupported channel format.
2013.09.12 12:54:08:255980 ERR KOMPONENT1/MODUŁ_C No transmission for userId:043.

RejestrZdarzen.txt, poziom logowania podczas testów (INF, WRN, ERR)
2013.09.12 12:54:08:254315 INF KOMPONENT1/MODUŁ_C New user attached the network (userId:043).
2013.09.12 12:54:08:254317 INF KOMPONENT2/MODUŁ_A New resources added for userId: 043.
2013.09.12 12:54:08:254980 WRN KOMPONENT2/MODUŁ_A userId: 043 Unsupported channel format.
2013.09.12 12:54:08:254981 INF KOMPONENT2/MODUŁ_A Resources for userId:043 are unused.
2013.09.12 12:54:08:255980 ERR KOMPONENT1/MODUŁ_C No transmission for userId:043.
2013.09.12 12:54:10:128341 INF KOMPONENT2/MODUŁ_A Resources removed for userId:043.
2013.09.12 12:54:10:128341 INF KOMPONENT1/MODUŁ_C userId: 043 deleted (cause: error).

RejestrZdarzen.txt, pełny poziom logowania (DBG, INF, WRN, ERR)
2013.09.12 12:54:08:254315 INF KOMPONENT1/MODUŁ_C New user attached the network (userId:043).
2013.09.12 12:54:08:254317 INF KOMPONENT2/MODUŁ_A New resources added for userId: 043.
2013.09.12 12:54:08:254319 DBG KOMPONENT2/MODUŁ_A userId:043 located at mem address: 0x12121
2013.09.12 12:54:08:254320 DBG KOMPONENT2/MODUŁ_A userId:043, channel format: 00 53 12 D1
2013.09.12 12:54:08:254980 WRN KOMPONENT2/MODUŁ_A userId: 043 Unsupported channel format.
2013.09.12 12:54:08:254981 INF KOMPONENT2/MODUŁ_A Resources for userId:043 are unused.
2013.09.12 12:54:08:255980 ERR KOMPONENT1/MODUŁ_C No transmission for userId:043.
2013.09.12 12:54:10:128341 INF KOMPONENT2/MODUŁ_A Resources removed for userId:043.
2013.09.12 12:54:10:128341 INF KOMPONENT1/MODUŁ_C userId: 043 deleted (cause: error).
2013.09.12 12:54:08:254315 INF KOMPONENT2/MODUŁ_A Resources removed for userId:043.
2013.09.12 12:54:08:254315 INF KOMPONENT1/MODUŁ_C userId: 043 deleted (cause: error).
```

Dobre praktyki dotyczące systemu logowania:

- każdy wpis powinien zawierać datę i dokładną godzinę wygenerowania oraz nazwę komponentu źródłowego,
- warto każdemu wpisowi nadać poziom istotności (czy jest to wpis diagnostyczny, informacyjny czy też zgłoszenie błędu),
- nie należy zmieniać raz ustalonego formatu wpisu w logach. Może to spowodować błędne werdykty w testach, które bazują na tych wpisach,
- należy uważać, aby nie ujawniać w logach zbyt wielu informacji na temat architektury systemu bądź sposobu działania algorytmów. Konkurencja nie śpi!

są najczęściej w postaci plików tekstowych. Zwykle praktykuje się rozwiązanie, gdzie każdy wpis oprócz dokładnego czasu wygenerowania i nazwy komponentu raportującego zawiera także przypisany odpowiedni poziom istotności (przykładowo: „debug", „info", „warning" lub „error"). Pozwala to testerowi bądź programiście na łatwiejsze wychwycenie najważniejszych wpisów.

Do działań rutynowych należy zwykle sprawdzenie po każdym teście (nawet zaliczonym), czy w rejestrze zdarzeń nie został zaraportowany błąd. Często przypadkowo wychwytywane są w ten sposób błędy nie wpływające na testowaną funkcjonalność, ale za to występujące w dość specyficznych warunkach, które akurat niezamierzenie udało się zapewnić.

Warto także rozważyć możliwość wprowadzenia konfigurowalnej prefiltracji logów. Na czas testowania bądź diagnostyki rejestrujemy logi wszystkich poziomów a w produkcie końcowym zmniejszamy ilość logów poprzez zapisywanie tylko tych o poziomie „warning" lub „error".

Oprócz zaoszczędzenia miejsca na dysku możemy także odciążyć w ten sposób system. Dla uzmysłowienia czytelnikowi wagi problemu podać można przykład stacji bazowej telefonii komórkowej, która w godzinach szczytu generować może w każdej minucie nawet 50 megabajtów logów typu „debug" i „info". Listing I zawiera przykład fikcyjnego logu ze stacji bazowej, który byłby wytworzony dla różnych poziomów logowania.

Do dobrych praktyk należy przyjęcie ustalonej formy wpisu i późniejsze trzymanie się tych zasad. Działy zajmujące się testowaniem oprogramowania chętnie bazują na rejestrze zdarzeń podczas sprawdzania czy zostały uruchomione odpowiednie procedury i czy np. kolejność ich uruchomienia była właściwa. Bardzo często sprawdzenia takie wykonywane są przez skrypty parsujące, więc może się okazać, że nawet poprawienie literówki we wpisach może wprowadzić sporo zamieszania w testach automatycznych. Zwykle osoby analizujące w swojej pracy logi definiują także reguły kolorowania poszczególnych wpisów po słowach kluczowych. Znacznie przyspiesza to orientację w zawartości logów.

Zrzuty pamięci

Podczas analizy zgłoszonego błędu okazuje się czasami, że zakres informacji zawarty w logach generowanych przez system nie jest wystarczający i jedyny sposób na dotarcie do przyczyny niepoprawnego zachowania programu to poproszenie testera bądź inżyniera wsparcia technicznego o dostarczenie zrzutów pamięci tuż po wystąpieniu defektu.

Dość dobrą i często stosowaną praktyką jest automatyczne generowanie zrzutu pamięci do pliku w pamięci flash na przykład w momencie uruchomienia procedury obsługi krytycznego wyjątku. Zwykle uzyskane w ten sposób dane nie są zrozumiałe ani dla testera ani dla inżyniera wsparcia technicznego gdyż wymagają znajomości kodu programu, znaczenia wszystkich zmiennych oraz przede wszystkim ich odczyt wymaga posiadania mapy pamięci, która przeważnie zmienia się wraz z każdą wersją kodu.Niemniej jednak programista znający dobrze kod programu i posiadający wszystkie niezbędne informacje jest w stanie wiele sie dowiedzieć o stanie w jakim znajduje się program ze zrzutu pamięci.

Zrzut diagnostyczny całego systemu

Bardzo wygodnym i efektywnym rozwiązaniem jest umożliwienie użytkownikowi końcowemu (a więc klientowi lub testerowi) samodzielnego wygenerowania raportu do celów diagnostycznych. Raport taki (na przykład w postaci automatycznie tworzonemu archiwum .zip) zawierać może chociażby wszystkie pliki konfiguracyjne, zrzuty pamięci kluczowych komponentów czy też ostatnie tysiąc wpisów dziennika zdarzeń. Dobrą praktyka jest ustalenie, że każdy z komponentów tworzonego systemu powinien posiadać bufor cykliczny o określonej wielkości przeznaczony na logi, zapis ostatnich wartości zmiennych itp. W momencie generowania raportu diagnostycznego zawartość tych buforów jest „zamrażana" i dołączana do raportu. Spotyka się także rozwiązania, w których raport zawiera zrzuty stanu wszystkich obiektów, co pozwala na późniejsze „załadowanie" go do systemu w taki sposób, aby programista szukający błędu mógł otrzymać w środowisku symulacyjnym dokładne odwzorowanie uzyskanej przez użytkownika sytuacji.

Kilka rad dla programistów

- staraj się skonstruować system logowania w taki sposób, abyś nie musiał go rozbudowywać na potrzeby znalezienia konkretnego błędu. Nie ma nic bardziej frustrującego dla testera niż niekończące się prośby o reprodukcję błędu i zebranie nowych logów,
- określ jasno, jakie logi są potrzebne by prześledzić działanie komponentu, który tworzysz. W razie potrzeby napisz instrukcję zbierania nietypowych logów,
- staraj się informować testera o wynikach analiz logów. Im lepiej pozna architekturę systemu i metody analizy jego działania, tym bardziej efektywnie w przyszłości będzie mógł identyfikować komponent odpowiedzialny za błąd.

Kilka rad dla testerów

- do zgłoszenia błędu dołączaj wszystkie logi, jakie tylko jesteś w stanie zebrać. Ułatwia to poszukiwania pomyłki w kodzie i często pozwala uniknąć niepotrzebnych retestów,
- przed zaraportowaniem błędu warto zapytać programistów, jakich dokładnie logów będą potrzebować do poszukiwań błędu w danym komponencie,
- jeśli programiści analizując logi komunikują się w formie pisemnej (e-mail, forum), śledź te dyskusje i proś o wyjaśnienia. Ułatwi to Twoją pracę, gdy napotkasz podobny błąd w przyszłości,
- dokonując wstępnej analizy logów staraj się dojść do przyczyny wystąpienia tego błędu. Przejrzyj wpisy poprzedzające usterkę i zweryfikuj widoczne parametry systemu. To, iż jakiś komponent raportuje błąd nie jest jednoznaczne z tym, że zawiera usterkę w swoim kodzie.

Zapis komunikacji pomiędzy komponentami oprogramowania

W przypadku oprogramowania złożonego z kilku wyraźnie wydzielonych komponentów bardzo przydatnym narzędziem diagnostycznym jest możliwość podglądu komunikacji wewnętrznej pomiędzy nimi (Rysunek 1, kolor pomarańczowy). Stosuje się tu najczęściej dwa konkurencyjne rozwiązania: skupione bądź rozproszone. Oba wymagają ustalenia, iż wszelka komunikacja odbywać się będzie poprzez wiadomości o jasno zdefiniowanym nagłówku i formacie. Nagłówek wiadomości zawierać powinien co najmniej nadawcę, odbiorcę i znacznik czasu. Rozwiązanie rozproszone polega na tym, iż każdy komponent automatycznie bądź na żądanie może przekazywać na zewnątrz kopię wszystkich wiadomości przychodzących i wychodzących. Rozwiązanie skupione z kolei zakłada, iż każda wiadomość pomiędzy komponentami przesyłana jest za pośrednictwem programowego przekaźnika (routera), który to może posiadać możliwość zapisywania kopii całej komunikacji. Kopia przechowywana może być w pliku lokalnym bądź też na przykład udostępniania poprzez interfejs zewnętrzny. W obu rozwiązaniach należy rozważyć możliwość filtracji takich logów już na poziomie ich tworzenia. Najczęściej nie potrzebujemy całej komunikacji pomiędzy komponentami a tylko kilka/kilkanaście określonych typów wiadomości.

Logi tego typu wymagają też zwykle specjalnego oprogramowania do dekodowania ich treści, a także posiadania klucza zawierającego szczegóły struktur wiadomości. Struktura ta może się przecież zmieniać wraz z rozwojem funkcjonalności systemu. Możliwość podglądu sygnalizacji pomiędzy komponentami oraz wartości przekazywanych parametrów pozwala często w jednoznaczny sposób zidentyfikować, który z komponentów zawinił, a co za tym idzie ułatwia rozstrzygnięcie sporu, który zespół programistów powinien zająć się naprawieniem błędu.

Podgląd wartości zmiennych

Często weryfikacja poprawności działania danego komponentu czy systemu wymaga znajomości wielu parametrów wejściowych, pośrednich i wyjściowych, które to z kolei mogą zmieniać się w sposób ciągły. Przykładem może być tu podsystem kontroli mocy w systemie radiowym. W zależności od zastosowanej technologii kilkadziesiąt bądź kilkaset razy na sekundę wyliczane są odpowiednie wskaźniki na podstawie zmierzonej mocy, dostępnych zasobów, poziomu zakłóceń, zajętości sąsiednich kanałów itd. Weryfikacja działania testowanego komponentu często nie jest możliwa bez posiadania zapisu przebiegu w czasie wszystkich wspomnianych parametrów oraz wyliczonych wskaźników.

Różnica pomiędzy zrzutami pamięci a rejestracją przebiegu zmiennych polega na tym, iż ten drugi sposób odwzorowuje jedynie wybrane wartości, ale za to znany jest ich przebieg w dziedzinie czasu. Analiza tego typu logów wymaga niestety bardzo specjalistycznej wiedzy na temat zaimplementowanego algorytmu. Niemniej jednak warto testerom udostępnić odpowiednie narzędzia i niezbędne informacje, gdyż często pozwala to zapobiec nieuzasadnionemu zgłoszeniu błędu. Z zapisu parametrów może być widoczne, iż komponent nie mógł zareagować w inny sposób, gdyż na przykład wyczerpały się kontrolowane zasoby

lub system działa w warunkach, dla których nie jest przeznaczony. Może też wystąpić sytuacja przeciwna, gdy tester widzi, iż parametry wejściowe są poprawne a komponent zachowuje się w sposób niezgodny ze specyfikacją. Nie ma wtedy wątpliwości co do istnienia pomyłki w kodzie.

Bezpieczeństwo systemu (inżynieria wsteczna na podstawie logów)

Na zakończenie warto zauważyć, że projektując system logowania oraz podejmując decyzję co do zawartości logów należy koniecznie przeanalizować kwestię bezpieczeństwa. Nie można dopuścić do sytuacji, gdy użytkownik końcowy z logów uzyska informacje, których z różnych powodów nie chcemy mu ujawniać.

Często przyjęty model biznesowy zakłada, że oprogramowanie dedykowane dla konkretnego klienta posiada zablokowane pewne funkcje bądź zgłaszane błędy nie są istotne dla działania systemu. Na pewno nie wzbudzimy zaufania klienta, który zobaczy rejestr zdarzeń wypełniony wpisami o wystąpieniu błędu.

W przypadku oprogramowania komercyjnego należy również zwrócić uwagę na to, żeby nie ujawnić w logach zbyt wielu informacji o architekturze systemu bądź zaimplementowanych algorytmach. Konkurencja chętnie skorzysta z każdej okazji, żeby metodami inżynierii wstecznej (odwrotnej) poznać jak zbudowany jest nasz produkt.

O AUTORZE

Marcin Dudek - Tester pracujący w dziale integracji i weryfikacji oprogramowania stacji bazowych systemu WCDMA. Posiada doświadczenie w analizie logów zarówno z sieci testowych (Field Verification) jak i pochodzących od klienta (Technical Support). W wolnych chwilach zajmuje się tworzeniem aplikacji do parsowania logów oraz do ich automatycznej analizy.

Testy integracyjne aplikacji webowych z wykorzystaniem języka JavaScript

Artykuł prezentuje wybrane narzędzia wspomagające tworzenie automatycznych testów integracyjnych oraz testów akceptacyjnych aplikacji webowych w środowisku JavaScript. Jego celem jest wprowadzenie programistów, testerów i kierowników testów w różnorodny, szybko rozwijający się świat rozwiązań dostępnych dla tej coraz bardziej popularnej platformy.

Dowiesz się

- Jakie narzędzia można wykorzystać do testów integracyjnych
- Czym się różnią się te narzędzia
- Na co zwracać uwagę dobierając narzędzia do swojego projektu

Powinieneś wiedzieć

- Podstawowy testowania oprogramowania

Projekty, w których coraz częściej uczestniczę, charakteryzuje bogaty w funkcje webowy interfejs użytkownika. W dużej mierze powstaje on dzięki technologiom HTML5 i CSS3 a także możliwościom języka JavaScript. Jako tester, a czasami również scrum master spotykam się z problemem organizacji i koordynacji pracy w takim projekcie. Zespół tworzący aplikacje zwykle dzieli się on na trzy grupy; na barkach programistów JavaScript spoczywa wygląd i funkcjonalność webowej części aplikacji, część serwerowa to domena programistów Java, trzecią zaś grupę stanowią testerzy. Z racji różnic w znajomości języków programowania współpraca pomiędzy tymi grupami jest trudna. Odbija się to również na jakości testów automatycznych. Testerzy w takim zespole rzadko mogą liczyć na wsparcie programistów. Odpowiedzialni za układ strony a często również za problemy z testami regresyjnymi programiści JavaScript, rzadko znają języki wykorzystywane przez narzędzia testerskie. Z kolei programiści tworzący część serwerową nie mają wiedzy o zmianach zachodzących w interfejsie webowym. Wydaje się, iż rozwiązań tego problemu jest kilka. Jednym z pomysłów jakie mogłyby pomóc, jest wybranie narzędzi możliwych do wykorzystania również przez programistów JavaScript. Pojawiło się ich w ostatnim czasie kilka, co chciałbym poniżej opisać.

Testy integracyjne i akceptacyjne

Testy integracyjne i akceptacyjne aplikacji webowych w dużej mierze symulują akcje, jakie użytkownik może wykonać poprzez przeglądarkę internetową. Zadaniem narzędzi automatyzujących testowanie jest umożliwienie wykonywania takich testów bez udziału człowieka. To, jak dokładnie mogą być zasymulowane akcje użytkownika, zależy od możliwości danego narzędzia. Każde z przezentowanych narzędzi dysponuje podstawowym zestawem funkcji umożliwiającym klikanie elementów strony, wypełnianie formularzy (w tym wybieranie z list rozwijanych i zaznaczanie opcji) oraz pobieranie tekstów, atrybutów lub kodu html wybranych znaczników HTML. Zestaw ten może okazać się jednak niewystarczający przy testowaniu aplikacji opartych o JavaScript oraz HTML5. Możliwości języka JavaScript pozwalają na dużą ingerencję w standardowe zachowanie elementów strony, przez co mogą powodować problemy z działaniem podstawowych funkcji narzędzi testujących. Integracja z wykorzystywanym w projekcie szkieletem aplikacyjnym (ang. *framework*) powinna być jednym z ważnych kryteriów wyboru narzędzia. Warto również zwrócić uwagę na zestaw dodatkowych funkcji, które akurat dla naszej aplikacji mogą okazać się istotne.

W zależności od wymagań i budowy projektu, należy zwrócić uwagę na wsparcie narzędzia dla przeglądarek internetowych i środowisko pracy. Początkujący użytkownicy powinni wybrać narzędzia jednolite i proste w instalacji. Bardziej zaawansowani mogą wykorzystać rozwiązania o budowie modułowej. Dają one więcej możliwości jeśli chodzi o integrację z infrastrukturą dużych projektów. Niekiedy pozwalają również na integrację z rozwiązaniami wykorzystywanymi do testów

jednostkowych. Czynnikami jakie należałoby następnie rozważyć są: dokumentacja, wsparcie w przypadku problemów oraz perspektywy rozwoju.

CasperJS

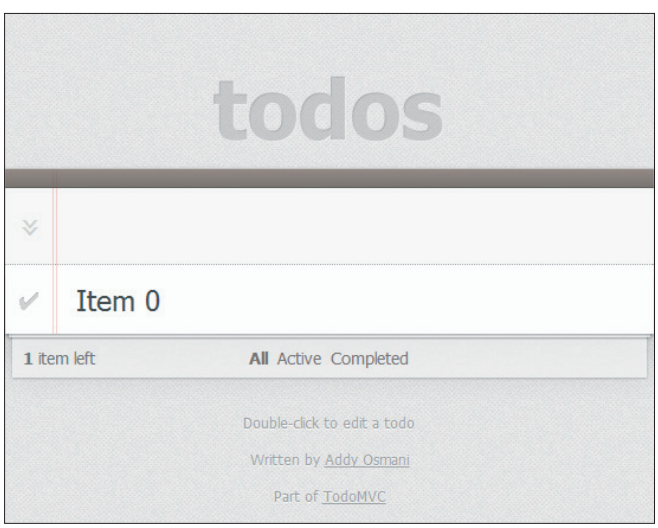
Pierwszym z prezentowanych narzędzi jest CasperJS [1]. Instalacja narzędzia wymaga pobrania archiwów opisanych w instrukcji, rozpakowania ich i uzupełnienia zamiennej PATH. Jest to narzędzie zbudowane w oparciu o projekt PhantomJS [2] – przeglądarkę internetową bez interfejsu graficznego (ang. headless) wykorzystującą silnik WebKit. W najnowszej wersji udostępniono testowo możliwość integracji z projektem SlimerJS [3], będącym rozwiązaniem podobnym do PhanotomJS, jednak wykorzystującym silnik Gecko. Niewątpliwą zaletą rozwiązań bez interfejsu graficznego jest możliwość uruchamiania ich w środowisku tekstowym. Może być to kluczowe dla projektów wykorzystujących serwery integracyjne, takie jak np.: Traivis. Mimo, iż mamy do czynienia z przeglądarką bez interfejsu graficznego, możliwe jest ustalenie rozmiaru okna w jakim będzie pracować aplikacja, jak również robienia zrzutów ekranu lub jego fragmentów.

CasperJS jako jedyne narzędzie umożliwia nasłuchiwanie zdarzeń przeglądarki i posiada wbudowane mechanizmy ułatwiające uwierzytelnianie zapytań. Pozwala także wysyłać zapytania HTTP, których nagłówki i treść mogą być ustalone przez użytkownika. W ten sposób możemy testować również część serwerową aplikacji symulując zapytania od różnych przeglądarek.

Jeśli chodzi o bardziej zaawansowane funkcje: mamy możliwość wykonywania kodu JavaScript w kontekście DOM testowanej aplikacji, a nawet wstrzykiwania całych bibliotek. Równie łatwo możemy zapisywać kod testowanych stron lub ich fragmentów.

Pisząc testy musimy opanować API oraz zestaw asercji jakie dostarczone są z CasperJS. Wyszukiwanie elementów odbywa się poprzez selectory CSS3 bądź XPath.

Pewnym mankamentem przy pisaniu testów jest konieczność deklarowania ilości asercji w teście oraz wywołania funkcji done() na końcu każdego z testów. W dużych projektach może być to kłopotliwe. Innym problemem dla korzystających z Caspera jest debug'owanie w środowisku bez interfejsu graficznego. Choć użytkownik ma do dyspozycji wbudowane mechanizmy ułatwiające tę czynność, praca z rzeczywistą przeglądarką wydaje się być łatwiejsza.



Rysunek 1. *Zrzut ekranu z CasperJS/PhantomJS*

Listing 1. *Przykładowy skrypt CasperJS*

