97 rzeczy które każdy programista powinien wiedzieć

Mateusz Książek

DEV
ENV

„*97 rzeczy które każdy programista powinien wiedzieć*" to zbiór zasad, które profesjonalista powinien przyswoić i starać się stosować w codziennej pracy. Zasady te są uniwersalne, ponieważ nie są ściśle związane z żadnym z paradygmatów programowania, technologią czy środowiskiem – dowiesz się raczej jak programista powinien zachowywać się podczas pracy w projekcie czy współpracując z innymi osobami.

Oryginalny zbiór został przygotowany przez wydawnictwo O'Reilly Media i wydany pod tytułem „*97 Things Every Programmer Should Know. Collective Wisdom from the Experts*" gdzie swoich rad udzieliły takie osoby jak Robert C. Martin, Dan North czy Heinz Kabutz. Przez lata zestawienie to było ogólnie dostępne, ponieważ było udostępnione przez wydawnictwo całkowicie za darmo.

W tym dokumencie znajdziesz streszczenie tych wszystkich zasad w języku polskim. Jak zobaczysz, niektóre z nich potrzebowały szerszego rozwinięcia, a niektóre zasady można opisać przy pomocy jednego zdania. Starałem się całkowicie przenieść sens i istotę wszystkich zaleceń ekspertów, ale czasem też wzbogacałem je o własne doświadczenia czy zewnętrzne linki.

Wszystkie streszczenia publikowane były na blogu devenv.pl w serii kilku postów. Postanowiłem jednak skompletować wszystkie elementy i stworzyć ten dokument by umożliwić jeszcze wygodniejszy sposób zapoznania się z tym ważnym materiałem.

Niech ten dokument dotrze do jak najszerszego grona odbiorców bo wszystkim nam powinno zależeć na tym aby nasza branża posiadała coraz więcej profesjonalistów, którym nie jest „wszystko jedno".

Pozdrawiam,
Mateusz Książek

1) Spłacaj dług techniczny tak szybko jak to możliwe

2) Zaadaptowanie zasad programowania funkcyjnego pozwoli zwiększyć jakość kodu

- funkcja zwracają zawsze ten sam wynik dla tego samego wejścia, bez względu na to gdzie i kiedy jest użyta
- powstaje wiele małych funkcji, które są łatwe w utrzymaniu i debugowaniu

3) Użytkownik, który używa Twojej aplikacji nie myśli tak jak Ty

Użytkownicy oprogramowania nie są zwykle tak samo bieli w obsłudze systemów i aplikacji jak ich Twórcy, należy obserwować i monitorować ich zachowania aby lepiej dostosować użyteczność produktu

4) Zadbaj o to aby cały zespół stosował ten sam „Code Style”

Używaj narzędzi do statycznej analizy kodu aby upewnić się, że wszystkie osoby, które uczestniczą w tworzeniu aplikacji posługiwały się tymi samymi zasadami kodowania.

5) Piękno drzemie w prostocie

Jeśli system jest poukładany z małych i prostych części to nie ważne jak bardzo jest duża i skomplikowana całość projektu. Małe obiekty z pojedynczą odpowiedzialnością to klucz do utrzymywalnego i ponadczasowego projektu, który będzie czysty, testowalny i prosty w rozwoju.

6) Dobrze się zastanów zanim zabierzesz się za refactor

- przed rozpoczęciem refaktoryzacji przyjrzyj się dobrze stanowi kodu i testów
- unikaj pokusy przepisania wszystkiego od nowa
- wiele mniejszych zmian jest lepsze od jednej masowej zmiany
- osobiste preferencje i ego nie powinny być powodem do refaktoryzacji (jeśli coś dobrze działa to po co to poprawiać?)

- nowa technologia nie jest powodem do refaktoryzacji
- ludzie zawsze popełniają błędy i nie ma żadnej gwarancji, że poprawki nie wprowadzą dodatkowych błędów

7) Strzeż się współdzielenia kodu

Zanim zdecydujesz się przepisać podobne części kodu z dwóch różnych warstw aplikacji do jednego miejsca zastanów się czy mieszanie kontekstów jest dobrym pomysłem.

8) Zawsze pozostaw moduł w lepszym stanie niż go zastałeś

Jeśli coś dodajesz do kodu to sprawdź czy nie można w nim przy okazji czegoś poprawić. Może warto zmienić nazwę zmiennej, która nie do końca jest jasna, albo podzielić funkcję na dwie mniejsze?

