

# Strategie doskonałości Najprostsze możliwe środowisko



*Mariusz Sierackiewicz*  
<http://msierackiewicz.blogspot.com>  
<http://www.bnsit.pl>  
<http://www.lodz.jug.pl>

Możesz swobodnie dystrybuować ten plik PDF, jeśli pozostawisz go w nietkniętym stanie. Możesz także cytować jego fragmenty, umieszczając w tekście odnośnik

<http://msierackiewicz.blogspot.com>

# Pracuj w możliwie najprostszym możliwym środowisku

Kto nie uwielbia tych wspaniałych chwil, kiedy projekt nabiera kształtu, kolejne funkcjonalności pojawiają się jedna za drugą i nie możemy się nacieszyć wspaniałym programistycznym dziełem. Pielęgnowujemy każdy fragment, aby nasze oprogramowanie było jeszcze wspanialsze.

Jest to również moment, kiedy projekt zaczyna stawać się coraz bardziej złożony. Jeśli na przykład pracujemy nad aplikacją internetową, to przychodzi taki moment, kiedy aplikacja obsługuje logowanie, transakcje, współpracę z bazą danych, wielojęzyczność, złożone reguły bezpieczeństwa. Istnieje również pewna logika przetwarzania zdarzeń użytkownika – najczęściej korzystamy z jakiegoś frameworka MVC. System rzeczywiście jest już całkiem złożony. Restart całej aplikacji zajmuje kilkanaście a czasem kilkadziesiąt sekund, a nawet kilka minut.

Każdy dzień pracy w takim środowisku powoduje, że się przyzwyczajamy do tej całej złożoności. Co więcej, orientujemy się w niej coraz lepiej, czujemy się jak ryba w wodzie, korzystając z dobrze już znanych niuansów. Osiadamy w przyzwyczajeniu.

Rozwijamy dalej aplikację - tworzymy nową funkcjonalność. Niech będzie to ekran z wykresami tworzonymi na podstawie danych przechowywanych w systemie. Jeszcze nigdy nie mieliśmy do czynienia z wykresami, a przynajmniej nie z tą biblioteką, z której mamy skorzystać. Zatem ucząc się umiejętności związanych z tworzeniem wykresu, zaczynamy osadzać nowe rozwiązanie w złożonym systemie. W międzyczasie musimy przetestować kilka wariantów i sprawdzić, jak dokładnie działa komponent, którego

chcemy użyć po raz pierwszy.

Zatem eksperymentujemy z różnymi parametrami. Zanim dokonamy testów, musimy odpowiednio skonfigurować framework MVC, odpowiednio osadzić komponent wykresu na stronie lub panelu, być może dodać elementy związane z bezpieczeństwem i wielojęzycznością. A wszystko to pojawia się tylko po to, żeby nauczyć się jak wykorzystywać komponent.

A przecież dopiero się uczymy...

Dopiero eksperymentujemy!

Nie jest nam potrzebny cały ten narzut.

Nagle jeden z parametrów nie do końca działa tak jak byśmy chcieli. W zasadzie nie wiemy o co chodzi. Jednocześnie cały system się nie uruchamia, bo na szybko oprogramowaliśmy controller (z MVC), więc przy okazji zmagamy się z jego błędami – problem z dostępem do danych.

W końcu udało się. Wracamy do naszego komponentu, ale jeszcze musimy zrestartować aplikację. Trwa to dość długo. Zbyt długo.

Czy w ogóle to całe środowisko jest potrzebne do przetestowania komponentu? Narzut ogromny – należy stworzyć wszystkie elementy potrzebne do uruchomienia testowego fragmentu, które narzuca środowisko. Czas potrzebny na efektywne badanie kolejnych parametrów jest subiektywnie oceniając wydłużony o 10-50 % z powodu złożoności systemu, w którym proces się odbywa.

A gdyby mieć pod ręką najprostsze możliwe środowisko aplikacji webowej (czasami dostarczane razem z bibliotekami jako blank application) i używać je do testowania nowych elementów lub prostych fragmentów funkcjonalności. Bez narzutu pozostałych elementów środowiska, takich jak bezpieczeństwo, wielojęzyczność, logowanie! Najprościej jak to tylko możliwe. Oczywiście, kiedy już

uda się przeprowadzić próby na prostym środowisku, przenosimy rozwiązanie do złożonego systemu, w którym pracujemy.

Być może nie tworzysz aplikacji webowych, ale z pewnością znajdziesz analogie w obszarze, w którym się specjalizujesz. Ogólny schemat postępowania jest taki sam bez względu na technologię.