```
casper.start("http://localhost:8080/", function() {
   this.test.assertTitle("Backbone.js • TodoMVC");});
casper.then(function() {
   this.sendKeys('#new-todo', "Item 1" );
   this.evaluate(function() {
      var e = jQuery.Event("keypress");
      e.which = 13; e.keyCode = 13;
      jQuery("#new-todo").trigger(e);
   });
   this.test.assertEvalEquals(function() {
      return __utils__.findOne('#new-todo').value;
   }, '', '#new-todo should be empty after pressing ENTER key.');
   this.test.assertSelectorHasText('ul#todo-list li:nth-child(1) label','Item 1');
   this.test.assertEval(function() {
      return __utils__.findAll("#todo-list .view label").length == 1;
   }, "Should be one item in the todo list.");});
casper.run(function() {
   this.test.done(5);
   this.exit();});
```

CasperJS to projekt zapoczątkowany w 2011 roku przez Nicolasa Perriault i od tego czasu zyskał już pewną popularność. Obecnie mamy do czynienia z wersją 1.1 co może wskazywać na pewną dojrzałość rozwiązania. Dokumentacja projektu jest spójna i jasna. Oprócz autora projekt rozwija grupa ok. sześćdziesięciu programistów. Statystyki stackoverflow.com pokazują, iż osoby zaangażowane w projekt, a szczególnie sam jego autor są bardzo pomocne użytkownikom.

DalekJS

DalekJS [4] to najmłodsze narzędzie z całej prezentowanej grupy. Uczestnicy konferencji Front-Trends 2013 mieli okazję zobaczyć prezentację tego narzędzia prowadzoną przez autora. Wzbudziła ona spore zainteresowanie, dlatego też postanowiłem je tutaj przedstawić. Podobnie jak CasperJS narzędzie daje możliwość pracy z przeglądarką PhantomJS (domyślna opcja) ale użytkownik ma jednak możliwość wykorzystania przeglądarek Chrome, IE oraz Firefox (muszą być one zainstalowane oddzielnie).

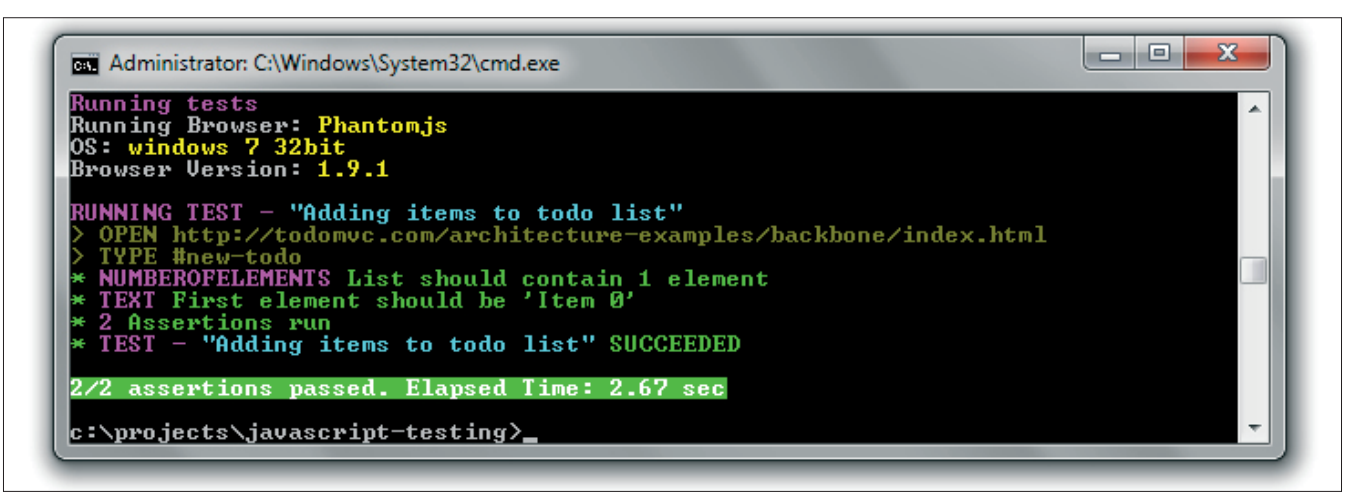
Narzędzie do pracy wymaga środowiska NodeJS [5]. Znajomość tego środowiska nie jest jednak wymagana. Autor zadbał o ukrycie niepotrzebnych szczegółów przed początkującym użytkownikiem. Osobom znającym to środowisko sposób organizacji i definiowania testów jednak na pewno wyda się znajomy.

Intencją autora było stowrzenie narzędzia prostego w użyciu. Porównując przykłady testów zamieszczone w artykule, rzeczywiście skrypt DalekJS jest najkrótszy i najbardziej czytelny. Do wskazywania elementów stron DalekJS wykorzystuje mechanizm znany z jQuery (nieznacznie różniący się od selectorów CSS).Test budowany jest z ciągu następujących po sobie wywołań akcji użytkownika i asercji, a kończy się wywołaniem funkcji done(). Interesujący jest też domyślny sposób prezentowanie raportu z testów. Do plusów rozwiązania należy zaliczyć wbudowaną obsługę alertów JavaScript i możliwość odczytywania danych z ciasteczek (ang. *cookies*).

Niestety DalekJS ma jeszcze kilka ograniczeń, w tym m.in. brak możliwości definiowania akcji wykonywanych przed i po każdym z testów. To też bardzo młody project. Wersja 0.0.1 została wydana zaledwie kilka tygodni temu. Jak pisze autor projektu: „Some features are missing, some things might change, there's a lot of optimisation to be done". Myślę, że warto jednak monitorować ten projekt.

WebdriverJS

Rozwiązanie WebDriver [8] również doczekało się swojej implementacji w środowisku JavaScript. Podobnie jak



Rysunek 2. *Raport z testów DalekJS*

Listing 2. *Przykładowy skrypt DalekJS*

```
module.exports = {
  "Adding items to todo list": function (test) {
  test
    .open("http://todomvc.com/architecture-examples/backbone/index.html")
    .type("#new-todo", "Item 0\n")
    .assert.numberOfElements("#todo-list .view label", 1)
    .assert.text("#todo-list .view label", "Item 0")
    .done();
  }
};
```

poprzednie narzędzie, do jego instalacji i pracy wymagane jest środowisko NodeJS. Oprócz instalacji samego narzędzia, wymagane jest także pobranie Selenium Server, a niekiedy również driverów do odpowiednich przeglądarek. Wymienione elementy pozwolą na interakcję z przeglądarką, nie dadzą jednak możliwości definiowania testów i tworzenia asercji. W tym celu wymagane jest zainstalowanie oddzielnego narzędzia, z których najpopularniejsze są Jasmine [7] i Mocha [8]. Oba pozwalają na tworzenie testów w stylu Behavior Driven Development przedstawiony na Przykładzie 3.

Użytkownikom bez dużego doświadczenia z językiem JavaScript pewne kłopoty może sprawić asynchroniczne API WebDriverJS. Warto zatem dokładnie zapoznać się z opisanym na stronie projektu mechanizmem promise [9]. W trakcie kilkutygodniowej pracy nie zauważyłem innych różnic w możliwościach tej implementacji w stosunku do wersji dla języka Java.

WebDriver ma ugruntowaną pozycję wśród testerów oprogramowania. Jest rozwiązaniem sprawdzonym i rozwijanym od kilku lat. Użytkownik ma szanse znaleźć w internecie olbrzymią ilość informacji na jego temat, a także rozwiązania wielu problemów. Nie ma zbyt wiele informacji na temat implementacji w JavaScript, ale tłumaczenie na ten język nie powinno być problemem. Jeśli chodzi o wspierane przeglądarki jest ono również najszersze. WebdriverJS współpracuje z:

- IE 8+
- Firefox 10+
- Chrome 12+
- Opera 12+
- Android 4.0+

Listing 3. *Organizacja testów w stylu BDD w WebdriverJS*

```
var assert = require('assert'),
     fs = require('fs'),
     webdriver = require('selenium-webdriver'),
     test = require('selenium-webdriver/testing'),
     remote = require('selenium-webdriver/remote');
var url = 'http://todomvc.com/architecture-examples/backbone/index.html';
test.describe('Adding items to todo list', function() {
  var driver;
  test.before(function() {
    driver = new webdriver.Builder().
      withCapabilities(webdriver.Capabilities.chrome()).
      build();
  });
  test.it('should add new item after pressing ENTER key', function() {
    driver.get(url);
    driver.findElement(webdriver.By.id('new-todo'))
      .sendKeys('Item 0' + webdriver.Key.ENTER);

    driver.wait(function() {
      return driver.findElements(
        webdriver.By.css('#todo-list .view label')).then(function(todos){
          return todos.length === 1;
   });
    }, 2000);