9) Musisz uwierzyć, że mogłeś popełnić błąd

Biblioteki, kompilatory i narzędzia do wytwarzania oprogramowania używane są przez sporą liczbę ludzi i prawdopodobieństwo, że coś w nich nie działa jest dużo mniejsze niż to, że Ty właśnie popełniłeś błąd. Zanim zaczniesz doszukiwać się problemów w zewnętrznych narzędziach najpierw dobrze się upewnij, że zrobiłeś wszystko jak należy.

10) Dobieraj narzędzia rozważnie

- projekt rozpoczynaj tylko z niezbędnym zestawem dodatkowych bibliotek
- izoluj zewnętrzne narzędzia od domeny biznesowej
- zadbaj o odpowiednią abstrakcję

11) Programuj w języku zrozumiałym dla Twojej domeny

- Nazywaj funkcje i zmienne tak aby były zrozumiałe dla wszystkich (nawet dla osób nietechnicznych)
- Ukrywaj szczegóły implementacyjne pod zrozumiałym kodem

12) Traktuj kod jako konstrukcję, która wymaga wysokiej jakości

Wielkie konstrukcje wykonane są przez wspaniałych konstruktorów, którzy są mistrzami swojego rzemiosła. Ciągle rozwijaj swoje umiejętności i doskonal projekty nad którymi pracujesz.

13) Styl i wygląd kodu ma znaczenie. Kod powinien wyglądać jak poezja

14) Proces Code Review pozwala utrzymać wysoką jakość kodu

- zapobiega błędom poprzez szybkie ich wykrycie
- wszyscy uczestnicy wiedzą co zmienia się w kodzie
- komentarze powinny być konstruktywne, a nie uszczypliwe
- pozwala na wymianę wiedzy

15) Kod powinien być samo dokumentujący się – rób małe funkcje skupione na jednym zadaniu

16) Jeśli dodajesz komentarze to muszą być zrozumiałe, proste, krótkie i klarowne

17) Komentuj tylko to czego kod nie może powiedzieć

Bardzo często kod się zmienia, ale komentarze pozostają te same i nie są aktualne. Z tego powodu programiści zaczynają je ignorować co prowadzi do tego, że można pominąć istotne informacje. *Komentarz powinien zawierać jakąś wartość dla czytającego*.

18) Aby zapewnić dobry rozwój swojej kariery musisz poświęcić na to trochę wolnego czasu

- czytaj książki i magazyny techniczne
- aby poznać lepiej technologię napisz trochę kodu

- staraj się mieć mentora, który będzie Cię uczyć
- subskrybuj blogi
- staraj się uczyć innych
- odwiedzaj konferencje

19) Staraj się jasno nazywać Twoje funkcje aby wyjaśniały to co mają robić

20) Nie zostawiaj sprawy związanej z procesem instalacji i deploymentu na koniec projektu. To równie ważna rzecz jak kod źródłowy

21) W aplikacjach można wydzielić dwa typy wyjątków: techniczne i wynikające z niezgodności logiki domenowej, które muszą być traktowane inaczej.

[*Więcej na ten temat można przeczytać w poście Obsługa wyjątków*]

22) Rozmyślna praktyka to umiejętność wychodzenia z własnej strefy komfortu

Ciągłe robienie czegoś w czym jest się dobrym nie pozwala na rozwój, warto czasem stawiać sobie cele aby wykonać zadania na mniej znanych płaszczyznach. Ciągle się ucz i obserwuj jak rozwój zmienia Ciebie i Twoje zachowania.

23) Specyficzna domena posiada wyspecjalizowany język do opisu szczegółów (DSL). Ukrywaj detale techniczne pod językiem domenowym.

24) Jeśli traktujesz swój projekt jak wieżę Jenga i boisz się wprowadzać zmiany to gdzieś jest błąd

Zawsze musisz dbać o „stan zdrowia” kodu aplikacji. Nie możesz się bać wprowadzać zmian w swoim kodzie, a jeśli uważasz, że najmniejsza zmiana może powodować problemy to powinieneś zainwestować czas w refaktoryzację.

25) Za każdym razem kiedy dokonujesz zmian w kodzie (komentarz, kod, logi) – zawsze zastanów się czy byłbyś gotów pokazać to światu bez obawy na krytykę

26) Jeśli ignorujesz błędy czy ostrzeżenie i wierzysz, że nic złego się nie wydarzy to podejmujesz ogromne ryzyko

27) Nie ucz się tylko języka, poznaj jego kulturę

Przykładowo, jeśli programujesz w języku funkcyjnym – poznaj co to lambda, kiedy programujesz w języku obiektowym zapoznaj się z zasadami OOP.