Być może wydaje się to bardzo oczywiste -bo jest oczywiste. Ale dlatego tak często o tym zapominamy i tak często tracimy niepotrzebnie czas i często szargamy swój system nerwowy analizując wyjątki z warstwy danych, mimo że w tym momencie zajmujemy się eksperymentowaniem z komponentem wykresu. Wiele razy byłem świadkiem (lub uczestnikiem) takiej sytuacji!

Całość powyższych rozważań można sprowadzić do zasadniczej myśli:

*Pracuj w możliwie najprostszym  
środowisku*

Bardzo łatwo ulec przekonaniu, które z pewnością w tym momencie przychodzi do głowy: „... ale w moim środowisku nie można nic uprościć! Nie można przenieść tego, co robię, w inne, prostsze środowisko...”. Mogę cię zapewnić Czytelniku, że prawie zawsze można środowisko uprościć. Główne pytanie, na które trzeba sobie odpowiedzieć, to czy rzeczywiście warto, ponieważ bywają sytuacje, w których stworzenie uproszczonego środowiska lub przygotowania środowiska, będzie mało opłacalne. Jednak w dużej części przypadków warto to robić, tym bardziej że wypracowane rozwiązanie często może zostać użyte podczas prac nad następnymi funkcjonalnościami. Zatem poniesiony wysiłek

zaprocentuje jeszcze wielokrotnie.

Kilka praktycznych sposobów na upraszczanie środowiska opisałem poniżej. Jestem przekonany, mój Czytelniku, że będą stanowić inspiracje, dzięki którym będziesz pracował jeszcze efektywniej i ten wspaniały zawód programisty, stanie się jeszcze wspanialszy.

## Twórz uproszczone konteksty pracy

W zasadzie przedstawiony na wstępie przykład jest ucieleśnieniem tej zasady.

Kiedy chcesz sprawdzić nowy komponent, bibliotekę, zastanów się jak możesz najprościej ją przetestować i poznać jej właściwości. Zazwyczaj złożona aplikacja, nad którą właśnie pracujemy, jest ostatnim miejscem nadającym się do tego celu.

Jeśli nigdy dotąd nie używałeś wyrażeń regularnych, a teraz będą ci potrzebne, zanim zaczniesz z nimi pracę najzwyczajniej świecie stwórz klasę z metodą main (tu mam na myśli Javę, ale każdy w to miejsce może wstawić sposób, w jaki tworzy się słynne już HelloWorld – bez narzutów technologicznych). I tam przetestuj wszystkie interesujące cię przypadki. W miejsce wyrażeń regularnych można wstawić każdą dowolną inną funkcjonalność: obiekty funkcyjne z jakarta-commons, generowanie pdf-ów, kopiowanie danych, łączenie się z serwerem SMTP i wysyłanie maili, bibliotekę do współpracy z bazą danych, funkcje statystyczne. Można wymieniać bez liku.

Kiedy już pracujesz w złożonym środowisku, np. budujesz strony internetowe z użyciem frameworka MVC, i chcesz przetestować nowy komponent graficzny, zazwyczaj lepiej zrobić to w środowisku poza projektem, a nie w aplikacji, którą właśnie tworzysz.

Jeśli chcesz przetestować działanie pola tekstowego, na który jest nałożona maska pewnego wzorca, który określa format danych przyjmowanych przez to pole, zrób to na możliwie najprostszym przykładzie, poza aplikacją. Zyskasz sporo czasu. Jeśli zrobisz to jeden raz – drugi, trzeci i kolejne będą coraz łatwiejsze.

## Testuj jednostkowo i hermetyzuj logikę

Jeśli kiedyś zastanawiałeś się, do czego mogą przydać ci się testy jednostkowe, to mogę cię zapewnić, że między innymi mogą posłużyć jako świetne narzędzie do upraszczania środowiska pracy. Po pierwsze są świetnym zamiennikiem dla metody main – dają większe możliwości tworzenia bardziej złożonych przypadków testowania. Dodatkowo dzięki użyciu *mock objects* jesteśmy w stanie pracować z minimalnym podzbiorem logiki biznesowej.