    driver.findElements(webdriver.By.css('#todo-list .view label'))
      .then(function(arr) {
        arr[0].getText().then(function(text) {
          assert.equal(text, 'Item 0');
        });
    });
  });
  test.after(function() { driver.quit(); });
});
```

REFERENCJE

- [1] „*CasperJS*," [Online]. Available: *http://casperjs.org/*
- [2] „*PhantomJS*," [Online]. Available: *http://phantomjs.org*
- [3] „*SlimerJS*," [Online]. Available: *http://slimerjs.org/*
- [4] „*DalekJS*," [Online]. Available: *http://dalekjs.com/*
- [5] „*Node.js Home Page*," [Online]. Available: *http://nodejs.org/*
- [6] „*WebDriverJS*," [Online]. Available: *http://code.google.com/p/selenium/wiki/WebDriverJs*
- [7] „*Jasmine Home Page*," [Online]. Available: *http://pivotal.github.io/jasmine/*
- [8] „*Mocka Home Page*," [Online]. Available: *http://visionmedia.github.io/mocha/*
- [9] „*Futures and Promises*," [Online]. Available: *http://en.wikipedia.org/wiki/Futures_and_promises*
- [10] „*Camme/Webdriver*," [Online]. Available: *https://github.com/camme/webdriverjs*
- [11] „*AngularJS*," [Online]. Available: *http://angularjs.org/*
- [12] „*Karma*," [Online]. Available: *http://karma-runner.github.io/*
- [13] „*E2E tests in Angular*," [Online]. Available: *http://docs.angularjs.org/guide/dev_guide.e2e-testing*
- [14] „*Matchers*," [Online]. Available: *https://github.com/pivotal/jasmine/wiki/Matchers*
- [15] „*Protractor on GitHub*," [Online]. Available: *https://github.com/angular/protractor*

W aspekcie dużego i długotrwałego projektu wydaje się on najlepszym rozwiązaniem.

Warto również zwrócić uwagę na projekt Camme/Webdriver [10]. Jest to projekt wykorzystujący, podobnie jak WebDriverJS, protokół JSON-Wire do komunikacji z przeglądarką. Dostarcza nieco innego interfejsu programistycznego, nie jest jednak oficjalną implementacją. Znajomość różnic pozwoli świadomie wybrać najbardziej odpowiednie do danego projektu narzędzie.

Testy aplikacji wykorzystujących AngularJS

Oddzielny fragment chciałbym poświęcić testowaniu aplikacji wykorzystujących, wspierany przez firmę Google, szkielet aplikacyjny AngularJS [11]. AngularJS posiada własne rozwiązanie do testowania o nazwie Karma [12](wcześniej Testacular). Jest to narzędzie pozwalające na uruchamianie testów (ang. Test Runner) w środowisku przeglądarki internetowej. Karma dostarcza infrastrukturę potrzebną do uruchamiania zarówno testów jednostkowych jak i integracyjnych. Bazuje jednak na rozwiązaniu innym niż narzędzia prezentowane wcześniej. Kod testu wykonywany jest w całości przez przeglądarkę; a strony, na których symulowane są akcje użytkownika wczytywane są poprzez znacznik iframe.

AngularJS dostarczany jest wraz z modułem ngScenario [13], który ma umożliwić symulowanie akcji użytkownika oraz ocenę poprawności działania aplikacji. Moduł ten sprawdza się dobrze w przypadku testów prostych aplikacji. Praca z bardziej złożonymi i dynamicznymi scenariuszami wymaga poznania szczegółów implementacji mechanizmu Matchers [14]. Niestety brak jest informacji na ten temat w dokumentacji i użytkownik zmuszony jest do przeanalizowania kodu modułu. Problemy pojawiają się również dalej, gdyż część aplikacji nie jest oparta o AngularJS (np. strona logowania). W niektórych przypadkach może to nawet uniemożliwić wykonywanie testów automatycznych.

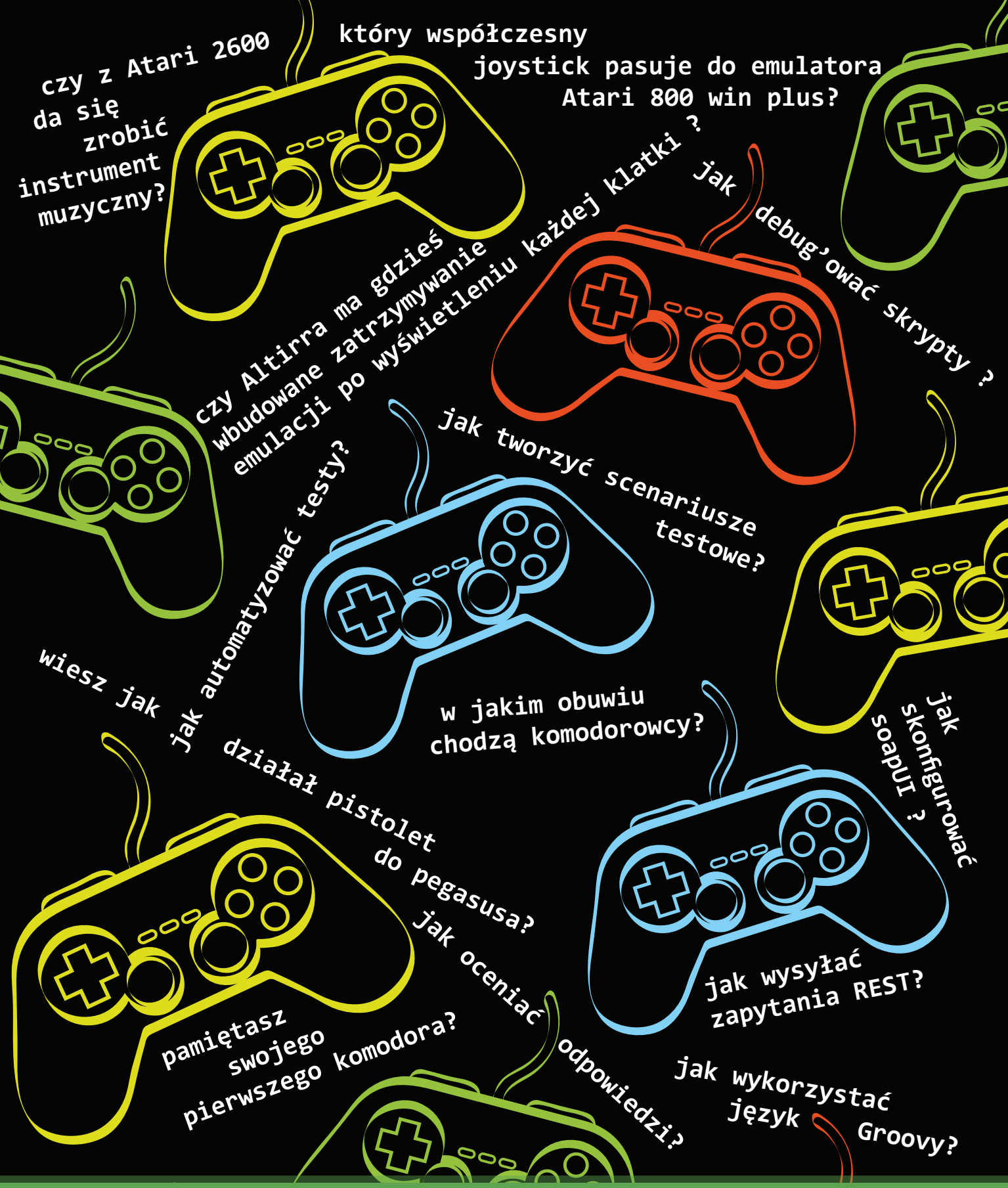
Google rozwija aktualnie kolejne narzędzie do testów integracyjnych, które ma zastąpić Karmę i ngScenario w kolejnej wersji Angular'a. Projekt nazywa się Protractor [15] i bazuje na rozwiązaniu Selenium-Webdriver. Rozszerza on standardowe mechanizmy wyszukiwania elementów o typowe dla Angular'a wyszukiwanie po dowiązaniach. Pozwala na lepszą synchronizację pomiędzy testem a aplikacją. Planującym testy aplikacji wykorzystujących AngularJS jak najbardziej polecam zapoznanie się z nim.

Podsumowanie

Z uwagi na zakres omawianego tematu, artykuł prezentuje narzędzia w sposób bardzo ogólny. Więcej szczegółów czytelnik może znaleźć na stronach projektów, których adresy wymienione są w z referencjach. Mam nadzieję, iż ułatwi czytelnikowi wybrać najbardziej odpowiednie dla własnej pracy narzędzie.

JACEK OKROJEK

Tester, koordynator i kierownik testów z wieloletnim doświadczeniem w testowaniu systemów wysokiej dostępności. Jako konsultant do spraw zapewnienia jakości prowadził i uczestniczył w wielu złożonych projektach dla klientów z sektora usług medycznych oraz telekomunikacyjnych, a obecnie bankowości inwestycyjnej. Pracował w obszarze testów integracyjnych, systemowych oraz akceptacyjnych. Autor rozwiązań automatyzujących proces testowania oprogramowania doceniający wagę testowania eksploracyjnego. Entuzjasta dynamicznych metod wytwarzania i testowania oprogramowania.



Bezpłatne warsztaty dla testerów
26.10.2013 hotel WESTIN, WARSZAWA

www.facebook.com/KarieraTestera

Testowanie
złożonych systemów telekomunikacyjnych

Kraków, laboratorium testów systemowych marzec 2011
Według specyfikacji próbujemy na 8000 użytkowników. No dawaj... zobaczymy jak się zachowa. (...) Przeszło, bardzo dobrze... No to teraz 10000. To już sporo powyżej wymagań, ale musimy to sprawdzić, tak na wszelki wypadek. (...) No popatrz, wypadły te powyżej 9500, trzeba to zgłosić zespołowi do poprawy...

Wielka Brytania, centrum dyspozytorskie policji czerwiec 2012
Trzeba wzmocnić ochronę w tym rejonie przed zawodami. Będzie masa kibiców... Mamy tam ok. 6000 ludzi, ale potrzebujemy kolejne 4000. Najlepiej wszyscy z radiami, żeby była pełna łączność... System powinien wytrzymać...

Przedstawiony powyżej przykład to tylko jeden z wielu, w przypadku krytycznych sieci radiowych, gdzie bezpieczeństwo zdrowia i życia ludzkiego może zależeć od tego, jak dokładnie i efektywnie zostało przetestowane dane rozwiązanie. Z punktu widzenia dowódcy w centrum dyspozytorskim operacja przegrupowania oddziałów lub wezwania posiłków na dany obszar geograficzny powinna być tak płynna jak to tylko możliwe i mieć minimalny wpływ na działającą tam sieć radiową.

Natomiast z punktu widzenia wnętrza tej sieci, tego typu operacja oznacza dość nagły wzrost obciążenia ruchem w określonym rejonie i by przebiegło to bez żadnych problemów – wszystkie warstwy, komponenty i podsystemy sieci muszą idealnie współgrać ze sobą. Stąd, poza oczywiście zaprojektowaniem sieci pod kątem takich przeciążeń, tak istotne jest odpowiednie przetestowanie oprogramowania wchodzącego w jej skład na każdym możliwym poziomie.

Ogólna charakterystyka systemów telekomunikacyjnych

Duże systemy telekomunikacyjne, w tym również te z przykładu powyżej, działające w standardzie TETRA czy ASTRO, mają kilka cech, które nie ułatwiają zadania testerom. Pomijając sam rozmiar (duża ilość różnych urządzeń składających się na system), są one hybrydowe, łączące w sobie wiele różnych, często diametralnie różnych, elementów.

Planowanie i wykonywanie testów systemów bezpieczeństwa publicznego musi zatem wziąć pod uwagę następujące różnice:

- **Technologiczne** – urządzenia pracują na różnych platformach sprzętowych, pod różnymi systemami operacyjnymi, wliczając w to środowiska zwirtualizowane. Napisane są w różnych językach programowania, także skryptowych czy z wykorzystaniem narzędzi modelujących.

- **Funkcjonalne** – na system składają się aplikacje czasu rzeczywistego, zarówno o miękkich ograniczeniach czasowych („soft RT") jak i twardych („hard RT"), pracujące razem z klasycznymi aplikacjami klienckimi; aplikacje wbudowane (embedded) oraz aplikacje typu klient-serwer; sieci transportowe, urządzenia infrastruktury i użytkowników końcowych (PC-ty, tablety, telefony).
- **Organizacyjne** – zespoły dzielą się odpowiedzialnością za podsystemy, lub nawet za poszczególne urządzenia. Dodatkowo fakt, że często zespoły znajdują się w różnych ośrodkach, nierzadko na różnych kontynentach, od Ameryki Północnej, przez Europę do Azji i we wszystkich strefach czasowych, sprawia, że istnieje ryzyko tworzenia się tzw. silosów organizacyjnych.

Typowy system telekomunikacyjny (Rysunek 1), składa się, zgodnie z modelem zarządzania siecią telekomunikacyjną TMN, z następujących warstw:

- **RAN** (*Radio Access Network*) – radiowa sieć dostępowa, w tym podsystemy zarządzające połączeniami i zasobami wymaganymi do ich realizacji,
- **EMS** (*Element Management System*) – warstwa zarządzająca poszczególnymi elementami w sieci,
- **NMS** (*Network Management System*) – warstwa wyższego poziomu, zarządzająca i monitorująca całą sieć (lub sieci), często według modelu FCAPS (z angielskiego: Fault, Configuration, Accounting, Performance, Security Management),
- **B/OSS** (*Business/Operations Support System*) – systemy sieci (i biznesu), wspierające procesy i działania klientów.

Równocześnie obok formalnych warstw, system dzieli się na wiele podsystemów, odpowiedzialnych za poszczególne funkcjonalności, jak rozmowy głosowe, transmisja danych, szyfrowanie i uwierzytelnianie, usługi lokalizacyjne i wiele innych.

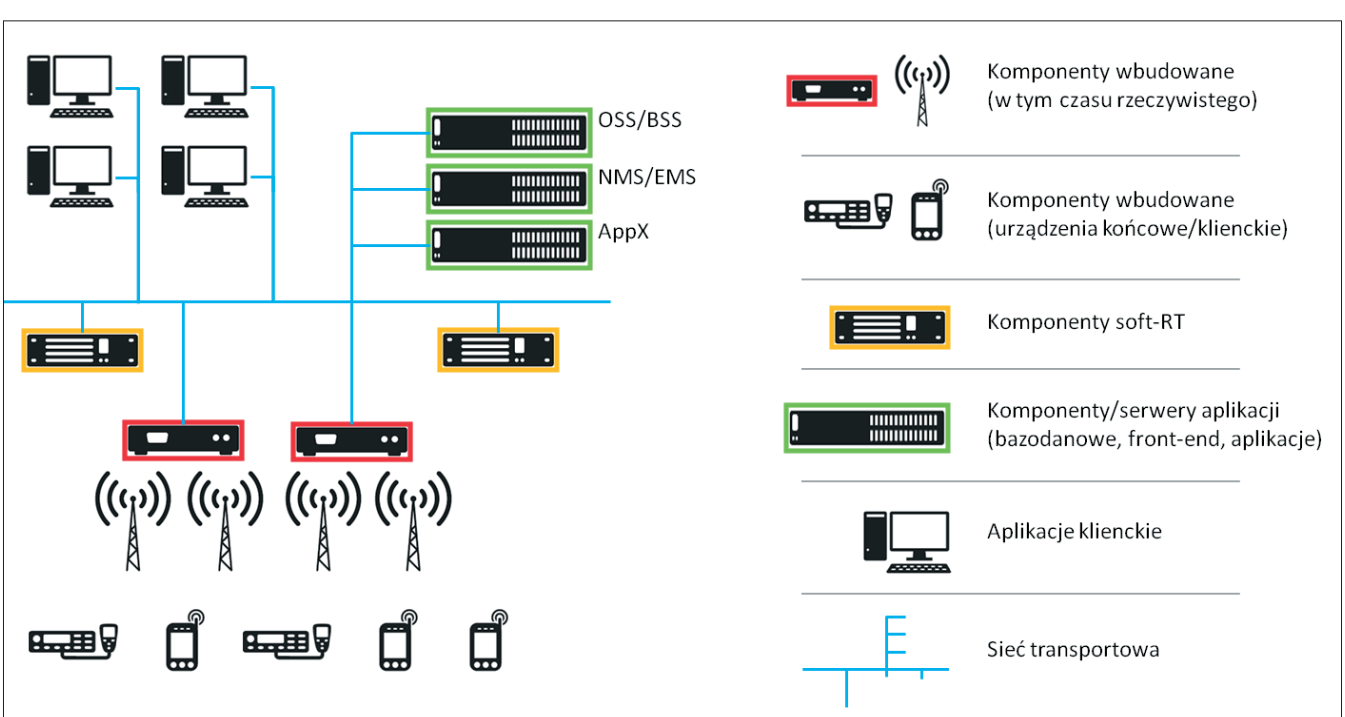
Jak to przetestować?

Ze względu na złożoność proces testowania takich systemów składa się z wielu poziomów lub faz. Międzynarodowy standard SWEBOK (Software Engineering Body of Knowledge) wyszczególnia trzy główne poziomy testowania w procesie wytwarzania oprogramowania: jednostkowe, integracyjne i systemowe. Mniej oficjalnie wyszczególnia się też inne fazy, głównie ze względu na cel, który im przyświeca.

Każdy kolejny (wyższy) poziom testowania posiada większą szansę wykrycia braku funkcjonalności lub jej błędnego wykonania, lecz jednocześnie dostarcza coraz bardziej rozmyte informacje na temat przyczyn wystąpienia błędu.

Testy jednostkowe

Większość programistów doskonale zdaje sobie sprawę z potrzeby pisania testów jednostkowych. Najbardziej popularny chyba rodzaj testów, tworzony już w fazie implementacji, najczęściej testuje przekazywane parametry, zwracane wartości i logikę funkcji. Czasem, zależnie od tego jak zdefiniujemy jednostkę, pojawiają się też testy komponentowe, gdzie testujemy przepływy danych



Rysunek 1. *System bezpieczeństwa publicznego jako złożony system telekomunikacyjny*

pomiędzy modułami i interfejsy w bardzo prostych scenariuszach.

Testy funkcjonalne (niestrukturalne)

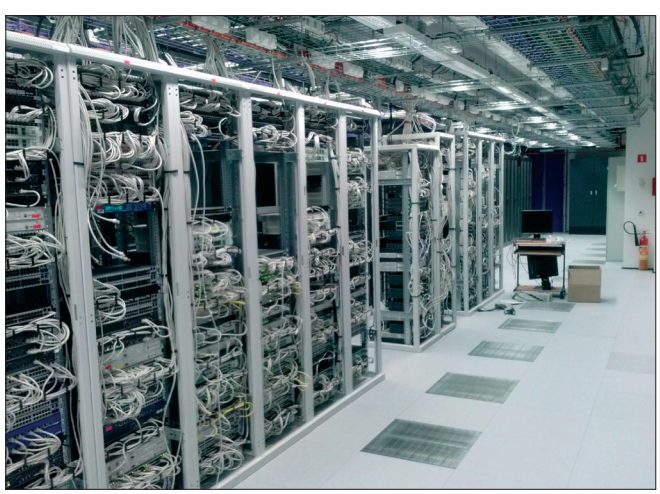
Po przejściu pierwszych bramek jakości mamy działającą aplikację. Czas na pierwsze testy funkcjonalne czyli testy czarnej skrzynki (twz. „black-box testing"). W odróżnieniu od testów jednostkowych, bardzo często wykonywane są przez inne osoby niż te, które pisały kod. Testerzy już na tym etapie testują funkcjonalności w oparciu o scenariusze zdefiniowane przez architektów, klientów lub dokumentację, nie mając wiedzy o wewnętrznej budowie aplikacji. W tej fazie najczęściej mamy optymalną równowagę pomiędzy szansą na wykrycie błędu, a szczegółowością danych na jego temat. Dzięki temu w tej fazie najczęściej znajduje i reprodukuje się defekty. To z kolei, oraz fakt, że testy funkcjonalne na tym poziomie są w większości wysoce zautomatyzowane, sprawia, że często są one źródłem regresji dla aplikacji. Ciekawą możliwością dla aplikacji wbudowanych (tzw. embedded) jest praca poza urządzeniem docelowym (tzw. off-target). Przeniesienie aplikacji z jej platformy sprzętowej do środowiska (aplikacje czasu rzeczywistego mogą wykorzystać np. Linuxa z jądrem RT), którego używają programiści w swojej codziennej pracy, umożliwia im weryfikację funkcjonalności już na wczesnym etapie implementacji. Kompromisem, który trzeba ponieść, jest najczęściej rozluźnienie zależności czasowych (występujących w systemach RT), przez co taka praca nigdy w pełni nie zastąpi prawdziwych testów niestrukturalnych wykonywanych na platformie docelowej.

Testy integracyjne

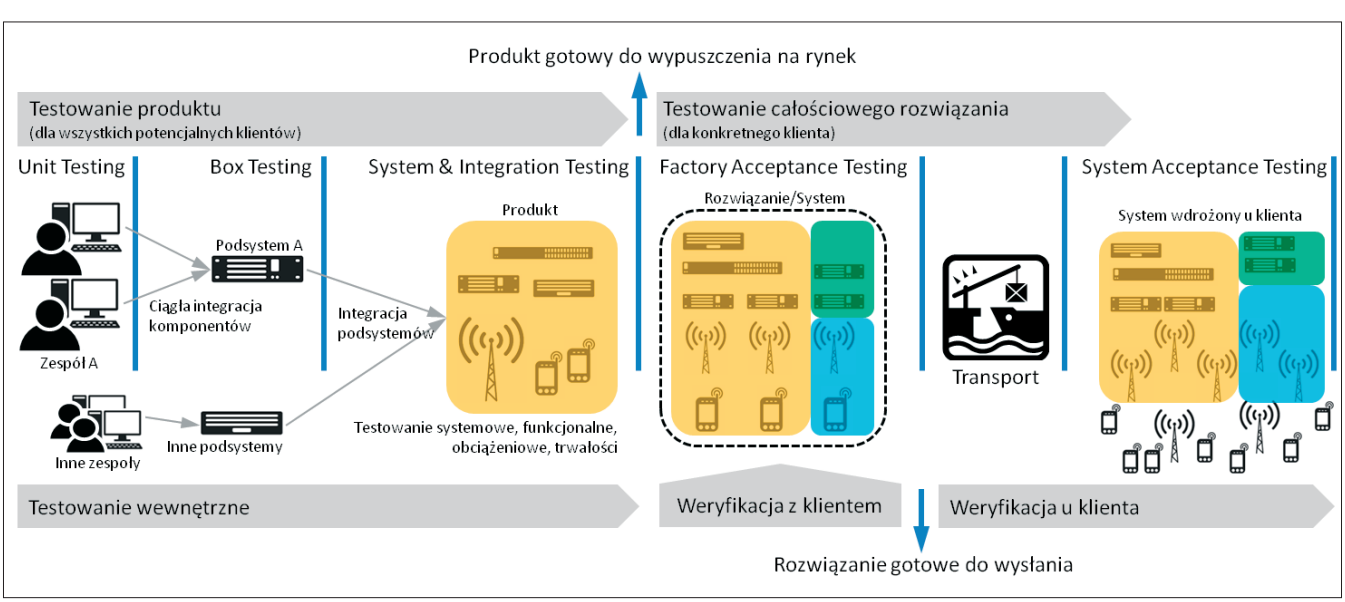
Ponieważ w dużej organizacji aplikacje często rozwijane są przez osobne zespoły warto, zadbać aby ich integracja przebiegła sprawnie. Testy systemowe (zwane żartobliwie wielkim wybuchem – „big bang"), o których za chwilę, integrują ze sobą jednocześnie cały system i wszystkie aplikacje. Testy integracyjne, lub wczesnej integracji, to podejście iteracyjne, mające na celu wczesne wychwytywanie błędów w projekcie interfejsów tak, by w stosunkowo kosztownych testach systemowych nie tracić czasu na znajdowanie i rozwiązywanie w sumie prostych problemów.

Testy systemowe

Kiedy cała funkcjonalność jest już zaprojektowana, zaimplementowana i wstępnie przetestowana, przechodzimy do testów systemowych. Ponieważ integrujemy tutaj ze sobą wszystkie warstwy i podsystemy jednocześnie, odbywają się one w dedykowanych laboratoriach, z dość ścisłym reżimem kontroli wersji oprogramowania i konfiguracji. Głównym celem integracji systemowej jest zweryfikowanie, że całość działa według projektu. Przeprowadzana jest analiza wymagań i ich pokrycia. Testowane jest nie tylko oprogramowanie, ale również procedury konfiguracyjne, instalacyjne oraz sprawdzana jest poprawność dokumentacji systemu.



Rysunek 3. *Laboratorium testów systemowych*



Rysunek 2. *Typowe fazy procesu testowania zaawansowanego rozwiązania telekomunikacyjnego*

Testy wydajnościowe

Ciekawym rodzajem testów systemowych są testy obciążeniowe (load), wydajnościowe (performance) i trwałości (longevity). Systemy telekomunikacyjne pracują zazwyczaj z dużą ilością użytkowników, przy zróżnicowanym obciążeniu i w sposób ciągły. Posiadają zdefiniowane profile użytkowania i ilość jednoczesnych rozmów. Co odróżnia systemy bezpieczeństwa od komercyjnych sieci komórkowych to oczekiwanie klientów, że będą funkcjonować w każdych warunkach. W przypadkach niespodziewanych wydarzeń, jak choćby uderzenie huraganu Katrina czy Sandy, czy nawet większej imprezy sportowej jak w początkowym przykładzie, kiedy sieci komórkowe są przeładowane lub uszkodzona mogła zostać infrastruktura – pracujący policjanci, strażacy czy służby medyczne muszą mieć zapewnioną komunikację. W tym celu przeprowadzane są testy symulujące duże obciążenie czy reakcje systemu na ruch wielokrotnie przekraczający specyfikację systemową.

FAT (Factory Acceptance Testing)

Po wykonaniu testów systemowych gotowy produkt może już trafić do konkretnych klientów. Klienci zamawiają dany system i wtedy rozpoczyna się proces jego budowania w tzw. „staging area". Skompletowany zostaje odpowiedni sprzęt (szafy, serwery, urządzenia sieci transportowej), zainstalowane oprogramowanie w odpowiednich wersjach, skonfigurowana sieć transportowa i sam system. Ponieważ różni klienci mają różne wymagania i w różny sposób planują wykorzystywać system, mogą go skonfigurować na wiele różnych sposobów, rozszerzyć o różne dodatkowe funkcjonalności, itd. Na tym etapie pojawia się więc konieczność wykonania dodatkowych testów na zbudowanym już systemie, według scenariuszy, które może dostarczyć sam klient. Są to tzw. testy FAT i często klient bezpośrednio w nich uczestniczy (tzw. „witnessed testing"). Najczęściej akceptacja wyników testów FAT jest warunkiem koniecznym do spełnienia zanim system zostanie wysłany w miejsce docelowe.

SAT (System Acceptance Testing)

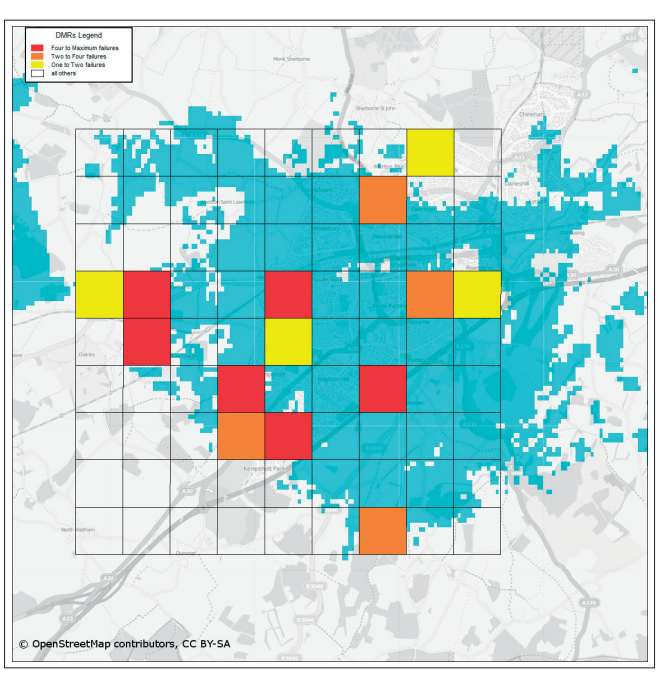
Kiedy system dociera na miejsce przeznaczenia i zostaje odbudowany – często następuje wtedy integracja z różnymi rozwiązaniami firm trzecich i w przypadku sieci radiowych – często po raz pierwszy system rozpoczyna pracę z prawdziwymi, rozlokowanymi w terenie urządzeniami sieci dostępowej (np. stacjami bazowymi). Ponieważ dostarczamy pełne rozwiązanie – zależy nam też na zapewnieniu, że system funkcjonuje poprawnie po transporcie, odbudowie i integracji z innymi podsystemami. Na tym etapie rozpoczyna się więc cykl tzw. testów SAT, poszerzonych w stosunku do FAT o dodatkowe scenariusze, niemożliwe do przetestowania w „staging area" (np. ze względu na ograniczenia geograficzne tej lokalizacji, brak możliwości podłączenia większej ilości stacji bazowych, itd.). Akceptacja wyników testów SAT przez klienta otwiera możliwość wykorzystania systemu w trybie produkcyjnym.

Drive testy, testy jakościowe

W przypadku systemów łączności radiowej krytyczne jest zapewnienie odpowiedniego pokrycia (mocy sygnału) na zadanym obszarze geograficznym oraz odpowiedniej jakości dźwięku pomiędzy urządzeniami końcowymi (np. radio do radia, czy radio do stanowiska dyspozytorskiego).

Cała sieć jest oczywiście projektowana (z symulacją pokrycia) odpowiednio wcześniej - na etapie planowania radiowego. Ponieważ jednak projektowanie to odbywa się

Rysunek 4. *KillerBox - przykład wykorzystania wielu urządzeń końcowych do symulacji wysokiego obciążenia sieci podczas testów systemowych*



Rysunek 5. *Przykładowy wynik testów pokrycia na zadanym obszarze geograficznym*

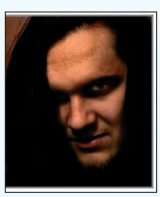
przy użyciu modeli matematycznych – jakkolwiek dobre by one nie były, w przypadku gdy klient chce upewnić się czy posiada pokrycie wszędzie tam gdzie zakładał, konieczna jest jego weryfikacja w terenie. Często w tym celu wykonuje sie tzw. „drive testy", gdzie technicy jeżdżą odpowiednio przygotowanymi pojazdami z aparaturą pomiarową i mierzą moc sygnału. Wyniki skojarzone z pozycją geograficzna (w oparciu o GPS) są opracowywane i przedstawiane na mapach (jak na przykładowym rysunku powyżej). Analiza jakości dźwięku jest realizowana np. poprzez przesłanie odpowiednio przygotowanej próbki do sieci radiowej z jednego urządzenia końcowego, odebranie na innym urządzeniu końcowymi i porównanie. Wyniki z „drive testów" służą do dalszej optymalizacji sieci.

Podsumowanie

Jak widzimy testowanie złożonych systemów telekomunikacyjnych, jakimi są m.in. systemy bezpieczeństwa publicznego, to bardzo szerokie zagadnienie. Różne metody testowania są wpisane w różne etapy implementacji oprogramowania: od testów jednostkowych pisanych i wykonywanych przez jednego programistę (które większość z Was pewnie dobrze zna), po testy projektowane i pisane przez cały zespół testów systemowych, na walidacji rozwiązania z klientem skończywszy.

W procesie powstawania systemów telekomunikacyjnych, krakowski ośrodek Motorola Solutions nie jest wyjątkiem, inżynierowie zajmują się testami na każdym z tych etapów. Piszą scenariusze, definiują strategie testowania, implementują narzędzia (m.in. symulatory, analizatory ruchu sieciowego) i dedykowane rozwiązania do testów systemowych i obciążeniowych. Zdarza się, że testerzy piszą więcej kodu niż programiści biorący udział w rozwijaniu produktu, co przełamuje stereotyp pracy i roli testera i wzbudza zazdrość kolegów. Testerzy biorą również udział w testach FAT i SAT, wspierając konkretne zespoły wdrożeniowe na całym świecie swoją wiedzą techniczną o systemach.

O AUTORACH



Andrzej Lichnerowicz – od 8 lat w Motorola Solutions. Architekt stacji bazowych Astro w ośrodku R&D w Krakowie.

Tomek Wojtowicz - od 10 lat w Motorola Solutions. Architekt systemów Tetra w ośrodku R&D w Krakowie.

Targi Pracy

Przetestuj nas
i znajdź pracę jako

TESTER OPROGRAMOWANIA

Organizujemy cykliczne Targi Pracy

poświęcone konkretnym działom IT.

Dzięki targom pracy możesz odnaleźć

oferty pracy jako Tester Oprogramowania,

ciekawe szkolenia i konferncje,

promocyjne zniżki u naszych partnerów.

Koordynacja Targów Pracy

mail *WirtualneTargi@pracaIT.com*
mail *WirtualneTargi@software.com.pl*
tel. 799 46 34 76

O outsourcingu i wynikających z niego korzyściach

O outsourcingu i wynikających z niego korzyściach dla pracowników opowiada Marcin Machnio, Country Manager w firmie NATEK Poland

natek
work IT with us

Rynek outsourcingu IT w Polsce jest...
Marcin Machnio: Dojarzały. Czerpał inspiracje głównie z Europy Zachodniej, implementując ciekawe rozwiązania, przyciągając światowych liderów. Szybko stał się popularny, a Polska jest obecnie outsourcingowym centrum Europy.

Dlaczego jest to tak popularny model zatrudnienia?
M.M: Ponieważ firma outsourcingowa, jako pracodawca, znacząco angażuje się w rozwój swojego pracownika, oferując mu możliwości niedostępne w innej formie zatrudnienia. Chodzi tutaj głównie o elastyczność oraz dopasowanie do konkretnych wymagań pracowniczych.

A jak to wygląda od strony firmy korzystającej z usług outsourcingowych?
M.M: Outsourcing musi być odpowiedzią na oczekiwania obu stron. Taka forma pozwala znacznie ograniczyć koszty związane z funkcjonowaniem własnego działu HR oraz IT. Skrócony i uproszczony zostaje proces selekcji, rekrutacji, penetracji trudnego rynku oraz doglądania spraw pracowniczych. Te obowiązki bierze na siebie firma outsourcingowa. Zarządzanie projektami staje się znacznie prostsze i tańsze dzięki takiemu wsparciu.

Część kandydatów może mieć obawy o stabilność takiego zatrudnienia. Jest się czego obawiać?
M.M: Wręcz przeciwnie. Firma outsourcingowa realizuje dużo projektów u wielu partnerów, dlatego nawet w przypadku zakończenia któregoś z nich, pracownik otrzymuje propozycję kolejnych. Poprzez takie działanie oferujemy stabilność zatrudnienia oraz nowe wyzwania.

Mógłby Pan to rozwinąć?
M.M: Oczywiście. Naszym zadaniem jest współplanowanie rozwoju kariery pracowników przy zachowaniu indywidualnego podejścia. Konsultant, któremu w ciągu najbliższych miesięcy może zakończyć się projekt, otrzymuje od nas kolejne oferty dopasowane do jego profilu. Oczywiście, może zdarzyć się sytuacja, że żadna nie spełni jego oczekiwań. Wtedy często decydujemy się na zatrudnienie w naszych wewnętrznych

projektach, mając w perspektywie kolejne, które mogą się dla niego pojawić. Naszym celem jest to, by raz zatrudniony pracownik już z nami pozostał, ponieważ skuteczność naszej działalności opiera się w stu procentach na ludziach. Bez nich nigdy nie udałoby się zbudować firmy o solidnych fundamentach.

Co jeszcze może zachęcić do outsourcingu, na przykład w NATEK?
M.M: Wielość i różnorodność projektów skupionych jedynie na innowacyjnych rozwiązaniach oraz najnowocześniejszych technologiach. Bezproblemowa migracja pomiędzy projektami, a więc rozwój i nowe możliwości.

Pełne wsparcie działu HR oraz opiekunów, zespołu, który wesprze w rozwoju i dopasuje pracę do aspiracji, wymagań czy określonej sytuacji życiowej. Sam proces rekrutacyjny jest często sprawniejszy i szybszy. Wielu z naszych pracowników wśród zalet wymienia także kwestie finansowe oraz międzynarodowe środowisko pracy.

Co Pan ma na myśli mówiąc o międzynarodowym środowisku pracy?
M.M: Przy naszych projektach pracują specjaliści z różnych krajów, a to doskonała okazja do wymiany wiedzy, poznania innych kultur i podszkoleniu się w użyciu „żywego" języka angielskiego. Wiele z takich znajomości z czasem przeradza się w przyjaźnie.

Relokacja do innych krajów nie stanowi dla nich przeszkody?
M.M: Przeważnie relokacja nie stanowi przeszkody, zarówno w ramach tego samego kraju czy również za granicę. Od wielu lat wspieramy rozwój kariery naszych pracowników, kompleksowo pomagając w relokacji. Służymy radą, zajmujemy się formalnościami, oferujemy specjalne pakiety relokacyjne. NATEK jest firmą międzynarodową, ale nie jesteśmy korporacją posiadającą siedziby na każdym kontynencie. Skupiliśmy się na rynku Europy Środkowo-Wschodniej, który dobrze znamy.

Wspomniał Pan, że pracownicy są też zadowoleni z kwestii finansowych...
M.M: Wynagrodzenia osób zatrudnionych w modelu outsourcingowym są atrakcyjne. Na ich wysokość wpływa kilka elementów, między innymi forma zatrudnienia. Większość naszych pracowników zatrudnionych jest na umowę o pracę i tę formę preferujemy. Istnieje również możliwość współpracy na zasadzie modelu firma-firma, powszechnie zwanego freelance. Czasami jest to jedyna możliwość realizacji projektu dla klienta finalnego, który nie zatrudnia inaczej niż na umowę o pracę.

A więc Wasz klient musi zapłacić za takiego pracownika więcej? Wcześniej wspominał Pan o oszczędnościach...
M.M: I to się nie zmienia. Suma oszczędności na wymienionych wcześniej kosztach jest istotnie wyższa niż różnica w wynagrodzeniach. Dodajmy do tego jeszcze ułatwione zarządzanie oraz oszczędność czasu i zasobów własnych. To wszystko pozwala spojrzeć na współpracę w dużo szerszej, korzystnej perspektywie.

Myślę, że wielu czytelników mogłoby rozważyć pracę w outsourcingu. Czy ma Pan jakieś rady dla takich osób?
M.M: Dobry pracodawca to ten, który potrafi spełnić oczekiwania swojego pracownika. Firma outsourcingowa, dzięki bogatemu portfolio swoich klientów, ma większe szanse by to osiągnąć. Dlatego warto sprawdzić, co może mieć do zaoferowania.

Najlepiej poszukiwać tych informacji u ich źródła, podczas bezpośredniego kontaktu. Zwróćmy uwagę na „miękkie" aspekty współpracy z firmą outsourcingową, czyli łatwość w komunikacji z zespołem, jeszcze zanim podejmiemy w nim pracę. Czy przyszły pracodawca jest otwarty i gotowy podjąć dyskusję o warunkach pracy? Czy nie będę pozostawiony sam sobie, czy zostanie mi przypisany opiekun, który zaangażuje się w bieżące sprawy? Czy pracodawca pomoże w ewentualnej relokacji? Trzeba na takie informacje uzyskać odpowiedź i ją przemyśleć.

Rzetelny kontakt z firmą już na etapie wstępnych rozmów jest tutaj kluczowy.

Jak więc znaleźć najlepszego pracodawcę?
M.M: Polecam szczególnie bezpośrednie metody kontaktu, na przykład poprzez portale społecznościowe, takie jak Goldenline, LinkedIN, a nawet Facebook. Firmy często prowadzą w nich swoje profile, umieszczają bezpośrednie kontakty do konkretnych osób – wystarczy wybrać i napisać. Cenię również uczestnictwo w targach pracy ukierunkowanych na rynek IT – to doskonała okazja do osobistego spotkania z przedstawicielami takiej firmy, rozmowy twarzą w twarz.

Dla zainteresowanych tematyką i poszukujących większej ilosci informacji, zamieszczamy kontakt do firmy NATEK Poland.
Email: poland@natek.pl

Zlokalizuj i zlikwiduj buga

szybko, po cichu i na jego terenie!

Testy jednostkowe (junity) są bardzo ważnym elementem procesu wytwarzania oprogramowania. Niestety bardzo często zaniedbywanym! W wielu projektach w ogóle nie ma junitów, jest ich bardzo mało lub są pisane w sposób, który daje niewiele korzyści.

Dowiecie się

- co to są testy jednostkowe
- jakie reguły powinny spełniać
- dlaczego warto je pisać
- co powinno być przetestowane
- jak pisać jUnity
- co to jest TDD
- co to jest pokrycie kodu
- jakie narzędzia pomogą nam utrzymać wysoką jakość kodu

Test jednostkowy jest kawałkiem kodu, którego wykonanie ma na celu sprawdzenie czy mały fragment aplikacji działa zgodnie z oczekiwaniami. Kent Beck i Erich Gamma w 1995 roku rozpoczęli pracę nad frameworkiem (strukturą) jUnit. Jak bardzo stał się on popularny od tamtego czasu, doskonale obrazuje stwierdzenie Martina Fowlera: „Nigdy w dziedzinie oprogramowania tak wielu nie zawdzięczało tak wiele tak niewielu liniom kodu". Każdy programista pracuje w pewnym cyklu. Koduje, kompiluje, uruchamia i testuje. Dzięki testom jednostkowym ostatnia część cyklu pozwala zaoszczędzić sporo czasu (poprzez wykrycie błędu zaraz po jego utworzeniu). Po zaimplementowaniu kolejnych funkcjonalności junity zapewniają nas, że nowostworzone funkcje nie wpływają w sposób niepożądany na już istniejące części systemu.

Teoria mówi, że kod powinien być pisany tak, aby był łatwo testowalny. W praktyce nie zawsze jest to takie proste. Ten problem ujawnia się w systemach, które powstały wcześniej, niż stworzono do nich testy, więc są one niezgodne z zasadami TDD (szczegóły w dalszej części artykułu). Prawidłowa struktura testu jednostkowego:

- *Setup* – ustawienie systemu testowego w stan wymagany przez test;
- *Execution* – wykonanie testowanej akcji i przechwycenie wyników;
- *Validation* – zweryfikowanie wyników;
- *Cleanup* – przywrócenie środowiska testowego do stanu sprzed wykonania testu.

Test jednostkowy powinien spełniać następujące właściwości:

- krótki;
- zautomatyzowany;
- wykonuje mały fragment kodu (z reguły pojedynczą gałąź metody);
- ma pełną kontrolę nad obiektami w teście i jest niezależny od wszystkiego;
- wykonuje się w bardzo krótkim czasie (milisekundy na test);
- dostarcza precyzyjną informację o tym, dlaczego test nie przeszedł.

Test nie jest testem jednostkowym, jeśli komunikuje się:

- z bazą danych;
- poprzez sieć;
- z systemem plików.

Taki test jest testem integracyjnym. Ze względu na swą czasochłonność, tego typu testy powinny być odseparowane od pozostałych testów jednostkowych i wyko-

nywane rzadziej.

Jak wygląda szablon klasy testu jednostkowego? Przykład poniżej:

```
package pl.sagiton.junit.sample;
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;
public class SimpleTest {
   @Test
   Public void shouldBeEmptyCollection() {
      Collection collection = new ArrayList();
      assertTrue(collection.isEmpty());
   }
}
```

Przykład ustawiania i resetowania zmiennych przed i po wykonaniu każdego testu:

```
private Collection<Object> collection;
private File file;
@BeforeClass
public static void oneTimeSetUp() {
   //one-time initialization code
}
@AfterClass
public static void oneTimeTearDown() {
   //one-time cleanup code
}
@Before
public void setup() {
   collection = new ArrayList<Object>();
   file = new File(…);
}
@After
public void cleanup() {
collection = null;
   file.delete();
}
```

Metoda oznakowana anotacją `@Before` jest wywoływana przed każdym testem, a metoda `@After` – po każdym teście. Pozwala to przygotować zmienne dla testów i ewentualnie po nich „posprzątać". Metoda `@BeforeClass` jest wywołana tylko raz przed uruchomieniem jakiegokolwiek testu i metody `@Before`. Metoda `@AfterClass` – jak łatwo się domyślić – jest realizowana po wykonaniu wszystkich metod w tej klasie (testy i metody `@After`).

Przeznaczeniem junitów jest testowanie logiki, która może nie działać. Nie ma więc sensu pisać testów weryfikujących *gettery* (odczyty) i *settery* (ustawienia). Chyba, że mają w sobie zaszytą jakąś logikę… ale wtedy trzeba by zastanowić się nad zmianą nazw tych metod.

Możliwa jest również weryfikacja wyjątków, kiedy spodziewamy się, że metoda powinna rzucić wyjątek:

```
@Test(excpected=IndexOutOfBoundsException.class)
Public void shouldIndexOutOfBoundsExceptionBeThrown()
                  {
   ArrayList empty = new ArrayList();
   empty.get(0);
}
```

W celu przetestowania metod typu protected, klasa z testami jednostkowymi musi być w tym samym pakiecie co klasa testowana.

Hamcrest

Aby nasze testy były bardziej czytelne, można wspomóc junity biblioteką Hamcrest, która jest frameworkiem ułatwiającym tworzenie obiektów porównujących (matchers). Przykłady poniżej:

```
import static org.hamcrest.Matchers.*;
Employee employee = new Employee( "Franek", "Polska"
      );
Employee anotherEmployee = new Employee( "Franek",
     "Polska" );
assertThat( employee, is( anotherEmployee ));
    //porównuje obiekty wywołując metodę equals
assertThat( "All employees are from Poland",
     employees, Matchers.<Employee>everyItem(
     Matchers.<Employee>hasProperty("country",
     equalTo("Poland"))));
    //sprawdza, czy wszyscy pracownicy są z Polski

…

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
assertThat( numbers.size(), is( equalTo( 5 )));
     //sprawdza, czy lista ma 5 elementów
assertThat( numbers, greaterThan( 4 ));
     //sprawdza, czy lista ma więcej niż 4 elementy
assertThat( numbers, notNullValue());
     //sprawdza, czy lista nie jest nullem
…
assertThat("TeKsT" equalToIgnoringCase("tEkSt"));
     //porównuje stringi, ignoruje wielkość liter.
…

assertThat(stringList, hasSize(2));
     //sprawdza, czy stringList składa się
          z 2 elementów
assertThat(stringList, contains("a1","b2"));
     //sprawdza, czy stringList zawiera
          teksty "a1" i „b2"
```

Oprócz bardzo dużej ilości metod porównawczych, największym atutem biblioteki *Hamcrest* jest możli-

wość definiowania własnych obiektów porównujących.

Mockito

W celu zwiększenia izolacji i niezależności testu jednostkowego często stosuje się tzw. duble testowe (test doubles), które tak naprawdę „udają" w działaniu obiekty zależne, lecz nimi nie są. Przykładem tego typu dubla może być obiekt udający odczyt informacji z bazy danych, potrzebnych do przetestowania logiki biznesowej. Wyróżniamy 5 typów dubli:

- *Dummy Object* (atrapa) - obiekt, który nigdy nie jest tak naprawdę używany, potrzebny jest tylko po to, by uzupełnić listę parametrów metody;
- *Fake Object* (podróbka/imitacja) - obiekt posiadający implementację, ale używający uproszczonych mechanizmów. Przykładem może tu być baza danych *InMemory* (np. *HqslDB* lub *Derby*);
- *Test Stub* (namiastka) – dostarcza prostych odpowiedzi na żądania wykonywane przez logikę testu. Stuba tworzy się poprzez rozszerzenie prawdziwego obiektu lub interfejsu oraz zaimplementowanie wszystkich jego metod zgodnie z wymaganiami scenariusza testowego;
- *Test Spy* (szpieg) - rozbudowana wersja stubów mogąca przechowywać stan mówiący o tym ile razy dana metoda stuba została wywołana;
- *Mock Object* (drwina) - obiekt, który jest instruowany w jaki sposób ma wyglądać jego interakcja z testem: czy ma zwracać konkretne wartości, czy może ma rzucić wyjątek. Następnie – w fazie weryfikacji – sprawdzane jest czy *mock* wchodził w interakcje z oczekiwanymi obiektami przez odpowiednią ilość czasu.

Istnieje bardzo duże podobieństwo między stubem a mockiem. Zasadniczo dzielą je jednak dwie różnice. Pierwsza: stub jest zorientowany na stan, natomiast mock na zachowanie (behavior). Druga: zachowanie stuba jest statyczne, czyli znane na etapie kompilacji, natomiast zachowanie mocka można modyfikować w czasie uruchamiania testu (*runtime*) i może być przez to w pełni dynamiczne.

Mocki są najlepszym dublem w kontekście zapewnienia szybkości, izolacji oraz prostoty testu. Jednym z czołowych frameworków służących do tworzenia mocków jest Mockito.

Główne cechy *Mockito*:

- Umożliwia mockowanie klas podobnie jak interfejsów;
- Adnotacje (`@Mock`);
- Jasne i klarowne opisy błędnych weryfikacji (assertów);
- Wsparcie dla weryfikacji ilości wywołań;
- Elastyczna weryfikacja wywołań metod z użyciem „domyślnych" obiektów (`anyObject()`, `anyString()`, itd.);
- Pozwala tworzyć własne argumenty porównujące (*matchers*) lub użyć np. biblioteki *Hamcrest*.

Mockito jest bardzo łatwe w użyciu. W pierwszym kroku dodajemy adnotację przed nazwą klasy:

```
@RunWith( MockitoJUnitRunner.class )
```

Aby stworzyć mocka robimy:

```
@Mock Employee employee;
```

Ktoś nie lubi adnotacji? Proszę bardzo:

```
Employee employee = mock(Employee.class);
```

Następnie definiujemy jak ma się zachować taki mock w przypadku wywołania jakiejś akcji na nim:

```
when( employee.getCountry() ).thenReturn( "Poland" );
```

Jeśli wiemy, że metoda będzie wywołana 2 razy, to podczas drugiego wywołania możemy zwrócić coś innego:

```
when( employee.getCountry() ).thenReturn(
      "Poland" ).thenReturn( "Spain" );
```

A jak wygląda sprawa wywołania metod mocków z parametrami? Można podać konkretną wartość lub parametry mieszane (wszystkie oparte na matcherach):

```
when( employee.getSkills( "IT" ) ).thenReturn(
      "Java" );
when( employee.getSkills( anyString() ) ).
      thenReturn( "Java" );
when( employee.getSkills( anyString(), eq( 3 )
       ) ).thenReturn( "Java, Junit, Mockito" );
```

W powyższym przykładzie chcemy pobrać 3 umiejętności (obiekty typu *skill*) z dowolnej dziedziny.

Mocki domyślnie zwracają wartość null, pustą kolekcję lub właściwą wartość domyślną dla obiektu np. 0 czy false:

```
verify(employee).setDepartment( "IT" );
verify(employee).setDepartment( anyString() );
verify(employee).getSkills( anyString(), eq( 3 ) );
```

W powyższym przykładzie chcemy zweryfikować czy metoda `getSkills` z dwoma parametrami została wy-

wołana z liczbą 3 jako drugi parametr. Wartość parametru pierwszego nas nie interesuje (może być dowolna).

Możemy również sprawdzić liczbę wywołań danej metody:

```
verify(employee, times(2)).addSkill ( "Java" );
      //czy wywołane dokładnie 2 razy
verify(employee, atLeast(3)).addSkill ( "Java" );
      //czy wywołane minimum 3 razy
verifyZeroInteractions(employee1, employee2 );
      //sprawdza, czy nie było użycia tych obiektów.
```

Inne podobne metody to: `atMost()`, `atLeastOnce()`. Domyślna wartość to: `times(1)`, która jest używana w przypadku nie podania żadnej innej metody.

Inną ciekawą funkcją jest sprawdzenie kolejności wywołań:

```
InOrder inOrder = inOrder(firstEmployee,
       secondEmployee);
inOrder.verify(firstEmployee).addSkill("Java");
inOrder.verify(secondEmployee).addSkill(".NET");
```

Mockito pozwala nam ustawić kaskadę zwracanych wartości dla wywołania tej samej metody:

```
when( employee.getSkills( "IT" ) )
   .thenReturn( "Java", ".NET" )
      //pierwsze wywołanie getSkills("IT")
            zwraca "Java" tekst, drugie –
            ".NET" tekst
   .thenThrow( new RuntimeException() );
      // trzecie wywołanie rzuca wyjątek
```

O tym, że szpiegostwo to poważna broń, wiemy wszyscy. Mockito umożliwia nam tworzenie szpiegów.

Poniżej został przedstawiony *Spy* (szpieg), który nie jest typowy wg definicji dubli, lecz stanowi pewnego rodzaju hybrydę między prawdziwym obiektem a mockiem. Może on wywoływać metody do czasu wywołania metody udawanej (*mocked*), jak również zapamiętywać integracje z innymi obiektami:

```
Employee spy = spy( new Employee() );
      // lub @Spy Employee spy;
when( spy.getSkills( "IT" ) ).thenReturn(
      "Java" );
      //tylko takie wywołanie metody obiektu
            spy jest zamockowane. Każde inne
            wywołanie na tym obiekcie będzie
            wywołaniem prawdziwej metody.
```

Istotnym faktem jest to, że nie można mockować metod typu *final*.

Szpieg pozwala nam podmienić prawdziwą metodę. Trochę odwrotna sytuacja jest możliwa, jeśli utworzymy mocka. Można wtedy wywołać prawdziwą metodę:

```
Employee employee = mock( Employee.class );
when( employee.setSalary() ).thenCallRealMethod();
```

PowerMock

Czasami przetestowanie jakiegoś kodu może być bardzo trudne. Do tego stopnia trudne, że trzeba sięgnąć po coś mocniejszego niż same junity z *Mockito*. *PowerMock* przybywa na ratunek! Jest to bardzo przydatny framework do testowania kodu normalnie nietestowalnego jak np. metody prywatne, metody statyczne czy konstruktory. *PowerMocka* można użyć w następujący sposób:

```
import static org.easymock.EasyMock.expect;
import static org.powermock.api.easymock.PowerMock.
      mockStatic;
import static org.powermock.api.easymock.PowerMock.
      createPartialMock;
import static org.powermock.api.easymock.PowerMock.
      replayAll;
import static org.powermock.api.easymock.PowerMock.
      verifyAll;
import org.powermock.api.easymock.annotation.Mock;
import org.powermock.core.classloader.annotations.
      PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;
@RunWith( PowerMockRunner.class )
@PrepareForTest( { EmployeeUtils.class } )
      //lista klas zawierających metody statyczne
…
createPartialMock( EmployerUtils.class,
      "hireNewEmployee", "fireEmployee" );
      //mokuje tylko te dwie wymienione metody
mockStatic( EmployeeUtils.class );
      //dajemy znać jaka klasa zawiera metody
            statyczne
expect( EmployeeUtils.someStaticMethod() ).
      andReturn( "value" );
      //zamokowana metoda statyczna
```

Przykład: Metoda, którą będziemy testować, wywołuje metodę statyczną:

```
public long generateKey()  {
   return KeyGenerator.generateNewKey ();
}
```

Klasa generatora z metodą statyczną:

```
public class KeyGenerator {
```

```
   public static long generateNewKey() {
      return System.currentTimeMillis();
   }
}
jUnit:
@Test
public void shouldGenerateKey() throws Exception {
   long key = 100;
   KeyService service = new KeyService();
   mockStatic( KeyGenerator.class );
   expect( KeyGenerator.generateNewKey() ).
            andReturn( key );
   replay( KeyGenerator.class );
   long result = service.generateKey();
   verify( KeyGenerator.class );
   assertEquals( key, result );
}
```

PowerMock stosuje się głownie do już istniejącego kodu (*legacy*), który chcemy pokryć testami. Nie powinien być nadużywany przy tworzeniu nowego oprogramowania. Jeżeli jesteśmy zmuszeni do jego użycia w nowych funkcjonalnościach, powinniśmy dwa razy zastanowić się nad tym, czy zamiast tego nie należy tak zmodyfikować logiki biznesowej, aby mogła być testowalna przez standardowe mechanizmy oferowane prze *jUnit*, *Hamcrest* i *Mockito*.

TDD (Test Driven Development – Programowanie Sterowane Testami)

TDD jest techniką tworzenia oprogramowania zaliczaną do tzw. metodyk zwinnych (*Agile*). Rozwój oprogramowania opiera się na krótkich cyklach. Technika ta polega na pisaniu testów jednostkowych zanim powstanie funkcjonalność, którą mają testować. Oznacza to, że testy w początkowej fazie nie kończą się sukcesem, z tej racji, że system nie ma jeszcze zaimplementowanych funkcji biznesowych.

Takie podejście ma wpływ na programistę, który pisząc test do nieistniejącej funkcjonalności nie myśli o tym, jak napisać junita sprawdzającego jakiś kod, tylko projektuje test pod kątem wymagań specyfikacji. Można powiedzieć, że test staje się tzw. implementacją specyfikacji przez przykład (*specifications by example*) oraz pierwszym klientem oprogramowania. Test wymusza na programiście takie zaprojektowanie i zaimplementowanie funkcjonalności, aby była łatwo testowalna. Cykl TDD wygląda następująco:

- Napisz test – test na początku musi nie przechodzić, ponieważ funkcjonalność jeszcze nie istnieje. Programista piszący test, musi bardzo dobrze rozumieć specyfikację i wymagania nowej funkcjonalności.
- Uruchom testy i sprawdź, czy nowe testy nie przechodzą – ten etap ma na celu sprawdzenie czy nowe testy nie mają błędu powodującego, że test zawsze przechodzi, co skutkuje tym, że jest on bezużyteczny. Test powinien również nie przechodzić z określonego powodu. Daje nam to pewność, że test sprawdza tą funkcjonalność, jaką chcemy.
- Zaimplementuj coś – implementujemy funkcjonalność, która powoduje, że test zacznie przechodzić. Na tym etapie kod może być „brzydki", ale ma zapewniać tylko przejście testu.
- Uruchom testy i sprawdź czy przechodzą – sprawdzamy czy funkcjonalność działa zgodnie z wymaganiami.
- Zrób refaktoring – teraz „upiększamy" kod. Usuwamy duplikacje kodu, weryfikujemy czy nazwy zmiennych i metod są zgodne z ich przeznaczeniem, sprawdzamy i upraszczamy konstrukcję kodu, kontrolujemy czy kod jest w „logicznie" odpowiednim miejscu (odpowiedni pakiet, klasa itd.). Każda zmiana powinna implikować ponowne wywołanie testów w celu potwierdzenia, że refaktoring niczego nie zepsuł.
- Wróć do punktu pierwszego – zaimplementuj kolejny test i wykonaj cały cykl od nowa dla kolejnego wymagania.

Takie podejście jest czasochłonne, nie ma się co oszukiwać. Badania wykazują, że stosowanie tej metodyki zwiększa czas potrzebny na implementację od 15% do 35%. Często jest to bariera nie do pokonania w wielu firmach i projektach, gdzie klient oczekuje gotowego rozwiązania na wczoraj i zawsze twierdzi, że jest za drogo. Rzadko który klient jest na tyle doświadczony lub bez problemu daje się przekonać, że ta inwestycja się opłaci. Główne korzyści wynikające z tego podejścia to:

- wiele błędów może być wyłapanych w bardzo wczesnej fazie implementacji i to pozwala zaoszczędzić godziny lub dni poświęcone na debugowanie (śledzenie krokowe) w poszukiwaniu głęboko zakamuflowanego *buga* (błędu). Ponadto poprawa błędu jest tym tańsza, im szybciej zostanie on wykryty. Późne wykrycie błędu na środowisku produkcyjnym może nie tylko powodować problemy związane z samą jego naprawą, ale dodatkowo wymuszać naprawy danych produkcyjnych i proces instalacji patcha (poprawionej wersji);
- kod jest czysty, przejrzysty i łatwo testowalny;
- programista skupia się na implementowaniu tylko tego, co jest ważne. Ponieważ implementuje funkcjonalność przedstawioną w teście, nie powstają zbędne metody;
- posiadanie testów dla każdej funkcjonalności;
- samodokumentujące się testy;

- wysokie pokrycie kodu testami jednostkowymi, które zabezpieczają kod przed wprowadzaniem kolejnych zmian w kodzie;
- kod jest bardziej zmodularyzowany, elastyczny i rozszerzalny. Wynika to z faktu, że programista myśli o aplikacji jak o małych fragmentach kodu, które mogą być napisane i przetestowane osobno i później zintegrowane. Prowadzi to do tworzenia mniejszych klas, bardziej wyspecjalizowanych, luźno powiązanych oraz przejrzystych interfejsów.

Reguła FIRST

Podczas tworzenia testów bardzo dobrą praktyką jest stosowanie się do założeń nakładanych przez regułę *FIRST*. Nazwa reguły pochodzi od pierwszych liter kryteriów, które ta zasada zawiera.

Fast (Szybkość) – testy jednostkowe powinny być szybkie. Od tego, ile czasu zajmuje wykonanie testów, zależy fakt jak często będziecie je wykonywać. Jak często uruchamiacie builda (proces kompilacji) z dodatkiem – *DskipTests*, bo testy trwają za długo? Podczas pisania testów należy starać się wyeliminować wszystkie czasochłonne operacje. Załóżmy, że operacja, którą chcemy przetestować pobiera dane z zewnętrznego serwisu, bazy danych, przetwarza te dane i zapisuje do pliku na dysku. Taki test w zależności od wielkości danych może trwać np. 0,3s lub więcej. Taki czas wywołania pojedynczego testu przy założeniu, że będzie ich setki czy tysiące, jest nie do zaakceptowania. Imitując dane wejściowe poprzez użycie w/w mocków oraz testowanie danych wyjściowych z użyciem assertów pozwoli nam skrócić czas wykonania do np. 0,002s.

Isolated (Izolacja) – testy powinny być niezależne od czynników zewnętrznych (inne systemy, system plików itd.) oraz innych testów. Nie zachowanie izolacji między testami może skutkować tym, że zmiana w jednym teście spowoduje, że kilka innych przestanie działać i zlokalizowanie powodu ich nie działania może zająć sporo czasu.

Repeatable (Powtarzalność) – każdorazowe wywołanie testu musi zwracać te samy wyniki. Zwiększenie izolacji wpływa na zwiększenie powtarzalności.

Self-Verifying (Samoweryfikacja) – jeżeli po wykonaniu wszystkich testów widzicie kolor zielony – co oznacza, że wszystkie testy zakończyły się zgodnie z oczekiwaniami – to macie dużą pewność, że kod jest dobrej jakości. Jeżeli choć jeden test nie przejdzie przez cały proces, musi zostać wstrzymany do czasu jego poprawienia. Dobrze napisany test sam jednoznacznie może określić czy zakończył się sukcesem czy nie. Klasycznym przykładem zakłamywania pokrycia kodu jest napisanie junita, który przechodzi przez dużą część aplikacji i nic nie sprawdza:

```
@Test
public void shouldFoundEmployeeOfTheYear() {
Employee employeeOfTheYear = service.
      foundEmployeeOfTheYear();
System.out.println(employeeOfTheYear.toString());
}
```

W powyższym przykładzie nie liczyłbym na to, że programista będzie sprawdzał co test wydrukował na konsolę. Zamiast drukować na konsolę wyniki działania testowanej metody lepiej sprawdzić rezultat assertem:

```
@Test
public void shouldFoundEmployeeOfTheYear() {
Employee employeeOfTheYear = service.
      foundEmployeeOfTheYear();
assert(..)
}
```

Testy, które nic nie testują, zakłamują wskaźniki pokrycia kodu. Menadżerowie cieszą się, że pokrycie kodu jest na poziomie 75%, podczas gdy 35% z tego nigdy nie zostało zweryfikowane. Jeśli zauważycie takie testy u Was w firmie, zlećcie ich przerobienie lub wręcz usunięcie, aby nie zakłamywać raportów pokrycia kodu.

Timely (Zrobiony w odpowiedniej chwili) – zgodnie z metodyką TDD testy powinny być pisane przed funkcjonalnością, którą testują. Pozwala to tworzyć testy, które niejako opisują specyfikację. Iteracyjna natura TDD pomaga tworzyć testowalny kod wysokiej jakości poprzez częsty refaktoring. A co z testami pisanymi w momencie, kiedy funkcjonalność jest już zaimplementowana? Ten proces również doczekał się swojej nazwy: TAD – Test After Development. W tym przypadku głównym celem programistów jest weryfikacja: czy pewne fragmenty kodu działają, czy nie. Nie można już tych testów traktować jako „specifications by example". Programiści piszący testy dla istniejącego kodu często nie mają już wpływu na to, jak ten kod wygląda i jakiej jest jakości. Często okazuje się, że oprogramowanie jest napisane w sposób utrudniający lub wręcz uniemożliwiający przetestowanie. Wtedy jedynym rozwiązaniem jest przerobienie części funkcjonalności lub wspomaganie się takimi mechanizmami jak wspomniany PowerMock, aby uniknąć zmian w implementacji. Dochodzimy więc do wniosku, że TAD jest mniej zyskowne, mniej produktywne i bardziej frustrujące niż TDD.

Pamiętajcie! Najważniejsze dla testów jednostkowych jest, aby były użyteczne i efektywne. Podejście FIRST pomoże Wam to osiągnąć.

Czytelność jUnit

Pewnie często się Wam zdarza, że przeglądacie istniejące testy jednostkowe, musicie je poprawić, lecz trudno

Wam się w nich odnaleźć z powodu nieklarownego kodu. Możemy zdecydowanie poprawić czytelność takiego testu wprowadzając standardowy szablon w oparciu o sekcje *//given //when //then*.

Sekcje te są odpowiedzialne za:

- `given` – stan początkowy systemu (np. stan bazy danych z informacjami o użytkownikach),
- `when` – zdarzenie w systemie (np. zalogowanie użytkownika do systemu),
- `then` – oczekiwane zachowanie systemu (np. przeniesienie użytkownika na stronę z jego profilem).

Testy te z reguły weryfikują jeden scenariusz będący całością większej historyjki (tzw. *user story*). Oprócz standardowej struktury mają też usystematyzowane nazwy. Z reguły zaczynają się od słowa „*should*", które sugeruje jak system powinien się zachować, a ich nazwa jest jednocześnie informacją o tym, co dany test weryfikuje. Test sprawdzający, czy po zalogowaniu przeniesiono użytkownika do jego profilu, powinien nazwać mniej więcej tak:

```
shouldRedirectUserToUserProfile.
```

Poniżej możecie zobaczyć jak to wygląda:

```
@Test
public void shouldGenerateReport() {
    //given
    when(employee.get(..)).thenReturn(…);
    //when
    employeeService.generateReport(…);
    //then
    assert(…);
}
```

Narzędzia do weryfikacji pokrycia kodu i nie tylko

Ponieważ ludzie z natury są leniwi i często popełniają błędy, powinniśmy dążyć do tego, aby jak najbardziej eliminować tzw. „czynnik ludzki" i zastępować go automatyzacją. Uruchamiając przygotowane przez nas testy na serwerach ciągłej integracji (*CI – Continous Integration*) takich jak Jenkins/Hudson, jesteśmy w stanie w sposób zautomatyzowany i na bieżąco weryfikować czy system zachowuje się poprawnie. Wraz z narzędziami do pokrycia kodu (*Emma*, *Cobertura*) i jego statycznej analizy (*Findbugs*, *PMD*, *Checkstyle*), możemy diametralnie zwiększyć jakość dostarczanego oprogramowania. Dobrą praktyką jest również zintegrowanie serwera CI z aplikacją Sonar, która agreguje w sobie wszystkie w/w mechanizmy pokrycia kodu i jego analizy statystycznej oraz dostarcza zaawansowanych raportów przydatnych podczas fazy refaktoringu kodu.

Podsumowanie

Programiści powinni tworzyć kod niezawodny, wysokiej jakości, odporny na wszelkiego typu nieprzewidziane sytuacje. Jednak nawet najlepszym programistom zdarzają się głupie błędy, każdemu może też trafić się gorszy dzień. Często po 12 godzinach pracy w pełnym skupieniu i gigantycznym stresie, z oddechem lidera na plecach i jego pytaniami: "dlaczego to nie działa? " czy "jak długo jeszcze? zgodnie z prawami Murphy'ego – błędy, niczym małe chochliki, po cichutku wkradają się w kod programisty.

Właśnie dlatego powinno się tworzyć oprogramowanie zgodnie z metodologią TDD i pisać testy jednostkowe jeszcze przed implementacją. Pomaga to lepiej zaprojektować system i wykryć wiele złośliwych bugów na wczesnym etapie.

Szeroki wachlarz bibliotek testowych pozawala wnikliwie a równocześnie w łatwy i przyjemny sposób przetestować nawet najskrytsze zakamarki rozwijanej aplikacji. W przypadku skomplikowanego, źle zaprojektowanego kodu, którego nie sposób przetestować w standardowy sposób, mamy do dyspozycji potężne działo "*PowerMock*".

Jeśli dołożymy do tego stosowanie się do reguły FIRST, otrzymujemy potężną armię testów jednostkowych - naszego sojusznika w nierównej walce programisty z tymi chochlikami w kodzie, które tylko czekają w ukryciu, żeby ujawnić się w najmniej oczekiwanym momencie.

Na wojnie tak się dzieje, że zawsze ktoś ginie. Dobrze, żebyśmy to nie byli my!

Rozwój aplikacji można przyrównać do działań wojennych. Po jednej stronie jesteśmy my, po drugiej one – bugi, które musimy zlikwidować za wszelką cenę. Skoro mamy czołgi i myśliwce, to dlaczego ich nie wykorzystać, aby szybko rozstrzygnąć walkę na naszą korzyść? Narzędzia do sprawdzania pokrycia kodu (*Emma*, *Cobertura*) wskażą nam białe pola w kodzie, które nie zostały zweryfikowane przez testy jednostkowe. Narzędzia do statycznej analizy kodu (*Findbugs*, *PMD*, *Checkstyle*) wyznaczą przypuszczalne miejsca, w których ukrywają się bugi, jak np. potencjalne wystąpienie *Nullpointera* czy *IndexOutOfBoundExceptiona*. Idąc dalej – wykorzystując terminologię wojskową – aplikację Sonar można porównać do lotniskowca. Jest to samodzielna jednostka operacyjna zdolna do wykonania każdego zadania. Sonar, integrując wszystkie wymienione wcześniej narzędzia, wykonuje zmasowany atak na kod i przeprowadza wnikliwą analizę, która jest przedstawiona w formie przyjaznego, zaawansowanego raportu.

Należy jednak pamiętać o regule tzw. złotego środka. W myśl tej reguły nie należy więc przekraczać pewnego progu pokrycia kodu (w zależności od projektu może być ustawiony na poziomie od 75% do 90%) ze względu na nieadekwatny poziom włożonego wysiłku do ostatecznie osiągniętych korzyści. Nie ma sensu stosować pokrycia

kodu na poziomie 100% – oznaczałoby to, że pokrywamy wszystkie możliwe linie, nawet proste settery i gettery, w których zasadniczo nie ma czego testować. Wykorzystując wcześniej użyte porównania militarne: nie ma sensu organizować zmasowanego ataku na nieprzyjaciela i wykorzystywać wszystkie zasoby swoich sił, bo trzeba przecież pozostawić rezerwy na obwodach.

Wiemy oczywiście, że samo stosowanie tych narzędzi nie zapewnia jeszcze sukcesu. Posiadanie genialnych generałów i nowoczesnej armii nie gwarantuje przecież zwycięstwa. Trzeba jeszcze umieć dobrze wykorzystać takie zasoby. W wielu firmach korzysta się z aplikacji Sonar i innych narzędzi, ale nikt nie weryfikuje informacji, które dostarczają. Jest to najlepsza droga do otrzymania telefonu od wściekłego klienta, w którego aplikacji nagle zaczęły dziać się rzeczy niepożądane.

Sonar, czy również wszystkie wymienione wcześniej narzędzia skonfigurowane pojedynczo z serwerem CI (Jenkins/Hudson), po każdym buildzie dostarczają informacji nt. aktualnego stanu systemu. Serwery CI mogą uniemożliwić zbudowanie się aplikacji w przypadku niepowodzenia któregoś z junitów lub pojawienia się potencjalnego błędu wykrytego przez firebuga, co może zaowocować tym, że klient jednak nie zadzwoni! Chyba, że z propozycją stworzenia nowego projektu! ;)

ADAM KOŃCA

Współtwórca i współwłaściciel marki Sagiton. Posiada 8 lat doświadczenia w programowaniu. Obecnie realizuje się jako Team Leader w projektach wykonywanych przez ArrowGroup Sp. z o.o., której jest współzałożycielem. Ukończył kierunek: Bazy danych, sieci i systemy komputerowe na Wydziale Automatyki, Elektroniki i Informatyki na Politechnice Śląskiej. Odbył kursy i szkolenia organizowane m.in. przez: Sun Microsystems, Microsoft, o czym świadczą uzyskane certyfikaty eksperckie .

Marka: www.sagiton.pl
Kontakt: adam.konca@sagiton.pl

Helion

SMASHING MAGAZINE

Thord Daniel Hedengren

PODRĘCZNIK WordPressa

SMASHING MAGAZINE

Najlepszy podręcznik o WordPressie!

Tytuł oryginału: Smashing WordPress: Beyond the Blog

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-6678-2

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/podwsm.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/podwsm
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

7 ANATOMIA WTYCZKI DO WORDPRESSA

NIE MA WĄTPLIWOŚCI, że wtyczki nie są tym samym co motywy, choć można znaleźć wiele łączących je podobieństw. Można powiedzieć, że gdy implementuje się jakąkolwiek funkcję w pliku *functions.php*, to w istocie pisze się wtyczkę.

Dzieli je jednak ogromna różnica. Motywy służą do prezentowania treści witryny przy użyciu dostępnych narzędzi. Natomiast wtyczki służą do rozszerzania funkcjonalności WordPressa o dodatkowe funkcje. Należy o tym pamiętać, ponieważ rozbudowywanie pliku *functions.php* w nieskończoność wcale nie jest najlepszym rozwiązaniem.

W tym rozdziale spojrzymy na wtyczki z nieco innej perspektywy niż do tej pory. Za pomocą wtyczki można zrobić wszystko. Ogólnie rzecz biorąc, wtyczki są metodą pozwalającą dodać do WordPressa dowolną funkcję bez żadnych ograniczeń. Porównaj to z kombinowaniem przy użyciu kilku tagów w plikach szablonowych motywu. W przypadku wtyczek kwestią nie jest, **co** można zrobić, lecz **po co** miałoby się coś robić.

RODZAJE WTYCZEK

Wyróżnia się trzy główne rodzaje wtyczek: zwykłe wtyczki, których na pewno nieraz używałeś, wtyczki do rdzenia (ang. *drop-in*), które zastępują rdzenne funkcje, oraz wtyczki obowiązkowe.

ZWYKŁE WTYCZKI

Pod pojęciem **zwykłych wtyczek** rozumiem wtyczki, do których używania jesteś przyzwyczajony. Są to wtyczki, które się pobiera z internetu oraz włącza, aby rozpocząć ich używanie. Taką zwykłą wtyczką jest np. Akismet (*http://wordpress.org/extend/plugins/akismet*). Aby taka wtyczka działała, wystarczy ją po prostu włączyć, ewentualnie skonfigurować jakieś drobne ustawienia. Zwykłe wtyczki są przechowywane w folderze *wp-content/plugins/*.

W zasadzie o tym rodzaju wtyczek wszystko już wiesz, a więc przejdźmy do omówienia następnych typów.

WTYCZKI DO RDZENIA

Wtyczki do rdzenia przesłaniają rdzenne funkcje systemu. Umieszcza się je bezpośrednio w folderze *wp-content* pod nazwą odpowiadającą plikowi, który mają zastąpić, np. *advanced-cache.php* albo *db.php*.

Poniżej podano dostępne wtyczki tego typu. Pamiętaj, że jeśli zdecydujesz się na ich użycie, musisz rzeczywiście napisać własny kod zastępujący standardowe skrypty. Jeśli tego nie zrobisz, prawie na pewno będziesz mieć problemy.

- *advanced-cache.php* — własne skrypty zaawansowanego buforowania;
- *db.php* — własna klasa bazy danych;
- *db-error.php* — własne powiadomienia o błędach bazy danych;
- *install.php* — własne skrypty instalacyjne;
- *maintenance.php* — własne wiadomości dotyczące spraw utrzymania serwisu;
- *object-cache.php* — zewnętrzne buforowanie;
- *sunrise.php* — skrypty, które mają zostać wykonane przed załadowaniem sieci witryn;
- *blog-deleted.php* — usuwanie blogów w sieci witryn;
- *blog-inactive.php* — wiadomości o nieaktywnych blogach w sieci witryn;
- *blog-suspended.php* — wiadomości o zawieszonych blogach w sieci witryn.

Wtyczek tych należy używać bardzo ostrożnie, chociaż dają one naprawdę bardzo duże możliwości. Możliwe, że niektórych z nich zdarzało Ci się używać, zapewne jako części innych wtyczek. W szczególności dotyczy to wtyczki *advanced-cache.php*. Zabawa z niektórymi jest mniej ryzykowna, np. wtyczka *maintenance.php* umożliwia wyświetlenie własnej wiadomości na temat uaktualniania instalacji systemu.

WTYCZKI OBOWIĄZKOWE

Wtyczki obowiązkowe różnią się od zwykłych wtyczek. Ich pliki przechowywane są w folderze *wp-content/mu-plugins/* i nie da się ich wyłączyć w panelu administracyjnym WordPressa. Jedyny sposób na ich dezaktywację to usunięcie ich z folderu na serwerze. Wtyczki obowiązkowe nie muszą zawierać specjalnego nagłówka. Bez niego również zostaną wykonane.

W folderze *wp-content/mu-plugins/* można umieścić dowolną wtyczkę, ale z niektórymi wtyczkami mogą być problemy, zwłaszcza w sieciach witryn. Dlatego postępuj ostrożnie. Najlepiej wtyczek używać w sposób zgodny z ich przeznaczeniem.

Wtyczki obowiązkowe są najlepszym rozwiązaniem, gdy chcemy mieć pewność, że jakieś funkcje nie zostaną przypadkowo wyłączone.

TWORZENIE WTYCZEK DO UŻYTKU W SIECIACH WITRYN

Od WordPressa 3.0 wersja systemu dla wielu użytkowników, zwana **WordPress MU**, stała się częścią podstawowej wersji. Teraz jest to tzw. funkcja tworzenia wielu witryn (ang. *multisite*), dzięki której można tworzyć sieci witryn. Większość wtyczek i motywów dobrze działa w tych sieciach. Problemy mogą się pojawić jedynie wtedy, gdy wtyczka będzie dodawać tabele do bazy danych albo modyfikować istniejące tabele rdzenia. Funkcję sieci witryn włącza się poprzez wpisanie kilku wierszy kodu w pliku *wp-config.php*. Na początek należy dodać poniższy wiersz kodu nad komentarzem `/* To wszystko, zakończ edycję w tym miejscu! Miłego blogowania! */`:

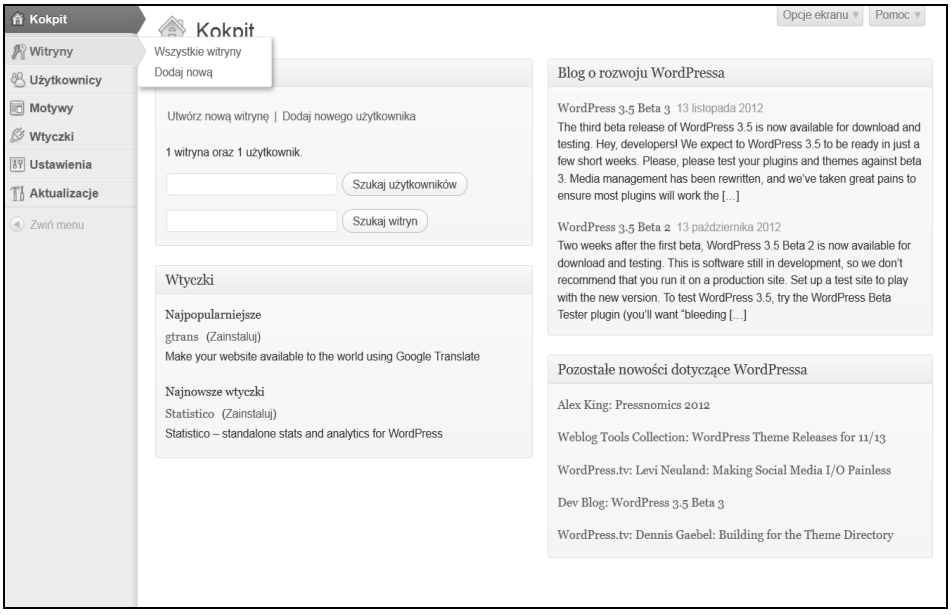