28) Nigdy nie pokazuj użytkownikowi raportu z wyjątku

29) Nie musisz rozumieć całej „magii”, która odbywa się w Twoim projekcie, jednak powinieneś ciągle poszerzać swoją wiedzę na temat środowiska w którym pracujesz

Kiedy wszytko działa dobrze, to jesteś zadowolony, jednak w sytuacji krytycznej możesz mieć duże problemy jeśli nie posiadasz wiedzy na temat szerszego kontekstu.

30) Nie powtarzaj się! (DRY)

- developer, który widzi powtórzenia i wie jak je usunąc tworzy czystszy kod
- każdy kawałek wiedzy musi mieć pojedyńczą jednoznaczą reprezentację w systemie
- *każdy proces wykonywany manualnie musi być automatyzowany*

31) Developerzy nie powinni mieć dostępu do kodu na środowisku produkcyjnym i stage

- wszystkie zmiany muszą być dokonywane na maszynie lokalnej
- **nigdy nie dokonuj zmian na żywym kodzie w środowisku prodykcyjnym**

32) Hermetyzuj zachowania, a nie tylko stan

Często w projektach spotyka się bardzo złą praktykę i zdarza się zauważyć klasy z samymi metodami get/set, które zarządzane są przez nadrzędne serwisy, gdzie znajduje się jedna funkcja, która „robi wszystko".

33) Liczby zmiennoprzecinkowe nie są liczbami rzeczywistymi.

Liczby zmiennoprzecinkowe to tylko przybliżone wartości liczb rzeczywistych. Musisz o tym pamiętam tworząc aplikacje operujące na takich wartościach (np. aplikacje finansowe) – aby zapewnić dobre wyniki warto używać zewnętrznych bibliotek.

34) Jeśli w Twojej obecnej pracy nie tworzysz aplikacji marzeń to zacznij rozwijać projekt open source – pozwoli to zrealizować Twoje programistyczne ambicje

35) Złota zasada projektowania API: unit testy API mogą być niewystarczające, potrzebne są również testy kodu, który korzysta z tego API

36) Nie budujmy mitu guru wokół osób bardziej doświadczonych

Mamy tendencję aby tworzyć mit „guru" dla osób doświadczonych jakoby posiadali umiejętność odpowiedzi na każde pytanie i mogli pomóc w każdym temacie. Doświadczeni programiści to nie magicy, tylko tacy sami ludzie jak my myślący w taki sam sposób jak my, posiadają tylko odrobinę więcej ciekawości i większej wiedzy.

Należy znieść takie sztuczne bariery. Nie budujmy mitu „guru", a ekspertów, którzy chętnie dzielą się swoją widzą i doświadczeniami, aby rozwijać kolejnych ekspertów.

37) Dłuższy czas spędzony w pracy i większy nakład energii nie musi sprawić, że będziesz lepszy.

Staraj się rozwijać i poznawać eco-system w którym pracujesz, a nie rozwiązuj tylko jak najszybciej zadań. Zamiast pracować w nadgodzinach przeczytaj książki, magazyny, poznawaj nowe narzędzia, wybierz się na konferencje branżowe.

38) Raport błędu powinien zawierać informacje jak go odtworzyć, jaki powinien być rezultat, a co się dzieje aktualnie

39) Pisz kod aby dodawał wartość, a nie dlatego, że Cię to bawi

Jeśli coś nie jest teraz potrzebne, to teraz tego nie twórz (YAGNI). Programiści nie są od tworzenia wymagań – nie dodawaj ekstra rzeczy bez uzgodnienia tego z klientem.

40) Zadbaj o to aby pierwsze kroki, które musi wykonać użytkownik w Twojej aplikacji były proste i intuicyjne, dobrze opisane i jasne

41) Jeśli w Twojej aplikacji spada szybkość działania to najpierw sprawdź czas komunikacji między procesami zanim zaczniesz optymalizować Twoje algorytmy

Rozwiązaniami może być lazy loading, optymalizacja interfejsów pomiędzy procesami, wielowątkowość, stosowanie cache.

42) Staraj się aby Twój proces budowania aplikacji był czysty

Jeśli pojawia się warning podczas budowania od razu staraj się go poprawić i usunąć. Ignorowanie ich sprawi, że po czasie będzie ich coraz więcej i może dojść do sytuacji, że pominiesz bardzo ważną informację.