Na przykład jeśli pracujemy nad metodą lub funkcją, która (1) przygotowuje treść maila i (2) go wysyła, nie musimy go fizycznie wysyłać. Podstawiamy *mock object* (bardzo uproszczoną implementację do celów testowych) i nie musimy czekać za każdym razem, żeby mail został wysłany. Oczywiście samo wysyłanie wiadomości też można przetestować. Ale można zrobić to osobno, a ewentualnie na końcu, kiedy oba mechanizmy (1) i (2) działają, przetestować je razem. Zgodnie z zasadą

*W danej chwili testuj, zmieniaj lub pracuj nad jedną rzeczą*

(ta zasada nie ogranicza się do programowania!)

Copyright © 2008 Mariusz Sierackiewicz. Pewne prawa zastrzeżone.

<http://msierackiewicz.blogspot.com> <http://www.bnsit.pl> <http://www.lodz.jug.pl>

Chciałbym zaznaczyć jedną rzecz. Stosowanie powyższych zasad nie uchroni cię przed błędami, ale pozwoli ci zmniejszyć ilość sytuacji, w których błędy się będą pojawiać. Wyobraź sobie, że dzięki nawykowi upraszczania środowiska oszczędzisz godzinę dziennie (a możesz z pewnością więcej), w ciągu tygodnia oszczędzasz 5 godzin, w ciągu miesiąca 20 godzin. W ciągu roku ... ?

Powróćmy jednak do testów. Żeby w pełni używać testów jednostkowych musimy nauczyć się pewnej cennej umiejętności, którą nazywam tutaj hermetyzacją logiki. Tworząc oprogramowanie musimy tworzyć je modułowo, tak aby poszczególne części były jak najmniej między sobą zależne (ang. *coupling*), tak aby można było ich używać niezależnie. W przypadku systemów obiektowych istnieje cała filozofia projektowania obiektowego, której celem jest osiągnięcie stanu, w którym poszczególne części systemu są od siebie jak najmniej zależne.

Jednak zarówno w przypadku systemów obiektowych jak i nieobektowych można zastosować ogólniejszą *zasadę odpowiedzialności*, tzn. tak tworzyć system, aby wszystkie jego części miały jednoznacznie określoną odpowiedzialność, bez względu czy to będzie metoda, klasa, pakiet, procedura, funkcja czy formularz interfejsu użytkownika. Część taka będzie zamykać w sobie dobrze zdefiniowaną funkcjonalność, bez zbędnych powiązań z zewnętrznymi elementami.

Jeśli zatem kodujesz logikę aplikacji w interfejsie użytkownika, jesteś daleki od realizacji tej zasady. W ten sposób wyraźnie wiążesz ze sobą interfejs użytkownika i logikę, nie jesteś w stanie jej w prosty sposób rozdzielić i pracować nad nimi osobno. Jeśli tworzysz interfejs użytkownika, umieszczaj w tej części kod, który dotyczy tylko i wyłącznie interfejsu użytkownika – czyli budowania jego

wyglądu, ustawiania pól i ich odczytywania. W interfejsie użytkownika nie powinniśmy tworzyć złożonych algorytmów przetwarzania danych (no chyba, że dotyczą wyświetlania elementów na ekranie), do tego celu używamy niezależnych funkcji, metod, klas.

Z pomocą w tym przypadku przychodzą między innymi wzorzec MVC oraz model architektury wielowarstwowej. Zawsze jednak można się posłużyć *z d r o w o r o z s ą d k o w y m* stosowaniem zasady odpowiedzialności, której konsekwencją są wymienione w poprzednim zdaniu koncepcje.

Chciałbym jeszcze raz podkreślić słowo „zdroworozsądkowym” – nie zawsze MVC czy inne koncepcje tego typu są do zastosowania w każdej sytuacji. Istnieje być może 3 % sytuacji, kiedy warto zakodować logikę w interfejsie użytkownika. Ale to jest tylko 3%. No może 5%, albo 10%. Ale cały czas jest to odstępstwo od reguły!

## **W danej chwili testuj, zmieniaj lub pracuj nad jedną rzeczą**

Powyzsza reguła, choć już wspomniana wcześniej, jest tak istotna, że zasługuje na osobny punkt. Być może znacie odpowiedź na pytanie: „Jak zjeść słonia”. „Po kawałku”. Dokładnie ta reguła powinna towarzyszyć nam bezkompromisowo w codziennej pracy. A często nie towarzyszy. Możecie mi wierzyć, że często. Co więcej, również i mnie się zdarza czasami błędzić i jej nie stosować.

Zabierając się do realizacji większego zadania (takiego przynajmniej na 3-4 godziny), warto poświęcić kilka chwil na to, żeby zaplanować swoje działania. Zaplanować po to, by naszym słoniem, jak duży by

on nie był, się nie zadławić.