```
define( 'WP_ALLOW_MULTISITE', true );
```

Dodanie tego wiersza kodu do pliku *wp-config.php* spowoduje pojawienie się nowego odnośnika w panelu administracyjnym (rysunek 7.1). Gdy go klikniesz, zostaną wyświetlone proste instrukcje, według których należy postępować, aby uruchomić sieć witryn. Bardziej szczegółowe informacje na ten temat znajdują się na stronie *http://codex.wordpress.org/Create_A_Network*.

Sieć witryn pozwala uruchomić coś w rodzaju nadrzędnego panelu administracyjnego, w którym administrator może zarządzać wieloma witrynami utworzonymi w jego sieci. W serwisie WordPress.com każdy może założyć własnego bloga, ale sieć witryn niekoniecznie musi działać właśnie w ten sposób. Równie dobrze można ją wykorzystać do uruchomienia wielu witryn, nie pozwalając użytkownikom tworzyć własnych blogów.

Tworzenie wtyczek przeznaczonych do użytku w sieciach witryn niewiele różni się od tworzenia zwykłych wtyczek. Największa różnica dotyczy bazy danych i w mniejszym stopniu struktury katalogów.

Jeśli chodzi o strukturę katalogów, to prawie wszystko jest tak samo jak w standardowym WordPressie. Różnica polega na dodaniu katalogu *blogs.dir* do folderu *wp-content*, w którym przechowywane są wszystkie dane utworzonych witryn, takie jak obrazy i inne pliki. Z folderu



Rysunek 7.1. Panel administracji siecią witryn

tego nie będziesz często korzystać, ponieważ motywy i wtyczki należą do folderu *wp-content*, tak jak zawsze.

Jeśli chcesz, aby wybrane wtyczki były włączone w całej sieci, możesz je aktywować w panelu administracyjnym sieci. Możesz też skorzystać z wtyczek obowiązkowych, czyli po prostu umieścić wybrane wtyczki w folderze *wp-content/mu-plugins/*, chociaż to może przysporzyć Ci problemów, jeśli któraś z tych wtyczek nie będzie przystosowana do takiego sposobu użycia. Zwykle najlepszym rozwiązaniem jest aktywowanie wtyczki dla całej sieci.

Sam proces powstawania wtyczki wygląda tak samo, jak dla normalnego WordPressa. Trzeba tylko bardziej uważać podczas tworzenia nowych tabel w bazie danych oraz przy pobieraniu treści z tabel rdzenia. W większości przypadków czynności te nie sprawiają kłopotów, ale baza danych dla sieci witryn ma trochę inną strukturę, więc należy uważać.

Kolejną kwestią, jaką należy wziąć pod uwagę podczas pisania wtyczki dla sieci witryn, jest jej przewidywany sposób użycia. Sieć witryn może być prowadzona na wiele różnych sposobów, może być otwarta lub zamknięta, użytkownicy mogą mieć możliwość używania wtyczek lub nie itd. Trzeba to wszystko wziąć pod uwagę, gdy tworzy się wtyczkę.

WTYCZKI DLA CAŁEJ SIECI

Wtyczki można też włączać dla całej sieci witryn w panelu administracyjnym sieci. W ten sposób można włączać wtyczki znajdujące się w folderze *wp-content/plugins/* dla wszystkich witryn. To jest oczywiście bardzo wygodne rozwiązanie i należy z niego korzystać, zamiast używać wtyczek obowiązkowych. Skoro WordPress może być automatycznie aktualizowany, to im więcej ustawień można zdefiniować kliknięciem w panelu administracyjnym, tym lepiej.

PODSTAWY BUDOWY WTYCZEK

Podstawy budowy wtyczek są podobne do podstaw budowy motywów:

- Główny plik wtyczki musi być w formacie PHP i mieć niepowtarzalną nazwę albo znajdować się w folderze o niepowtarzalnej nazwie, jeśli wtyczka składa się z wielu plików.
- Główny plik PHP wtyczki musi mieć specjalny identyfikujący go nagłówek, podobny do pliku *style.css* w motywach.

Wtyczka może składać się z pliku głównego i wielu plików zawierających różne funkcje, podobnie jak motyw może składać się z wielu plików szablonowych i pliku *style.css*.

Zanim przejdziemy do szczegółowego omawiania podstaw budowy wtyczek, muszę Cię ostrzec, że tworzenie wtyczek jest znacznie bardziej wymagającym zadaniem niż tworzenie motywów. Do tego potrzebna jest solidna znajomość języka PHP i jeśli masz w tej kategorii braki, lepiej trochę się podszkol, zanim zaczniesz pisać jakąś poważniejszą wtyczkę.

Bardzo ważną kwestią jest wybór nazwy pliku lub folderu wtyczki, ponieważ wtyczka ta będzie zapisana w folderze *wp-content/plugins/* z innymi wtyczkami, a więc może dojść do konfliktów nazw. Nazwij zatem swoją wtyczkę w taki sposób, aby ktoś, kto będzie jej szukał na FTP, bez problemu mógł ją znaleźć, znając jej nazwę tylko z panelu administracyjnego WordPressa.

Blok identyfikacyjny wtyczki wygląda znajomo. Poniżej znajduje się przykład takiego bloku:

```
<?php
/*
Plugin Name: Moja wtyczka
Plugin URI: http://url-mojej-wtyczki.com/
Description: Opis mojej wtyczki.
Version: 1.0
Author: Thord Daniel Hedengren
Author URI: http://tdh.me/
*/
?>
```

Tak naprawdę obowiązkowo nagłówek musi zawierać tylko nazwę wtyczki, która w powyższym przykładzie została zapisana w pierwszym wierszu komentarza. Jednak pozostałe informacje również należy dodawać, aby użytkownik mógł przeczytać, co to za wtyczka, skąd pobierać aktualizacje, jaki jest numer wersji itd.

Powinno się także dodać informację o licencji. W dokumentacji WordPressa zalecane jest używanie poniższego standardowego wyciągu z tekstu licencji GPL:

```
<?php
/* Copyright ROK AUTOR (e-mail : E-MAIL AUTORA WTYCZKI)

This program is free software; you can redistribute it and/or modify it under the terms of the GNU
General Public License as published by the Free Software Foundation; either version 2 of the License,
or (at your option) any later version.
```

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
/
?>

Oczywiście napisy `ROK`, `AUTOR` i `E-MAIL AUTORA WTYCZKI` należy zastąpić prawdziwymi informacjami. Można także dołączyć cały tekst licencji GPL w pliku o nazwie *license.txt*. Tekst ten jest dostępny pod adresem *www.gnu.org/copyleft/gpl.html*.

To wystarczy, aby WordPress znalazł i poprawnie zidentyfikował wtyczkę. Jeśli umieścisz ją w folderze *wp-content/plugins/*, zostanie wyświetlona na liście wtyczek sekcji *Wtyczki* w panelu administracyjnym systemu. Można ją tam włączyć, aby była dostępna w motywie i akcjach WordPressa.

Od tego momentu zaczyna się prawdziwa zabawa, ponieważ teraz musisz zastanowić się, co wtyczka ma robić i jak sprawić, żeby to robiła.

Niezależnie od tego, czy planowana wtyczka ma zmienić sposób działania WordPressa, czy tylko dodać do niego jakąś funkcję, przed rozpoczęciem pracy powinieneś przejrzeć poniższą listę punktów do sprawdzenia. Dzięki temu możesz zaoszczędzić sobie sporo czasu i problemów.

- Czy taka wtyczka już istnieje? Jeśli tak, zastanów się, czy nie lepiej jej użyć, zamiast tworzyć nową o prawie identycznym działaniu.
- Upewnij się, że nazwa Twojej wtyczki nie jest już zajęta. Sprawdź nie tylko w zasobach WordPress.org, ale również w Google. Dobrym sposobem na utworzenie niepowtarzalnej nazwy jest wstawienie na początku nazwy firmy, np. `acme_nazwawtyczki`.
- Nazwy wszystkich funkcji konsekwentnie zaczynaj od jakiegoś przedrostka. W ten sposób unikniesz konfliktów nazw z innymi funkcjami. To ważne: przedrostki stosuj przed wszystkimi nazwami!
- Czy chcesz umożliwić przetłumaczenie wtyczki na różne języki? Powinieneś. Przygotowywanie wtyczki do internacjonalizacji wygląda tak samo jak w przypadku motywów, a więc jest bardzo łatwe.
- Czy wtyczka będzie tworzyła widżety? Jeśli tak, to jakie ustawienia powinna udostępniać?
- Czy potrzebna będzie strona ustawień w panelu administracyjnym? Staraj się, aby ustawień było jak najmniej, ponieważ użytkownicy wolą proste rozwiązania.
- Na jakiej licencji będzie udostępniana wtyczka? Pamiętaj, że musi być to licencja zgodna z GPL, jeśli chcesz umieścić wtyczkę w zbiorach WordPress.org.
- Na koniec sprawdź, czy nagłówek jest aktualny, czy numer wersji jest poprawny, czy wszystkie łącza działają, czy w paczce są wszystkie pliki oraz czy nie ma błędów w tekście.

METODY INKORPOROWANIA WTYCZEK

Do budowy wtyczek o wiele częściej używa się zwykłego kodu PHP niż przy budowie motywów. Podczas gdy tagów szablonowych i warunkowych także można używać, to jednak w zdecydowanej większości przypadków będziesz pisać własne funkcje. Oczywiście dużo zależy też od rodzaju tworzonej wtyczki, jednak ogólnie rzecz biorąc, kod wtyczki jest w większości dziełem jej autora, a nie autorów WordPressa.

Działanie wtyczek zwykle opiera się na własnych funkcjach, które programista podłącza do haków i filtrów WordPressa. Umieszczając funkcję w odpowiednim miejscu, np. podłączając do `wp_head` lub komentarzy, możesz ją uruchomić w odpowiednim czasie.

W kolejnych trzech podrozdziałach znajduje się opis trzech sposobów pisania i używania wtyczek. Zaczniemy od jak zawsze ważnych haków.

UŻYWANIE HAKÓW

Pamiętasz opis haków z poprzedniego rozdziału? **Haki akcji** są wyzwalane przez określone zdarzenia podczas działania WordPressa, np. dla publikacji wpisu jest hak `publish_post`. Drugi rodzaj haków to **haki filtrów**. Są to funkcje, przez które WordPress przekazuje dane, a więc można ich używać do obróbki danych. Do przydatnych haków filtrów zaliczają się np. `the_excerpt` i `the_title`. Trzeba je odróżniać od tagów szablonowych.

Haki są przydatne także przy budowie wtyczek, nie tylko motywów. Dzięki nim można podłączyć wtyczki do odpowiednich części WordPressa, np. `wp_head` albo `wp_footer`. Dobrym tego przykładem jest dodanie funkcji w pliku *functions.php*, a następnie podpięcie jej do haka `wp_footer` za pomocą funkcji `add_action()`, opisane w rozdziale 6.

Podczas budowy wtyczek często pisze się funkcje, które następnie podłącza się do wybranych haków za pomocą funkcji `add_action()`:

```
add_action ( $nazwa_haka, $nazwa_funkcji, $priorytet, $parametry );
```

W miejsce `nazwa_haka` należy wpisać nazwę haka, do którego chcemy dodać naszą akcję. Napotykając ten hak podczas przetwarzania kodu, WordPress sprawdza, czy nie ma dla niego zarejestrowanych żadnych funkcji. Jeśli są, wykonuje je. Funkcja, która zostanie wykonana, jest zdefiniowana w miejscu `nazwa_funkcji`. Na przykład w poniższym fragmencie kodu zostanie wywołana funkcja `smashingshortcode` podczas wykonywania funkcji `wp_head()`:

```
add_action ( 'wp_head', 'smashingshortcode' );
```

Argumenty `$priority` i `$parameters` są nieobowiązkowe. Argument `$priority` jest liczbą całkowitą określającą priorytet (domyślnie ma wartość 10), według którego akcje są sortowane podczas dodawania do haka. Im mniejsza liczba, tym wcześniej dana funkcja zostanie wykonana. Jeśli więc chcesz, aby jedna funkcja została wykonana przed inną funkcją, za pomocą tego atrybutu możesz to ustawić. Natomiast argument `$parameters` określa liczbę argumentów

przyjmowanych przez naszą funkcję (domyślna wartość to 1). Jeśli chcesz, aby funkcja przyjmowała więcej niż jeden argument, możesz wpisać w tym miejscu dowolną liczbę całkowitą.

Argumenty priorytetu i liczby argumentów nie są często używane, ale w pewnych przypadkach są naprawdę bardzo przydatne. Ponieważ są opcjonalne, to jeśli się ich nie potrzebuje, można je po prostu opuścić. Jak już jednak napisałem, są sytuacje, w których argument priorytetu jest bardzo przydatny, ponieważ umożliwia uniknięcie konfliktów między wtyczkami. Jeśli masz taki problem, możesz ustawić priorytet swojej wtyczki tak, aby była ładowana jako pierwsza lub ostatnia.

Filtry działają mniej więcej tak samo, tylko zamiast `add_action()` używa się funkcji `add_filter()`. Parametry są takie same i tak samo się je przekazuje. Jedyna różnica polega na tym, że z funkcją `add_filter()` nie używa się haków akcji, tylko filtrów.

Wiemy już, że dodawanie akcji do haków każdego rodzaju jest łatwe, ale co z ich **usuwaniem**? Czasami nie chcemy, aby jakiś hak był uruchamiany, więc musimy go usunąć. Dla haków akcji istnieje funkcja `remove_action()`, a dla haków filtrów — `remove_filter()`. Składnia tych funkcji jest prosta:

```
remove_action( $hook_name, $function_name )
remove_filter( $hook_name, $function_name )
```


Funkcje te służą nie tylko do usuwania własnych funkcji, ale również funkcji samego systemu, a więc za ich pomocą można usunąć praktycznie każdy filtr i każdą akcję, od pingowania po usuwanie załączników. Dlatego niektóre wtyczki mogą tylko usuwać funkcje WordPressa, a nie rozszerzać jego funkcjonalność.

TWORZENIE WŁASNYCH TAGÓW SZABLONOWYCH

Innym sposobem na uzyskanie dostępu do funkcjonalności wtyczki jest utworzenie własnych tagów szablonowych, jak `bloginfo()` czy `the_title()`. Nie jest to wcale trudne. Wystarczy po prostu utworzyć funkcję we wtyczce (lub w pliku *functions.php*), a następnie ją wywoływać:

```
<?php nazwa_funkcji(); ?>
```

Proste, prawda? Może proste, ale to nie oznacza, że jest to najlepszy sposób dodawania funkcjonalności wtyczki do systemu. W tej metodzie nie musisz tworzyć żadnych haków, funkcja zostanie wykonana w chwili załadowania tagu szablonowego wtyczki, a więc można ją umieścić w plikach szablonowych motywu, gdzie się chce. Jest to szczególnie przydatne, gdy nie ma takiego haka, jaki jest w danej chwili potrzebny. Potem można utworzyć tag szablonowy. Oczywiście lepiej jest jednak używać istniejących haków, kiedy to tylko możliwe.

Zanim zdecydujesz się na takie rozwiązanie, powinieneś zastanowić się nad kwestią użyteczności. Nie każdy lubi i potrafi modyfikować pliki szablonowe, dlatego zwłaszcza jeśli planujesz przekazać wtyczkę do ogólnego użytku albo klientowi, zmuszanie użytkowników do grzebania w szablonach nie będzie najlepszym pomysłem. Jeśli wtyczki używać będziesz

tylko Ty, nie ma to żadnego znaczenia. Jeśli jednak inni użytkownicy będą samodzielnie wybierać miejsce do umieszczenia wtyczki albo będą zmieniać jej parametry, to powinieneś poszukać innego rozwiązania.

Jednak w niektórych sytuacjach samo dodanie tagu szablonowego nie wystarczy i trzeba nadpisać część funkcjonalności systemu. Wówczas należy użyć specjalnych funkcji nadpisujących (ang. *pluggable functions*).

FUNKCJE NADPISUJĄCE

Czasami trzeba nadpisać wybrane części rdzenia WordPressa, aby np. zastąpić je własnymi rozwiązaniami albo żeby po prostu się ich pozbyć, ponieważ chcemy używać WordPressa w niestandardowy sposób. Może nie chcesz, aby w panelu administracyjnym działały lokalizacje (wówczas pozbądź się funkcji `load_textdomain()`), albo chcesz zmienić stopkę panelu administracyjnego na własną. Tych rzeczy nie da się zrobić, usuwając tylko wybrany hak. Z tego typu problemami trzeba zwrócić się do pliku *pluggable.php* znajdującego się w folderze *wp-includes*. Oczywiście nie będziemy go modyfikować, bo mielibyśmy z nim same kłopoty przy każdej aktualizacji WordPressa. Zamiast tego napiszemy wtyczkę, która będzie go przesłaniać. Miej świadomość, że jest to niebezpieczna praca. Przede wszystkim każdą funkcję można nadpisać tylko raz, a więc jeśli dwie wtyczki nadpiszą tę samą funkcję w pliku *pluggable.php*, witryna w najlepszym wypadku będzie źle działać, a w najgorszym przestanie działać w ogóle. To oznacza, że nie można zainstalować dwóch wtyczek przesłaniających tę samą funkcję w pliku *pluggable.php*, co jest poważną wadą systemu. Z tego powodu funkcji nadpisujących najlepiej jest używać wyłącznie w witrynach, nad którymi ma się pełną kontrolę.

Aby zapobiec wyświetlaniu niepotrzebnych powiadomień o błędach, dodatkowo można kod wtyczki umieścić w instrukcji warunkowej sprawdzającej, czy dana funkcja istnieje:

```
<?php if ( ! function_exists( 'function_name' ) ); ?>
```

Oczywiście funkcje nadpisujące są czasami przydatne. Aktualna lista funkcji, które można przesłaniać, znajduje się w dokumentacji na stronie *http://codex.wordpress.org/Pluggable_Functions*.

Pamiętaj, że samo poprawne napisanie wtyczki, o czym jest mowa dalej, nie wystarczy do zapewnienia jej poprawnego działania, gdy wtyczka ta usuwa wybrane części rdzenia WordPressa. Nie dziw się, jeśli coś przestanie działać albo pojawią się jakieś konflikty, których nigdy byś się nie spodziewał. Bądź co bądź funkcje WordPressa, które próbujesz obejść, po coś przecież są.

WŁASNE TAKSONOMIE I TYPY WPISÓW

Własne taksonomie i typy wpisów to bardzo przydatne narzędzia, zwłaszcza gdy planuje się używanie WordPressa do prowadzenia czegoś więcej niż prosty blog. Jeśli nie wiesz, do czego mogą Ci się one przydać, ciekawe przykłady ich zastosowań znajdziesz w rozdziale 14.

Dla przypomnienia: własne taksonomie są dodatkowym sposobem organizacji treści. Domyślne tagi i kategorie są przykładami taksonomii odpowiednio niehierarchicznej i hierarchicznej, natomiast konkretne tagi i kategorie to terminy tych taksonomii. Natomiast własne typy wpisów są dodatkowym rodzajem publikacji, podobnie jak wpisy i strony, które też są rodzajem wpisów.

POWODY, ABY UŻYĆ WTYCZKI

Powodem, dla którego do tworzenia własnych taksonomii i typów wpisów najlepiej używać wtyczek, jest przenośność. Jeśli kod umieścisz w pliku *functions.php* (co jest możliwe), to po przenosinach do innego motywu zostanie on utracony, chociaż oczywiście można go skopiować do tego nowego motywu. We wtyczkach powinno się implementować raczej funkcje dotyczące treści niż projektu, dzięki czemu można ich używać w różnych motywach. Wystarczy włączyć wtyczkę i gotowe.

Więcej na temat przenośności danych piszę w rozdziale 8.

TWORZENIE WŁASNEJ TAKSONOMII

Aby utworzyć własną taksonomię, należy napisać funkcję zawierającą wywołanie funkcji `register_taxonomy()` i związać ją z hakiem `init`. Ustawienia funkcji `register_taxonomy()` są proste i dotyczą sposobu prezentacji taksonomii w panelu administracyjnym oraz tego, czy taksonomia powinna mieć własny bezpośredni odnośnik (argument `rewrite`), czy ma mieć strukturę hierarchiczną itd.

Poniżej znajduje się przykład tworzenia taksonomii *Moje tagi*, która jest niehierarchiczna, a więc bardzo podobna do domyślnej taksonomii tagów (rysunek 7.2):

```
// Powiązanie funkcji z hakiem init
add_action( 'init', 'smashing_tax', 0 );

// Funkcja taksonomii
function smashing_tax() {

    // Rejestracja taksonomii
    register_taxonomy( 'smashing_taxonomy', 'post',
        array(
            'hierarchical' => false,
            'labels' => array(
                'name' => 'Moje tagi',
                'singular_name' => 'Moje tagi',
                'search_items' => 'Przeszukuj Moje tagi',
                'popular_items' => 'Popularne Moje tagi',
                ↪'add_new_item' => 'Dodaj nowy Mój tag'
            ),
            'query_var' => true,
            'rewrite' => true
        )
    );

}
```

Rysunek 7.2. Taksonomia Moje tagi

Niezbyt skomplikowane, prawda? Kod ten możesz zapisać w pliku *functions.php*, ale zapewne Twoja nowa taksonomia nie jest związana z konkretnym motywem, tylko z treścią, a więc lepiej byłoby utworzyć wtyczkę. W tym celu wystarczy przenieść ten kod do nowego pliku PHP zawierającego na początku odpowiedni nagłówek. Poniżej znajduje się treść takiego pliku:

```
<?php
/*
Plugin Name: Moje tagi
Plugin URI: http://tdh.me/wordpress/moje-tagi/
Description: Dodaje taksonomię Moje tagi.
Version: 1.0
Author: Thord Daniel Hedengren
Author URI: http://tdh.me/
*/
// Powiązanie funkcji z hakiem init
add_action( 'init', 'smashing_tax', 0 );

// Funkcja taksonomii
function smashing_tax() {

    // Rejestracja taksonomii
    register_taxonomy( 'smashing_taxonomy', 'post',
        array(
            'hierarchical' => false,
            'labels' => array(
                'name' => 'Moje tagi',
                'singular_name' => 'Moje tagi',
                'search_items' => 'Przeszukuj Moje tagi',
                'popular_items' => 'Popularne Moje tagi',
                 ↪_new_item' => 'Dodaj nowy Mój tag'
            ),
            'query_var' => true,
            'rewrite' => true
        )
    );