43) Poznanie opcji linii poleceń może być dla Ciebie bardzo pomocne

Dzięki zrozumieniu linii poleceń poznasz co Twoje IDE wykonuje „pod spodem” i nie będzie to dla Ciebie żadna „czarna magia”. Po czasie będziesz gotowy do pisania własnych skryptów, które pozwolą Ci na automatyzację.

44) Dobry programista powinien być ciekawy nowych języków programowania w różnych paradygmatach

Spróbuj nauczyć się języków z różnych paradygmatów (programowanie obiektowe, funkcyjne, proceduralne). Dla przykładu przejście z Fortrana do C nie jest dużym wyzwaniem, jednak z C++ na Haskell to już jest wyczyn.

45) Poświęć trochę czasu aby lepiej poznać swoje IDE. Poznaj skróty klawiszowe i inne możliwości. Twoja praca po czasie stanie się jeszcze bardziej wydajna.

46) Zasoby są ograniczone, powinieneś znać złożoność Twoich algorytmów aby stworzyć optymalny system.

47) Dekomponuj zadania na mniejsze.

Z dużych tasków staraj się tworzyć zadania, które wykonasz w ciągu dwóch godzin. Pozwoli Ci to szybko reagować jeśli nie dajesz rady czegoś wykonać to możesz porzucić tę drogę i zacząć od nowa ten etap od zdefiniowania tego zadania od nowa. Nie commituj do repozytorium domysłów.

48) Używanie RDBMS pozwala na optymalizację zapytań i koncentrowaniu się na logice aplikacji zamiast na optymalizacji algorytmów.

49) Ucz się „języków obcych", czyli języków z innych dziedzin. Powinieneś znać język domeny rozmawiając z osobą, która ją reprezentuje

50) Musisz nauczyć się estymować projekty. Cele i zobowiązania powinny być oparte na estymacjach.

- Estymata – przybliżony czas realizacji zadania na podstawie doświadczenia programisty.

- Cel – biznesowe postanowienie, np. jak obsłużyć 400 użytkowników w jednym momencie.
- Zobowiązanie – obietnica dostarczenia funkcjonalności na pewnym poziomie jakości w konkretnym czasie.

51) Nie bój się kopiować jakiegoś fragmentu kodu do nowego projektu tylko po to aby go zrozumieć

Pracując nad istniejącym projektem czasem ciężko zweryfikować jak działa dany fragment kodu, a jego przetestowanie jest bardzo trudne. Warto wtedy skopiować ten fragment do nowego projektu w celu szybszej i łatwiejszej weryfikacji działania funkcji.

52) Postaraj się aby informacja o tym, że nie przechodzą testy docierała do Ciebie możliwie jak najszybciej

Bardzo często na CI ustawiany jest limit na minimalne pokrycie testami i uruchamiane są zestawy testowe, aby przyspieszyć informację o tym, że jakieś kryteria nie są spełnione możesz zastosować technikę XFD (extreme feedback device) – zainstalowanie w biurze urządzenia (np. czerwona lampka), które będzie informowało zespół o tym, że testy nie przeszły.

53) Linker nie jest magicznym narzędziem

Wielu programistów uważa, że etap podczas procesu kompilacji nazwany Linkerem to jakaś czarna magia. Tak naprawdę nie robi on nic skomplikowanego i jego jedynym zadaniem jest połączyć ze sobą cały kod z wszystkich plików, złączyć referencje symboli z ich definicjami i zapisać plik wykonywalny.

54) Większość rozwiązań tymczasowych zostaje w kodzie na zawsze

Często te rozwiązania są użyteczne, ale odbiegają od przyjętych standardów. Z czasem kiedy w projekcie przybywa takich „tymczasowych” rozwiązań staje się on coraz trudniejszy w utrzymaniu. Unikaj takich implementacji.

55) Dobre interfejsy są łatwe dla poprawnego ich użycia i trudne do złego użycia. Warto najpierw tworzyć oczekiwania, a dopiero później implementację.

56) Stan Twojego kodu i postęp prac nie jest widoczny bez odpowiednich mechanizmów. Zadbaj o to aby aspekty niewidoczne zostały ujawnione.

- pisanie unit testów pozwala na ujawnienie jakości Twojego kodu i sposobu działania aplikacji
- korzystanie z narzędzi do zarządzania zadaniami ujawnia postęp prac nad projektem
- inkrementacyjny rozwój projektu ujawnia postęp rozwoju

57) Podczas programowania współbieżnego wystrzegaj się współdzielenia pamięci

Programiści często obawiają się problemów związanych z programowaniem współbieżnym, jednak większość z nich wynika ze współdzielenia pamięci, dlatego warto tego unikać. Zamiast wątków używaj procesów i interfejsu transmisji wiadomości (MPI).