Pamiętam bardzo dobrze moje doświadczenia programistyczne, kiedy podejmując się jakiegoś zadania, tworzyłem kilka godzin pełne rozwiązanie i wtedy uruchamiałem je... po raz pierwszy. No cóż, nigdy nie działało w tym momencie, bo w zasadzie nie miało prawa. Następne kilka godzin spędzałem na debugowaniu i poprawianiu kodu. Myślałem, że uda mi się zjeść całego słonia, od razu.

Zatem co warto zrobić na początku realizacji zadania? Mały plan. Tak aby ustalić przybliżony sposób tworzenia rozwiązania fragment po fragmencie. Załóżmy, że mam napisać sieć neuronową. (Jeśli nie wiesz zbyt dobrze co to takiego, nie przejmuj się, nie ma to dużego znaczenia w dalszych rozważaniach.) Nie tworzę gotowego rozwiązania. Krok po kroku zaczynam implementować najprostsze rzeczy. Na początek być może warto zaimplementować funkcję symulującą działanie neuronu. Być może później stworzyć klasę neuronu. Przetestować jego metody. Następnie być może implementować sieć. Następnie regułę propagacji wstecznej. To tylko przykładowa kolejność. *Ważne, żeby w danym momencie rozwijać, zmieniać jedną rzecz na raz*. Kiedy ona zacznie działać i będzie przetestowana, brać się za następną. Oczywiście całość warto przemyśleć już na samym początku (przynajmniej na pewnym poziomie ogólności), ale implementować kawałek po kawałku, krok po kroku. Takie mikroiteracje. Ciekawą formą osiągnięcia opisanego wyżej efektu jest Test-Driven Development (aczkolwiek, nie jest to z pewnością jedyny sposób – na pewno warty uwagi).

Wiele razy widziałem sytuację, kiedy należało na przykład zaimplementować w aplikacji internetowej generowania raportu na podstawie danych przechowywanych w systemie i danych z

bieżącego formularza. Programista jednocześnie rozwijał akcję (klasa obsługująca zdarzenie z interfejsu użytkownika), zmieniał interfejs użytkownika i pracował nad generowaniem raportu do pliku pdf (testując przy okazji jak generować pliki pdf). Wyobraźcie sobie, jak wiele szczegółów należy ogarniać umysłem, aby zapanować nad tym wszystkim. W takiej sytuacji zazwyczaj wszystkie elementy interferują ze sobą, a że są nie w pełni skończone, więc trudno znaleźć przyczynę popełnianych błędów.

## Szukanie przyczyn błędów

Powyżej opisane sposoby mogą znacząco zmniejszyć ilość popełnianych błędów, ale bądźmy szczerzy – przyjdzie taki moment, kiedy przydarzy się sytuacja, w której nasz system lub jego fragment nie działa poprawnie. Jeśli stosujesz zasadę *W danej chwili testuj, zmieniaj lub pracuj nad jedną rzeczą*, to zadanie znalezienia przyczyny błędu masz znacząco ułatwione. Ponieważ dokonujesz niewielkich kroków rozwijając kod i łatwo możesz określić zmiany, które doprowadziły do wystąpienia błędu. Ta reguła działa, kiedy kroki są wystarczająco małe. Jeśli przez ostatnią godzinę dokonałeś mniej więcej kilkunastu zmian w plikach konfiguracyjnych, kodzie i w HTML-u, to znaleźć przyczynę będzie trudniej. Jeśli natomiast najpierw dokonałeś zmiany w pierwszym pliku konfiguracyjnym i całość zadziałała, później zmiany w drugim pliku konfiguracyjnym i całość zadziałała, następnie dokonałeś zmian w kodzie i tu pojawił się błąd – to wszystko jasne.

Oczywiście w życiu nie jest tak różowo i zdarzają się sytuacje, kiedy błędy pojawiają się po pewnym czasie (na przykład kilka dni po zmianach albo nawet po kilku miesiącach). Otóż w takich sytuacjach proponowana zasada brzmi następująco:

*Uprość dany fragment rozwiązania, tak aby zaczął działać, a następnie dodawaj kolejne elementy, aż do momentu wystąpienia błędu.*

Wtedy sytuacja będzie już jasna.