}
```

TWORZENIE WŁASNEGO TYPU WPISÓW

Własne typy wpisów również tworzy się w prosty sposób. Mimo że je również można tworzyć we wtyczkach (istnieją nawet wtyczki dodające interfejs pozwalający tworzyć własne typy wpisów), tutaj skoncentruję się na robieniu tego bezpośrednio w motywie. Kluczowe znaczenie w tym przypadku ma funkcja `register_post_type()`:

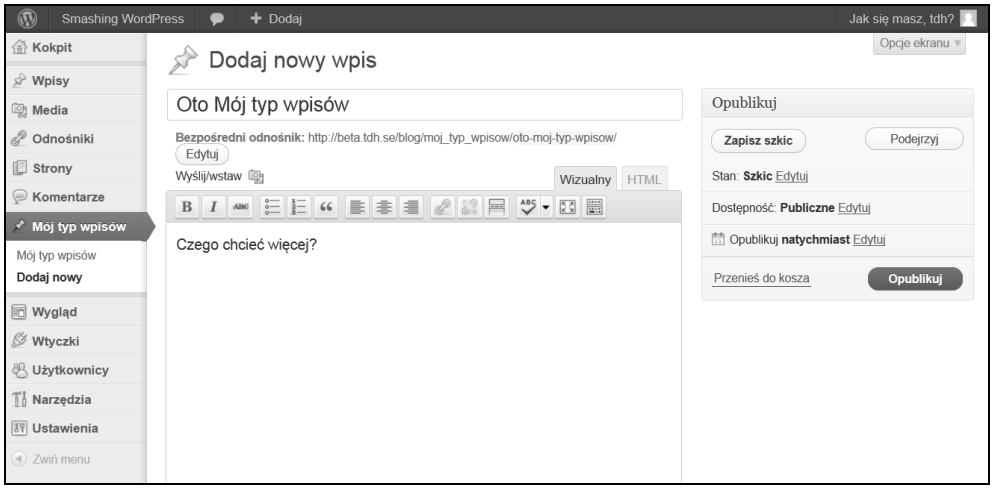
```
// Rejestracja nowego typu wpisów
register_post_type( 'moj_typ_wpisow',
    array(
         'labels' => array(
             'name' => 'Mój typ wpisów'
        ),
         'singular_label' => 'Mój typ wpisów',
          'public' => true,
          'show_ui' => true,
          'capability_type' => 'post',
          'has_archive' => true,
          'hierarchical' => false,
          'show_in_menu' => true,
          'supports' => array( 'title', 'editor', 'author', 'revisions', 'comments'
)
    )
);
```


To spowoduje zarejestrowanie nowego typu wpisów o nazwie `moj_typ_wpisow` określonej w pierwszym parametrze. Drugi parametr to tablica zawierająca ustawienia wyglądu i właściwości tego typu wpisów, np. treść etykiet w panelu administracyjnym, czy ten typ jest publiczny, czy ma on być wyświetlany jako opcja w menu i które role użytkowników mogą z niego korzystać. Większości tych ustawień nie trzeba objaśniać, ale warto zwrócić uwagę na tablicę `supports` znajdującą się w tablicy ustawień. W niej definiuje się, co dany typ wpisów obsługuje, tu: tytuł, edytora tekstu, możliwość wyboru autora, wersje wpisu oraz komentarze. Żadnych taksonomii, wypisów, własnych pól — tylko to, co zostało wpisane.

Aby nowy typ wpisów pojawił się w panelu administracyjnym, wystarczy powyższy kod wkleić do pliku *functions.php* (rysunek 7.3). Prawdopodobnie będzie trzeba ponownie wygenerować wszystkie bezpośrednie odnośniki, aby adresy wpisów nowego typu zaczęły działać.

Treść własnych typów wpisów nie jest uwzględniana przez standardową pętlę. Aby je wyświetlić, trzeba tę pętlę zmodyfikować za pomocą funkcji `query_posts()` lub stosując jakąś inną metodę — więcej informacji o pętli znajduje się w rozdziale 3. Ale jeśli masz nowy typ wpisów, możesz bez przeszkód umieścić w menu łącze do strony ich archiwum.

Istnieje wiele opcji tworzenia wpisów. Ich kompletną listę można znaleźć w dokumentacji WordPressa na stronie *http://codex.wordpress.org/Function_Reference/register_post_type*. Własnych typów wpisów będziemy używać w projektach opisanych w rozdziale 14., a więc jeśli chcesz dowiedzieć się więcej na temat ich możliwości, zajrzyj do tego rozdziału.



Rysunek 7.3. Nowy typ wpisów

Jak kod tworzący własny typ wpisów wyglądałby we wtyczce? Bardzo podobnie do własnej taksonomii. Oto wtyczka Smashing Post Type w całej okazałości:

```
<?php
/*
Plugin Name: Smashing Post Type
Plugin URI: http://tdh.me/wordpress/smashing-post-type/
Description: Adding the Smashing Post Type.
Version: 1.0
Author: Thord Daniel Hedengren
Author URI: http://tdh.me/
*/

// Dodanie do haka init
add_action( 'init', 'moj_typ_wpisow' );

// Dodanie własnych typów wpisów
function smashing_post_types() {

    // Rejestracja Mojego typu wpisów
    register_post_type( 'moj_typ_wpisow',
        array(
            'labels' => array(
                'name' => 'Mój typ wpisów',
                'menu_name' => 'Moje wpisy'
            ),
            'singular_label' => 'Mój typ wpisów',
            'public' => true,
            'show_ui' => true,
            'menu_position' => 5,
            'capability_type' => 'post',
            'has_archive' => true,
            'hierarchical' => false,
            'show_in_menu' => true,
```

```
            'supports' => array( 'title', 'editor', 'author', 'revisions', 'comments' )
        )
    );

}

?>
```

CO POWINNA MIEĆ KAŻDA WTYCZKA

Tak naprawdę jedyną obowiązkową rzeczą, jaką musi mieć każda wtyczka, jest plik PHP zawierający nagłówek identyfikacyjny oraz kod potrzebny do wykonania zadania. W rzeczywistości jednak powinno się troszkę więcej popracować. Przecież wtyczki może używać jeszcze ktoś inny, a więc powinna ona być jak najłatwiejsza w użyciu i jak najbardziej dostępna. A to oznacza, że należy dołożyć wszelkich starań, aby wtopić wtyczkę w panel administracyjny WordPressa.

To samo dotyczy wszystkich elementów widocznych dla użytkownika witryny. Niektóre wtyczki dodają elementy wizualne i jeśli wtyczka ma być udostępniana publicznie, elementy te powinny nadawać się do wyświetlenia w jak największej ilości motywów. Oczywiście kwestii tej nie ma w przypadku wtyczek pisanych tylko dla konkretnych projektów. Podobnie jest z lokalizacją. Jeśli nie planujesz dodawać obsługi innych języków, nie ma sensu przygotowywać wtyczki do lokalizacji.

Moim zdaniem każda wtyczka powinna mieć dołączoną licencję i instrukcję obsługi. Nie mam nic przeciwko plikom *readme.txt*, ale wielu użytkowników nie otwiera ich, bo im się nie chce albo są zbyt niecierpliwi. Dlatego dobrze jest, jeśli podstawowe instrukcje znajdują się w samej wtyczce. Można też dodać informacje na karcie Pomoc WordPressa za pomocą funkcji `add_help_tab()`. Więcej informacji na ten temat znajduje się na stronie *http://codex.wordpress.org/Function_Reference/add_help_tab*.

USTAWIENIA WTYCZEK

Czasami trzeba zapisać jakieś informacje w bazie danych. Jeśli chodzi o bazę danych, to możesz z nią robić wszystko, co można zrobić przy użyciu skryptów PHP, a więc dodawać tabele itd. Nie będę się o tym rozpisywał.

Pokażę Ci natomiast, jak używać API ustawień (więcej informacji znajdziesz na stronie *http://codex.wordpress.org/Settings_API*), aby zlecić zapisywanie danych ustawień WordPressowi. To nie tylko pozwala zaoszczędzić mnóstwo czasu, ale również jest bezpieczne, ponieważ w WordPressie stosowane są różne zabezpieczenia, takie jak np. nonce (ang. *number used once* — liczba użyta tylko raz).

W celach testowych utworzymy prostą stronę ustawień, którą dodamy do menu *Ustawienia* na lewym pasku bocznym panelu administracyjnego. Strona ta będzie zawierała pole tekstowe i pole wyboru.

Najpierw musimy utworzyć wtyczkę. Wystarczy nam zwykły plik PHP o nazwie *smashing-settings.php* zaczynający się od poniższego nagłówka:

```
<?php
/*
Plugin Name: Ustawienia Smashing
Plugin URI: http://tdh.me/wordpress/ustawienia-settings/
Description: Prosta wtyczka ustawień.
Author: Thord Daniel Hedengren
Author URI: http://tdh.me/
*/
```

Teraz dodamy stronę ustawień do menu *Ustawienia*. W tym celu podłączymy funkcję do haka `admin_menu`. W tym przykładzie użyjemy funkcji `add_options_page()`, ale są też inne funkcje do wyboru:

```
// Dodaje funkcję strony ustawień do menu
add_action( 'admin_menu', 'smashings_settingsdemo_add_page' );

// Dodanie do menu
function smashings_settingsdemo_add_page() {
    add_options_page( 'Przykład ustawień Smashing',
         'Ustawienia Smashing',
         'manage_options',
         'smashings_settingsdemo',
         'smashings_settingsdemo_do_page' );
}
```

Za pomocą funkcji `add_options_page()` utworzyliśmy stronę o nazwie *Przykład ustawień Smashing*, która w menu po lewej stronie jest skrócona do *Ustawienia Smashing*. Strona ta jest dostępna tylko dla użytkowników mających rolę `manage_options`. Identyfikator tej strony to `smashings_settingsdemo`. Ponadto treść samej strony jest tworzona przez funkcję `smashings_settingsdemo_do_page()`, którą musimy jeszcze napisać:

```
//Rzeczywiste dodanie strony z ustawieniami
function smashings_settingsdemo_do_page() {
// Opuszczamy PHP na moment ?>

    <h2>Ustawienia Smashing</h2>
    <p>To jest nasza strona ustawień.</p>
    <form action="options.php" method="post">
    <?php settings_fields( 'smashings_settingsdemo' ); ?>
    <?php do_settings_sections( 'smashings_settingsdemo' ); ?>
    <?php submit_button(); ?>
    </form>

<?php
} // Powrót do PHP
```

Jest to cały kod HTML strony ustawień — na razie jest go niewiele, ale powoli go rozbudujemy. Zwróć uwagę na funkcję `settings_fields()` wskazującą pole ustawień o nazwie `smashings_settingsdemo` utworzone za pomocą funkcji `add_settings_field()`

oraz funkcję `do_settings_sections()`, która wyświetla sekcję o podanej nazwie. Nie trzeba też samodzielnie tworzyć pliku zatwierdzania formularza, ponieważ można go wstawić za pomocą wywołania `submit_button()`.

Następnie tworzymy funkcję dla naszych ustawień i dodajemy sekcję ustawień za pomocą funkcji `add_settings_section()` oraz pole wejściowe za pomocą funkcji `add_settings_field()`:

```
// Funkcja ustawień
function smashings_settingsdemo_init(){

    // Dodanie sekcji
    add_settings_section('smashing_settings_section',
         'Smashing Settings',
         'smashing_settings_section_callback',
         'smashings_settingsdemo');

    // Dodanie pola ustawień
    add_settings_field('smashing_sample_input',
         'Input sample',
         'smashing_sample_input_callback',
         'smashings_settingsdemo',
         'smashing_settings_section');

    // Rejestracja ustawień
    register_setting( 'smashings_settingsdemo', 'smashing_sample_input',
         'smashing_settingsdemo_validate' );
}

// Inicjacja smashings_settingsdemo_init() w panelu administracyjnym
add_action( 'admin_init', 'smashings_settingsdemo_init' );
```


Nie jest to wcale takie skomplikowane. Najpierw przy użyciu funkcji `add_settings_section()` tworzymy sekcję o nazwie `smashing_settings_section`. Zwróć uwagę na wartość `smashing_settingsdemo` będącą parametrem `$page`. Używaliśmy jej już w funkcji `do_settings_sections()` w kodzie strony. Do nowo utworzonej sekcji dodajemy pole ustawień `smashing_sample_input`, w ostatnim parametrze funkcji `add_settings_field()` wpisując `smashing_settings_section`. Listy wszystkich parametrów funkcji `add_settings_section()` i `add_settings_field()` znajdują się w dokumentacji na stronach *http://codex.wordpress.org/Function_Reference/add_settings_section* i *http://codex.wordpress.org/Function_Reference/add_settings_field*.

Następnie dodajemy funkcję `smashings_settingsdemo_init()` do haka `admin_init`.

Następnie dodamy coś do sekcji tylko po to, aby pokazać, że jest to możliwe:

```
// Funkcja wykonywana na początku sekcji
function smashing_settings_section_callback() {
    echo '<p>Informacja na początku sekcji.</p>';
}
```

Nazwę funkcji `smashing_settings_section_callback()` widziałeś już w funkcji `add_settings_section()`. Jest to funkcja zwrotna, która będzie wywoływana na początku sekcji i będzie wyświetlała element p z tekstem.

W podobny sposób utworzymy funkcję zwrotną dla funkcji `add_settings_field()` o nazwie `smashing_sample_input_callback()`:

```
// Implementacja funkcji smashing_sample_input_callback()
function smashing_sample_input_callback() {
// Opuszczamy PHP na chwilę ?>
    <input type="text" name="smashing_sample_input" value="<?php echo
      get_option( 'smashing_sample_input' ); ?>" />
<?php }
// Powrót do PHP
```

Funkcja ta zawiera tylko pole wejściowe, którego wartość `smashing_sample_input` jest przekazywana przez funkcję `get_option()`. Wartość tę będziemy zapisywać i jest ona identyfikatorem pola, które utworzyliśmy.

Na koniec musimy trochę oczyścić dane wprowadzane do formularza. W tym przykładzie użyjemy tylko funkcji `esc_attr()`, ale jeśli chcesz, możesz skorzystać też z innych metod oczyszczających:

```
// Oczyszczanie
function smashing_settingsdemo_validate($input) {

    // Kodowanie
    $newinput = esc_attr($input);
    return $newinput;
}
```

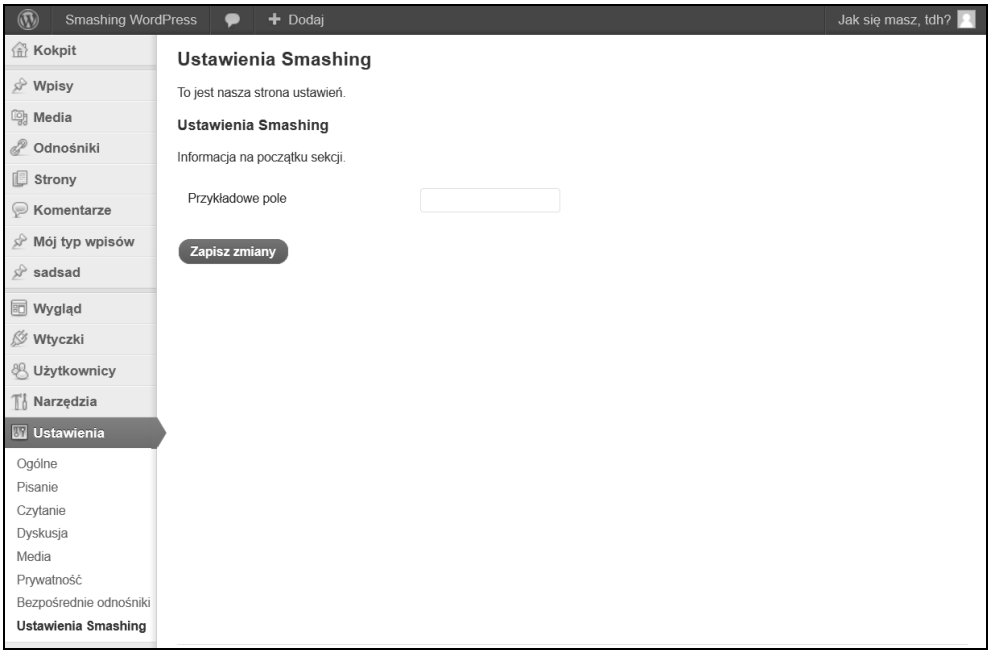
Co robi ten kod? Utworzyliśmy stronę ustawień zawierającą pola ustawień tworzone za pomocą funkcji `settings_fields()` (w tym przykładzie tylko jedno, ale można dodać więcej) i sekcje utworzone za pomocą funkcji `do_settings_section()`. Poniżej znajduje się kompletny kod wtyczki.

```
<?php
/*
Plugin Name: Ustawienia Smashing
Plugin URI: http://tdh.me/wordpress/ustawienia-settings/
Description: Prosta wtyczka ustawień.
Author: Thord Daniel Hedengren
Author URI: http://tdh.me/
*/

// Dodaje stronę ustawień do menu
add_action( 'admin_menu', 'smashings_settingsdemo_add_page' );

// Dodanie do menu
function smashings_settingsdemo_add_page() {
    add_options_page( 'Przykład ustawień Smashing',
```

```
        'Ustawienia Smashing',
        'manage_options',
        'smashings_settingsdemo',
        'smashings_settingsdemo_do_page' );
}

// Rzeczywiste dodanie strony ustawień
function smashings_settingsdemo_do_page() {
// Opuszczamy PHP na moment ?>

    <h2>Ustawienia Smashing</h2>
    <p>To jest nasza strona ustawień.</p>
    <form action="options.php" method="post">
    <?php settings_fields( 'smashings_settingsdemo' ); ?>
    <?php do_settings_sections( 'smashings_settingsdemo' ); ?>
    <?php submit_button(); ?>
    </form>

<?php
} // Powrót do PHP

// Funkcja ustawień
function smashings_settingsdemo_init(){

    // Dodanie sekcji
    add_settings_section('smashing_settings_section',
        'Ustawienia Smashing',
        'smashing_settings_section_callback',
        'smashings_settingsdemo');

    // Dodanie pola ustawień
    add_settings_field('smashing_sample_input',
        'Przykładowe pole',
        'smashing_sample_input_callback',
        'smashings_settingsdemo',
        'smashing_settings_section');

    // Rejestracja ustawień
    register_setting( 'smashings_settingsdemo', 'smashing_sample_input',
        'smashing_settingsdemo_validate' );
}

// Inicjacja smashings_settingsdemo_init() w panelu administracyjnym
add_action( 'admin_init', 'smashings_settingsdemo_init' );

// Funkcja wykonywana na początku sekcji
function smashing_settings_section_callback() {
    echo '<p> Informacja na początku sekcji.</p>';
}

// Implementacja funkcji smashing_sample_input_callback()
function smashing_sample_input_callback() {
```

```
// Opuszczamy PHP na moment ?>
    <input type="text" name="smashing_sample_input" value="<?php echo
      get_option( 'smashing_sample_input' ); ?>" />
<?php }
// Powrót do PHP

// Oczyszczanie
function smashing_settingsdemo_validate($input) {

    // Kodowanie
    $newinput = esc_attr($input);
    return $newinput;

}

?>
```

W ten sposób utworzyliśmy bardzo prostą stronę ustawień widoczną na rysunku 7.4. Za jej pomocą można zapisywać ustawienia w bazie danych.



Rysunek 7.4. Prosta strona ustawień z tekstem i polem tekstowym

BAZA DANYCH A ODINSTALOWYWANIE WTYCZKI

Tworząc wtyczkę przechowującą w bazie danych jakieś informacje, należy zastanowić się, co zrobić, gdy ktoś, kto ją zainstalował, zechce z niej po pewnym czasie zrezygnować. Czy wtyczka powinna po sobie sprzątać? W większości przypadków tak, zwłaszcza jeśli zapisała w bazie danych sporo informacji, które nie powinny tam zalegać bezużytecznie.

Jest kilka sposobów na usunięcie niepotrzebnych danych. Jednym z nich jest utworzenie pliku *uninstall.php* zawierającego kod usuwający z bazy danych treść, która została dodana tam przez wtyczkę:

```
delete_option( 'my-data' );
```

Powyższa instrukcja spowoduje usunięcie pola `my-data` z tabeli `option` bazy danych. Jako że wiele wtyczek zapisuje różne opcje w bazie danych, tabela ta szybko może stać się bardzo zabałaganiona, a to niczemu dobremu nie służy. Oczywiście w swoim pliku *uninstall.php* powinieneś zapisać własne instrukcje usuwające niepotrzebne dane. To samo dotyczy deinstalacji wykonywanych poprzez panel administracyjny.

Oto przykładowa zawartość pliku *uninstall.php*:

```
<?php
    // Dla starych wersji
    if ( !defined( 'WP_UNINSTALL_PLUGIN' ) ) {
        exit();
    }
    // Usuwanie danych opcji
    delete_option( 'myplugindata_post' );
    delete_option( 'myplugindata_feed' );
?>
```


W pierwszej części skryptu sprawdzamy, czy funkcja deinstalacji jest dostępna. W starszych wersjach WordPressa nie było jej, a więc jeśli używasz systemu w wersji starszej niż 3.0 (choć nie powinieneś!), skrypt zakończy działanie, nic nie robiąc. Dzięki temu ten plik jest zgodny ze starszymi wersjami WordPressa. Pozostały kod służy do usuwania danych z bazy: `myplugindata_post` i `myplugindata_feed`. Instrukcje te są wykonywane podczas usuwania wtyczki z panelu administracyjnego, dzięki czemu po deinstalacji wtyczki baza danych jest od razu posprzątana.

Ważne jest, aby pamiętać o zaimplementowaniu funkcji deinstalacji, jeśli wtyczka zapisuje cokolwiek w bazie danych. Można też pozwolić użytkownikowi wybrać, czy chce usunąć dane, czy woli je pozostawić, aby móc ich użyć w przyszłości. Dobrym pomysłem jest też usuwanie danych przy wyłączaniu, ponieważ zaleca się zrobienie tego dla wszystkich wtyczek podczas ręcznego uaktualniania WordPressa.

PO DEINSTALACJI

Łatwo jest zapomnieć, że po odinstalowaniu wtyczki w systemie mogą jeszcze być jakieś jej pozostałości. Dane w bazie danych to jedno (użytkownik powinien mieć przynajmniej możliwość wyboru, czy chce je usunąć), ale jest jeszcze jedna rzecz, która może być nawet bardziej kłopotliwa: tzw. skróty kodowe (ang. *shortcode*).

Co się dzieje, gdy zostanie odinstalowana wtyczka tworząca skróty kodowe? Skróty kodowe to specjalne ciągi znaków, które powodują wyświetlenie treści we wpisie w miejscu, w którym zostaną umieszczone. Jednym z najczęściej używanych w WordPressie skrótów tego typu jest

`[gallery]`. Można go zobaczyć, gdy doda się galerię do wpisu, a następnie przełączy się edytor w tryb HTML.

Co dzieje się ze skrótami wtyczki, gdy wtyczka ta zostanie odinstalowana? Nie zostaną przetworzone przez system jako skróty, tylko wyświetlone jak zwykły tekst. W tekście wówczas pojawią się napisy typu `[mojshortcode]`.

To nie będzie dobrze wyglądać.

Dlatego na wypadek gdyby wtyczka została wyłączona albo odinstalowana, powinniśmy udostępnić jakieś rozwiązanie kwestii skrótów kodowych. Rozwiązanie to musi umożliwiać użytkownikowi łatwe pozbycie się nieprzetwarzanych ciągów z tekstu wpisów. Jednym ze sposobów jest napisanie zapytania SQL po prostu usuwającego wszystkie wystąpienia danego skrótu, ale to jest dość drastyczne posunięcie i nie da się przewidzieć, co się stanie, gdy coś w trakcie procesu usuwania się nie powiedzie. W ten sposób można uszkodzić całą bazę danych. Poza tym nie wiadomo, czy z powodu błędu użytkownika wtyczka nie usunie za dużego fragmentu tekstu.

Sposób radzenia sobie z niepotrzebnymi skrótami kodowymi zależy od sposobu działania wtyczki. Bardziej szczegółowo kwestią tą zajmuję się w rozdziale 8.

Oczywiście nie z każdą wtyczką jest ten problem. Na przykład wtyczki tworzące tylko widżety nie są problematyczne, ponieważ po ich wyłączeniu widżety po prostu znikają.

WTYCZKI TWORZĄCE WIDŻETY

Za pomocą widżetów można łatwo dostosować sposób wyświetlania treści w blogu lub witrynie. Widżety umieszcza się w specjalnie wyznaczonych do tego obszarach za pośrednictwem panelu administracyjnego. W WordPressie dostępnych jest kilka standardowych widżetów, np. wyświetlający kanały RSS, najnowsze wpisy, listę stron, listę kategorii itp. Funkcjonalność tych widżetów może być niewystarczająca dla użytkownika i dlatego tworząc wtyczkę, można dać użytkownikom możliwość skorzystania z niej także w formie widżetu. Jest to o wiele lepsze niż zmuszanie użytkownika do wpisywania tagów szablonowych w plikach PHP motywu. Jeżeli więc funkcjonalność Twojej wtyczki ku temu przemawia, warto umożliwić używanie jej jako widżetu.

Tworzenie widżetów dla wtyczek nie jest trudne, głównie dzięki API widżetów, którego szczegółowy opis można znaleźć na stronie *http://codex.wordpress.org/Widgets_API*. Polega to na rozszerzeniu wbudowanej klasy `WP_Widget`, przekazaniu kilku instrukcji i zarejestrowaniu widżetu, aby mógł być wyświetlany, na przykład:

```
class SmashingWidget extends WP_Widget {
    function SmashingWidget() {
        // Kod widżetu
    }
    function widget( $args, $instance ) {
        // Zwrócenie treści widżetu
```

```
    }
    function update( $new_instance, $old_instance ) {
        // Przetworzenie i zapisanie opcji widżetu
    }
    function form( $instance ) {
        // Wyświetlenie formularza opcji w panelu administracyjnym
    }
}
register_widget( 'SmashingWidget' );
```

W tym przykładzie utworzyliśmy podklasę klasy `WP_Widget` o nazwie `SmashingWidget`. Pierwsza funkcja, `function SmashingWidget()`, zawiera rzeczywisty kod widżetu, a więc to ona jest odpowiedzialna za jego działanie. Funkcje `widget()`, `update()` i `form()` pozwalają sprawić, aby widżet zachowywał się tak, jak chcemy. Oczywiście widżet należy zarejestrować za pomocą funkcji `register_widget()`. Łącza *Anuluj* i *Zapisz* są wbudowane w API widżetów, a więc nie musimy implementować ich obsługi, aby użytkownik mógł zapisać lub anulować swoje ustawienia.

TWORZENIE WIDŻETU

W tym podrozdziale przedstawiony jest krok po kroku proces tworzenia widżetu. Opisany tu widżet będzie wyświetlał tekst powitalny oraz będzie można zmienić jego tytuł w panelu administracyjnym:

1. Pamiętaj, że cały opisywany kod powinien znajdować się w pliku PHP wtyczki zawierającym nagłówek identyfikacyjny. Jeśli nie masz żadnej wtyczki, którą mógłbyś teraz rozbudować, utwórz nową.

   Zaczniemy od utworzenia klasy widżetu:

   ```
   class SmashingHello extends WP_Widget {
   ```

   Ten widżet będzie nazywał się `SmashingHello`, dzięki czemu od razu wiadomo, co prawdopodobnie będzie robił.

2. Następnie definiujemy funkcję widżetu:

   ```
   function SmashingHello() {
       parent::WP_Widget( false, $name = 'Witajcie, cześć i czołem' );
   }
   ```

3. Do wykonania wielu czynności potrzebne są też funkcje `widget()`, `update()` i `form()`. Zaczniemy od definicji funkcji `widget()`:

   ```
   function widget($args, $instance) {
       extract( $args );
       ?>
           <?php echo $before_widget; ?>
               <?php echo $before_title
                   . $instance['title']
                   . $after_title; ?>
               Cześć! Czy to nie jest wspaniałe?
   ```

```
        <?php echo $after_widget; ?>
    <?php
}
```

W funkcji tej pobieramy argumenty. Zwróć uwagę na ustawienia `$before_widget`, `$after_widget`, `$before_title` oraz `$after_title`. Nie należy ich zmieniać, jeśli nie jest to konieczne. Są one kontrolowane przez API widżetów i domyślne funkcje motywów i dzięki nim widżety dobrze wyglądają.

Wartości zmiennych `$before_widget` i `$before_title` po prostu wysyłamy na wyjście, nie robiąc z nimi niczego szczególnego, a więc po prostu otrzymamy domyślny kod. Następna jest zmienna `$instance` reprezentująca tytuł widżetu, który użytkownik może wpisać w polu tekstowym dostępnym w panelu administracyjnym. Dalej znajduje się zmienna `$after_title`, a za nią tekst, który zostanie wyświetlony jako treść widżetu: *Cześć! Czy to nie jest wspaniałe?*. To jest oczywiście tylko przykład, więc nie ma tu żadnych fajerwerków, ale w tym miejscu możesz umieścić dowolny kod, choćby pętlę WordPressa. Na końcu widżet zamyka zmienna `$after_widget`.

Przypomnę, że zmienne ze słowem `before` i `after` w nazwie umożliwiają zmuszenie widżetu do zachowywania się zgodnie z konstrukcją motywu. Jest to kwestia zależna od projektanta motywu, a więc jeśli chcesz, aby Twój widżet był prawidłowo wyświetlany we wszystkich motywach, pozostaw ustawienia domyślne.

4. Następnie trzeba zadbać o to, aby widżet był poprawnie zapisywany w przypadku aktualizacji:

```
function update($new_instance, $old_instance) {
    return $new_instance;
}
```

Funkcja `update()` pobiera tylko argumenty `$new_instance` i `$old_instance`. Oczywiście zwraca `$new_instance`, ponieważ jest to reprezentacja zmian. Jeśli obawiasz się nieprzyjaznego kodu HTML, możesz zastosować filtrowanie znaczników przy użyciu funkcji `strip_tags()`. Użycie tej funkcji jest bardzo łatwe, poniżej znajduje się przykład dla pola wejściowego o nazwie `music`:

```
$instance['music'] = strip_tags( $new_instance['music'] );
```

W tym kodzie funkcja `strip_tags()` pilnuje, aby nie przedostał się żaden niepożądany kod HTML — jest to bardzo przydatne.

5. Teraz dodamy jeszcze jedno ustawienie pozwalające zmienić tytuł widżetu:

```
function form( $instance ) {
    $title = esc_attr( $instance['title'] );
    ?>
        <p>
            <label for="<?php echo $this->get_field_id( 'title' ); ?>">
                Tytuł: <input class="widefat" id="<?php echo $this->
                get_field_id( 'title' ); ?>" name="<?php echo $this->
                get_field_name( 'title' ); ?>" type="text" value="<?php
                  echo $title; ?>" />
            </label>
```

```
            </p>
        <?php
}
```

Kluczowe w tym przypadku są funkcje `get_field_name()` i `get_field_id()`. Pierwsza określa nazwę, a druga identyfikator elementu. W ten sposób został utworzony formularz ustawień widżetu, który można zapisać za pomocą przycisku *Zapisz* automatycznie tworzonego przez API widżetów.

6. Na koniec zamykamy klasę klamrą i rejestrujemy widżet:

```
}

function smashing_widget_init() {
    register_widget( 'SmashingHello' );
}

add_action( 'widgets_init', 'smashing_widget_init' );
```

Gotowy widżet jest przedstawiony na rysunku 7.5.

Rysunek 7.5. Utworzony widżet umieszczony na pasku bocznym w panelu administracyjnym

Masz już gotowy widżet, któremu możesz ustawiać tytuł i który wyświetla tekst. Oczywiście zamiast tekstu można wyświetlić cokolwiek, ponieważ wynik działania widżetu jest po prostu wynikiem działania skryptu PHP.

Należy też pamiętać, że nie wszystkie widżety muszą przyjmować opcje. Jeśli chcesz tylko umożliwić umieszczenie widżetu w obszarze widżetów, to nie twórz formularza ustawień. Nie ma sensu dodawać niepotrzebnych funkcji.

WIDŻETY KOKPITU

Można tworzyć nie tylko zwykłe widżety, ale również widżety kokpitu, czyli takie, które umieszcza się w panelu administracyjnym WordPressa potocznie nazywanym **kokpitem**. Widżetami są wszystkie ramki, które widać po wejściu do panelu administracyjnego; można też tworzyć własne takie ramki.

Aby utworzyć widżet kokpitu, należy utworzyć wtyczkę, a więc też i nowy plik. Poniżej znajduje się krótkie przypomnienie dla użytkowników grupowego bloga, żeby weszli na wewnętrzną stronę. Składa się ono tylko z tekstu i odnośników. Najpierw należy utworzyć odpowiednią funkcję:

```
function dashboard_reminder() {
    echo '
        Hej! Nie zapomnijcie przeczytać ważnych informacji na wewnętrznych
        stronach:<br />
        &larr; <a href="http://domain.com/internal/forum">Forum</a><br />
        &larr; <a href="http://domain.com/internal/docs">Dokumentacja</a><br />
        &larr; <a href="http://domain.com/internal/staff">Obsada</a><br />
        DZIĘKI!
     ';
}
```

Ta prosta funkcja o nazwie `dashboard_reminder()` wysyła na wyjście kod HTML, który stanowi treść widżetu. Następnym krokiem jest dodanie samego widżetu:

```
function dashboard_reminder_setup() {
    wp_add_dashboard_widget( 'dashboard_reminder_widget', 'Przypomnienie',
         'dashboard_reminder' );
}
```

Najważniejsza w tym kodzie jest funkcja `wp_add_dashboard_widget()`, której przekazaliśmy identyfikator widżetu (`dashboard_reminder_widget`), etykietę tekstową widżetu oraz nazwę funkcji zawierającej treść widżetu (`dashboard_reminder()`). Warto też wiedzieć, że identyfikator widżetu będący pierwszym parametrem funkcji `wp_add_dashboard_widget()` zostanie również użyty jako klasa tego widżetu, za pomocą której można go dowolnie sformatować przy użyciu CSS.

Zatrzymamy się na moment przy funkcji `wp_add_dashboard_widget()`. Ma ona jeszcze jeden parametr, o nazwie `$control_callback`, który jest opcjonalny i domyślnie przyjmuje wartość `null`. Nie został użyty w tym przykładzie, ale warto wiedzieć o istnieniu jeszcze jednego parametru, który również może być przydatny.

Wracając do przykładu, musimy jeszcze dodać akcję widżetu do haka `wp_dashboard_setup` za pomocą funkcji `add_action()`:

```
add_action( 'wp_dashboard_setup', 'dashboard_reminder_setup' );
```

To wszystko, widżet kokpitu jest już gotowy (rysunek 7.6)! Na razie nie istnieje żadne API pozwalające określać kolejność widżetów w panelu administracyjnym, a więc nasz widżet zostanie wyświetlony na dole. Użytkownik może go przesunąć w dowolne miejsce, ale są też sposoby na wyniesienie widżetu na górę automatycznie. Jeśli Cię to interesuje, zajrzyj np. do dokumentacji — na stronę *http://codex.wordpress.org/Dashboard_Widgets_API*.

Wiesz już, jak tworzyć wtyczki i widżety, a więc czas zająć się kwestią bazy danych. Zapisać informacje w bazie danych jest bardzo łatwo, ale nie zawsze jest to najlepsze rozwiązanie.

Rysunek 7.6. Widżet kokpitu w całej okazałości

KWESTIA KORZYSTANIA Z BAZY DANYCH WE WTYCZKACH

Czasami aby wtyczka działała prawidłowo, konieczne jest umożliwienie jej zapisywania informacji w bazie danych. Baza danych to bardzo przydatny magazyn, w którym można przechowywać wszystko — od prostych ustawień konfiguracyjnych po całe tabele danych do użytku użytkowników. Dane umieszczone w bazie danych bardzo łatwo jest pobrać i jest to bardzo wygodne rozwiązanie.

Niestety ta wygoda ma swoją cenę. Brak porządku w bazie utrudnia jej utrzymanie i dlatego trzeba pamiętać o tym, aby użytkownik mógł wraz z wtyczką usuwać dane tej wtyczki z bazy danych. Ponadto trzeba zdecydować, gdzie należy zapisywać dane. Można do tego użyć API ustawień, dzięki czemu mamy pewność, że dane trafią tam gdzie trzeba — jest to dobre rozwiązanie w przypadku niewielkich ilości danych konfiguracyjnych — albo utworzyć własną tabelę przeznaczoną tylko dla naszej wtyczki. Drugie rozwiązanie jest zwykle lepsze w przypadkach, gdy trzeba zapisywać duże ilości danych. Rozszerzanie bazy danych może powodować problemy z innymi wtyczkami również korzystającymi z tej bazy. Ponadto taka dodatkowa tabela nie należy do WordPressa i trzeba oddzielnie tworzyć jej kopię zapasową, a ponadto jeśli zechcesz przenieść instalację w inne miejsce, nie zostanie ona standardowo uwzględniona przez narzędzia eksportu i importu.

Pozostaje też kwestia buforowania, którą warto rozważyć w przypadku niektórych wtyczek, zwłaszcza regularnie pobierających dane z zewnętrznego źródła. Oczywiście do buforowania można używać plików, ale można też wykorzystać API Transients, które zostało właśnie do tego stworzone. Opis tego API znajduje się na stronie *http://codex.wordpress.org/Transients_API*.

Które rozwiązanie jest najlepsze? Zapisywać dane w opcjach, czy we własnej tabeli? Może należałoby użyć tabeli metadanych albo wpisów? Wybór zależy od tego, co chcesz przechowywać. Na podstawie własnego doświadczenia mogę powiedzieć, że ustawienia najlepiej zapisywać w tabeli opcji; realną treść i większe ilości danych często umieszczam w osobnej tabeli.

Pamiętaj tylko, aby poinformować użytkownika o sytuacji, i dodaj możliwość zrobienia porządku w bazie danych po odinstalowaniu wtyczki.

ZGODNOŚĆ WSTECZNA WTYCZEK

Kolejnym aspektem, jaki należy rozważyć podczas budowy wtyczki, jest to, czy wtyczka ta ma być zgodna ze starszymi wersjami systemu. Do WordPressa wciąż są dodawane nowe funkcje i haki i jeśli będziemy używać tych najnowszych, to naturalnie w starszych wersjach systemu nasza wtyczka nie będzie działać. Dlatego trzeba ostrożnie korzystać z najnowszych dodatków do systemu, ponieważ mogą one nie działać w jego starszych wersjach.

Co więcej, nie tylko funkcjonalność WordPressa się zmienia. Także wymagania systemowe co jakiś czas podlegają modyfikacjom. W tej chwili do działania systemu wymagany jest PHP 5.2.4. Wtyczka powinna obsługiwać te same wersje oprogramowania co sam WordPress, ale jeśli chcesz użyć nowszych wersji PHP lub MySQL, koniecznie wyświetl informację o błędzie, gdy ktoś spróbuje uruchomić ją w systemie ze starszymi wersjami PHP lub MySQL.

Jak daleko chcesz sięgać w przeszłość w swojej wtyczce, zależy tylko od Ciebie. Od kiedy wprowadzono możliwość automatycznej aktualizacji systemu, częstotliwość aktualizowania wzrosła. Jednak wciąż jest zaskakująco dużo witryn działających w oparciu o stare wersje WordPressa, co jest niekorzystne zarówno dla programistów, jak i właścicieli witryn. W każdym razie przestarzałe oprogramowanie jest mniej bezpieczne i powoduje problemy ze zgodnością z nowszymi modułami.

PUBLIKOWANIE WTYCZEK W PORTALU WORDPRESS.ORG

Dla wtyczek, podobnie jak dla motywów, istnieje oficjalny serwis w portalu WordPress.org, w którym można je publikować. Oczywiście nie musisz tego robić, ale dzięki temu użytkownicy otrzymują automatycznie powiadomienia o dostępności aktualizacji i mogą ich dokonywać na bieżąco.

Aby wtyczka została opublikowana w portalu WordPress.org, musi spełniać pewne wymagania:

- musi być na licencji zgodnej z GPL;
- nie może wykonywać nielegalnych ani „niemoralnych” działań;

- musi zostać wysłana do repozytorium Subversion WordPress.org;
- musi zawierać poprawny plik *readme.txt*.

Aby móc opublikować wtyczkę, trzeba być użytkownikiem portalu WordPress.org. Następnie wysyła się wtyczkę (*http://wordpress.org/extend/plugins/add*) i czeka na jej zatwierdzenie. Czas oczekiwania zależy od tego, ile aktualnie pracy ma zespół zajmujący się weryfikowaniem wtyczek.

Gdy wtyczka zostanie zatwierdzona, otrzymasz dostęp do katalogu Subversion, do którego możesz wysłać wtyczkę z plikiem *readme.txt*. Poprawność pliku *readme.txt* sprawdza specjalne narzędzie (*http://wordpress.org/extend/plugins/about/validator*). Weryfikuje ono, czy w pliku znajdują się wszystkie informacje potrzebne do publikacji wtyczki.

Przed wysłaniem wtyczki należy przeczytać odpowiedzi na najczęściej zadawane pytania dotyczące wtyczek opublikowane pod adresem *http://wordpress.org/extend/plugins/about/faq*. To na pewno usprawni proces zatwierdzania wtyczki.

Korzyścią z opublikowania wtyczki w zbiorach WordPress.org są nie tylko automatyczne aktualizacje, ale również prowadzenie statystyk. Dowiesz się, ile osób pobrało Twoją wtyczkę, oraz otrzymasz oceny i komentarze od użytkowników. Ponadto witryna WordPress.org jest centrum społeczności WordPressa, a więc stwarza największą szansę, że użytkownicy w ogóle znajdą wtyczkę. Jeśli opublikujesz ją na własnym serwerze, to z jej znalezieniem może być różnie, tymczasem dzięki narzędziu wyszukiwania wtyczek w panelu administracyjnym Twoją wtyczkę będzie mógł znaleźć i zainstalować dosłownie każdy użytkownik WordPressa.

Ale warunkiem jest obecność w zbiorach WordPress.org. Dlatego postaraj się tam zaistnieć ze swoją wtyczką!

Szukasz innego miejsca do opublikowania swojej wtyczki niż WordPress.org? Sprawdź GitHub (http://github.com). Jest to serwis do publikowania programów z otwartym kodem źródłowym. Bardzo wygodnie się z niego korzysta, zwłaszcza jeśli zna się obsługę programu Git. Jeśli go nie znasz, zawsze możesz poznać.

SŁOWO OSTRZEŻENIA NA TEMAT TWORZENIA WTYCZEK

Wtyczka to nie to samo co motyw. Oczywiście można znaleźć jakieś podobieństwa w procesie powstawania, ale jednak w przypadku wtyczek tak naprawdę pisze się zwykłe skrypty PHP podłączone do WordPressa. W związku z tym podczas gdy każdy posiadający podstawową wiedzę na temat pisania skryptów może nagiąć WordPressa do swoich potrzeb w motywie, z pisaniem kodu wtyczek nie będzie już miał tak łatwo. Do tego potrzebna jest praktyczna umiejętność pisania skryptów PHP i trzeba przy tym bardzo uważać, ponieważ łatwo można coś zepsuć, zwłaszcza jeśli szpera się w bazie danych.

Biorąc pod uwagę te kwestie i znając się na programowaniu w PHP, za pomocą wtyczek możesz sprawić, że Twoja witryna będzie działała dokładnie tak, jak sobie zaplanujesz.

W następnym rozdziale dowiesz się, kiedy używać wtyczek, a kiedy lepiej jest pozostać przy pliku *functions.php*.

Skorowidz

G

H

I

J

K

L

M

0

P

R

S

T

U

V

W

Z

KOMPLETNE I PRZYJAZNE ŹRÓDŁO INFORMACJI O PLATFORMIE WORDPRESS!

Dwóch gigantów postanowiło połączyć swoje potencjały. Razem stworzyli niezwykłą książkę, którą właśnie trzymasz w rękach. Jednym z nich jest Smashing Magazine, należący do najpopularniejszych serwisów poświęconych tworzeniu stron WWW i nie tylko, a drugim WordPress, lider wśród systemów do prowadzenia blogów. Obok tej pozycji nie możesz przejść obojętnie!

W tej niepowtarzalnej i wyśmienitej książce znajdziesz zbiór najświeższych, najciekawszych i najlepszych informacji poświęconych systemowi WordPress. W czasie lektury dowiesz się, jak używać tej platformy jako systemu CMS oraz narzędzia do blogowania czy projektowania witryn dowolnego typu. Ponadto nauczysz się tworzyć genialne motywy, przydatne wtyczki oraz integrować Wordpressa z portalami społecznościowymi. Szczególną uwagę powinieneś zwrócić na zagadnienia poświęcone SEO — przestrzeganie dobrych zasad z pewnością ułatwi Ci zaistnienie w sieci. Książka ta jest obowiązkową pozycją dla każdego użytkownika platformy WordPress!

Tylko krok dzieli Cię od:

- szczegółowego poznania platformy Wordpress
- przygotowania własnego motywu graficznego
- poprawnej konfiguracji, zgodnej z SEO
- odniesienia sukcesu w sieci WWW

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-6678-2
9 788324 666782
Cena: 59,00 zł