58) O każdej linii twojego kodu, który tworzysz myśl jak o wiadomości do kogoś w przyszłości. Spróbuj napisać kod tak czytelny aby w przyszłości ktoś się nim zachwycił.

59) Polimorfizm jest jednym z ważniejszych mechanizmów w programowaniu obiektowym, poprawne jego stosowanie pozwala zachować czystszy kod i ograniczyć ilość warunków.

60) Testerzy, którzy są zdeterminowani do tego aby odnaleźć błąd w twoim kodzie to twoi najlepsi przyjaciele. Nie oburzaj się kiedy raportują „każdą błahostkę".

61) Projekt powinien być budowany tylko raz

Wdrożenia na różne środowiska (developerskie, stage, produkcyjne itp.) powinny korzystać z raz zbudowanej paczki, a zmieniać powinny się jedynie ustawienia środowiskowe. Detale środowiska powinny zawierać się wewnątrz niego, a wszystkie informacje nt. środowiska powinny być wersjonowane osobno od kodu aby szybko zweryfikować zmiany.

62) Tylko kod mówi prawdę!

Dokumentacja często bywa nieaktualna lub różni się od implementacji. Komentarze też bywają błędne lub nieaktualne. Jeśli kod potrzebuje komentarzy to prawdopodobnie potrzebuje refactoru. To kod odpowiada za działanie programu i to on powinien mówić sam za siebie.

Zadbaj o to żeby Twój kod był czytelny. Używaj dobrych nazw, a struktura powinna być spójna z funkcjonalnością. Pisz testy, które tłumaczą zachowanie kodu.

63) Poświęć swój czas na naukę procesu budowania

Zainwestowany czas w dobre zaprojektowanie procesu budowania w przyszłości zaprocentuje. Proces budowania powinien być traktowanie na równi z powstającym kodem, a dzięki temu zyskasz lepszą jakość i mniejszą ilość błędów. **Programowanie nie jest skończone dopóki nie dostarczymy działającego programu**

64) Wypróbuj w zespole programowanie w parach

Praca w parach [i ich ciągła rotacja] może przynieść Twojemu zespołowi dużo pozytywnych rezultatów.

- Wymieniaj zadania między parami w celu dzielenia się wiedzą z wszystkimi członkami zespołu.
- Kiedy jedna para rozwiązuje problem, druga para może zweryfikować to rozwiązanie.
- Nowe osoby w zespole mogą się znacznie szybciej wdrożyć.
- Kiedy jedna osoba z pary musi wykonać jakieś zadanie niezwiązane z projektem to druga w tym czasie może utrzymać skupienie na konkretnym zadaniu.

65) Zamiast używać typów prostych lepiej używać typów domenowych (zdefiniowanych przy pomocy klas), które określają zachowanie danego bytu. Dobrym przykładem może być stosowanie Value Object

66) Staraj się zrozumieć użytkowników i przewidzieć jakie mogą popełnić błędy aby uodpornić system na zwracanie błędów – nie zawsze jest to potrzebne

Łatwo jest zwrócić użytkownikowi error kiedy wprowadzi złe dane, jednak czasem użytkownik może źle zinterpretować działanie systemu. Wartość w polu formularza wcale nie musi od razu zwracać informacji o błędzie, ponieważ można przygotować aplikację w taki sposób aby usunęła tę spację.

Z pomocą może przyjść system logowania zachowań użytkowników aby sprawdzić gdzie najczęściej popełniają błędy, dzięki temu lepiej zrozumiemy użytkowników.

67) Profesjonalny programista to osoba odpowiedzialna!

- odpowiedzialny za swój rozwój, o którego dba po godzinach swojej pracy
- bierze odpowiedzialność za swój kod, nie odda czegoś za co nie ma pewności i nie wie czy to zadziała
- profesjonaliści są graczami drużynowymi – nie dbają tylko o swoją część pracy, ale również o jakość pracy całego zespołu
- nie toleruje narastającej liczby tasków i bugów w „backlogu”
- nie lubi bałaganu, stosuje dobre praktyki i nigdy nie działa w pośpiechu (*„zrobię teraz na szybko gorzej, a później poprawię„*)

68) Trzymaj wszystko w systemach kontroli wersji (git, svn).