Założmy, że występuje taka sytuacja:

Aplikacja oparta o MVC – zdarzenie polega na tym, żeby pobrać odpowiednie dane i wygenerować plik pdf. Istnieje komponent, w jakiś sposób zainicjalizowany, który realizuje to zadanie:

1. pobiera dane ze źródła danych
2. na ich podstawie generuje obiekt potrzebny do wygenerowania pliku pdf
3. tworzony jest plik pdf
4. plik pdf wysyłany jest na ekran

Jak upraszczamy? Przykład postępowania (oczywiście szczegóły działania będą zależne od poszczególnego przypadku). Najpierw upewnijmy się, że komponent, którego używamy, jest poprawnie zainicjalizowany. Można na przykład podstawić za niego inny komponent (np. bardzo uproszczoną implementację komponentu) lub zainicjować go ręcznie, jeśli domyślnie jest tworzony na podstawie pliku konfiguracyjnego. Jeśli konfiguracja komponentu nie była przyczyną błędu, możemy zamiast pobierać dane ze źródła danych spreparować wyniki (badamy czy błąd związany jest z wynikami pochodzącymi ze źródła danych). Następnie możemy sami wygenerować sztuczny obiekt do tworzenia pliku pdf (lub go w ogóle nie tworzyć). Zamiast tworzyć pdf (strumień binarny) można pokusić się o proste dane znakowe (strumień znakowy) i w ten

sposób wykluczyć ewentualne błędy tworzenia pliku pdf i pokazywania go na ekranie. Jeśli na którymś etapie odkryjemy, że pojawia się błąd, analizujemy dany etap, stosując powyższy algorytm ograniczając go do tego etapu.

W procesie wyszukiwania błędów niezwykle przydatne są pliki logów czy systemy debugujące (choć te przy złożonych systemach, są bardzo czasochłonne). Są to jednak tylko narzędzia i mogą co najwyżej wesprzeć powyżej opisany proces. Całość pracy intelektualnej musimy wykonać sami.

Oczywiście przebieg postępowania będzie inny dla innej technologii czy innej sytuacji. Jesteś ekspertem w swojej dziedzinie, więc najlepiej będziesz wiedział jak tę strategię dostosować do środowiska, w którym działasz.

Skoro jesteśmy przy wyszukiwaniu błędów, to przy okazji jeszcze jedna wskazówka:

*Przyczyny błędów są najczęściej ukryte tam, gdzie dalibyśmy sobie głowę uciąć, że ich tam nie ma. Najprostsza i jedna z częstszych przyczyn błędów to literówka.*

## Podsumowanie

Powyższe przykłady miały nieco wyraźniej nakreślić regułę upraszczania środowiska, w którym rozwiązuje się dany problem czy zadanie. Ponieważ opisane powyżej aspekty dotyczą sfery naszych nawyków i przekonań, z pewnością Czytelniku kilkakrotnie mogłeś czuć wewnętrzny opór przed zaakceptowaniem ich treści lub stwierdzić, że w twoim przypadku nie ma to zastosowania. Jest to

całkiem naturalny proces, ponieważ przedstawiane powyżej tezy przynależą do świata nauk humanistycznych, a szczególnie społecznych i psychologicznych. Nie są to pewniki. Tezy te potwierdzają się w większości przypadków, ale nie muszą we wszystkich. Ale jak to się mówi... wyjątki potwierdzają regułę.

Dokładniejsze przyjrzenie się swoim nawykom i eksperymenty związane z proponowanymi strategiami na pewno mogą stanowić dobrą zabawę, czego ci Czytelniku życzę. I oczywiście zwiększania efektywności oraz wzbogacania przyjemności z tworzenia oprogramowania.

## O mnie



Mariusz Sierackiewicz

Trener, konsultant, menedżer projektów IT, coach. Założyciel zespołu programistów Equilibrium. Współinicjator JUGa Łódź. Autor artykułów o inżynierii oprogramowania. Współwłaściciel firmy szkoleniowej BNS IT. Szczęśliwy mąż :-)

<http://www.linkedin.com/pub/2/a24/812>

## O BNS IT



BNS IT jest firmą szkoleniowo-doradczą zajmującą się **rozwijaniem i doskonaleniem zespołów programistycznych.**

Nasze usługi skierowane są do firm zatrudniających programistów. Pracujemy dla trzech grup klientów:

- firm zajmujących się wytwarzaniem oprogramowania
- firm, które posiadają zespoły programistyczne, lecz działają na innych rynkach niż IT
- firm integrujących systemy informatyczne

<http://www.bnsit.pl/>

Copyright © 2008 Mariusz Sierackiewicz. Pewne prawa zastrzeżone.

<http://msierackiewicz.blogspot.com> <http://www.bnsit.pl> <http://www.lodz.jug.pl>