- kazdą zmianę logiki traktuj jako osobną operację i zacommituj ją
- każdy commit powinien posiadać krótki i wyjaśniający komentarz czego dotyczą zmiany
- Nie commituj kodu, który nie działa

69) Jeśli nie potrafisz rozwiązać problemu i siedzisz nad nim od dłuższego czasu zrób sobie przerwę.

Oderwanie się od kontekstu tego zadania pozwala ci sporzeć na nie z innej perspektywy po powrocie. Jest duże prawdopodobieństwo, że zrobisz to od razu. Pewnie przeżyłeś to już nie raz.

70) Programiści uwielbiają pisać kod, ale nie lubią go czytać.

Czytanie kodu bywa trudne, szczególnie cudzego. Następnym razem gdy będziesz czytać czyjś kod postaraj się z tego coś wyciągnąć i czegoś się nauczyć. Jeśli jest nieczytelny lub źle skonstruowany zadaj sobie pytanie dlaczego, odpowiedz na nie i nie popełniaj cudzych błędów.

71) Aplikacje tworzone są przez ludzi wraz z innymi ludźmi dla ludzi. Skupmy się na „dialogu” z innymi współtwórcami i odbiorcami aplikacji

72) Odkrywanie koła na nowo pozwala rozwinąć umiejętności programistyczne.

Tworząc własne narzędzie od podstaw mimo, że istnieje już gotowe pozwoli to poznać jak zostało to zrobione i zdobyć dodatkowe doświadczenie. Dodatkowo można zobaczyć jak to jest zrobione „pod spodem” i przekonać się z czym mierzyli się inni programiści.

73) Unikaj stosowania wzorca Singleton

- jeśli stosowany jest Singleton i wymagania odnośnie aplikacji się zmieniają trzeba wiele rzeczy zmieniać – dobry design jest odporny na zmiany
- Singleton sprawia, że pojawiają się problemy zależnościami i tworzy niepotrzebne połączenia między elementami systemu co utrudnia np. tworzenie unit testów
- Singleton często przechowuje ukryty stan co utrudnia testowanie
- wywołuje problemy w programowaniu wielowątkowym

74) Używaj narzędzi do analizy wydajności

Metryki oprogramowania pozwolą znaleźć zły kod, którego można wyeliminować zanim stanie się realnym problemem.

75) Kod musi być prosty!

W kodzie powinna występować minimalna liczba zmiennych, funkcji, deklaracji it… Wszystko co jest dodatkowe powinno zostać natychmiast usunięte.

76) Zasada jednej odpowiedzialności

Na temat zasady SRP napisano już tak wiele, że lepiej jak pojawią się tutaj po prostu odnośniki do innych artykułów:

- SOLID – część 1. Zasada Jednej Odpowiedzialności
- SOLID z easyGALib: Zasada pojedynczej odpowiedzialności
- Zasada pojedynczej odpowiedzialności

77) Zawsze staraj się mówić „TAK" do swojego rozmówcy

Zaczynając odpowiedź od „tak" pracujesz z twoimi kolegami, a nie przeciwko nim. Jeśli ktoś będzie chciał wprowadzić jakieś „dziwne" zmiany z twojej perspektywy w systemie, nad którym pracujecie to zamiast od razu go odrzucać zadaj pytanie „dlaczego" i postaraj się zrozumieć potrzeby.

78) Zatrzymaj się, confnij się i zautomatyzuj pracę, którą wciąż wykonujesz cyklicznie

- automatyzacja nie jest przeznaczona tylko do testów
- IDE nie zwalnia z automatyzowania zadań
- aby automatyzować nie jest potrzebna znajomość egzotycznego języka i narzędzi
- nie poddawaj się jeśli na pierwszy rzut oka wydaje ci się, że nie da się tego zautomatyzować
- nie zasłaniaj się brakiem czasu, przecież masz czas na to żeby wciąż robić to samo

79) Stosowanie statycznej analizy kodu pozwoli ci znaleźć potencjalne problemy w kodzie

80) Testy powinny sprawdzać wymagania, a nie implementację

Lepiej stosować zasadę czarnej skrzynki gdzie na podane wejście oczekujemy jakiegoś wyjścia, a nie zasadę białej skrzynki gdzie testujemy kolejno kroki wykonywane przez funkcję.

81) Testy powinny być precyzyjne i dokładne

Powinny być testowane konkretne rezultaty, np. jeśli testujemy funkcję sortującą to test powinien sprawdzić dokładny wynik, a nie liczbę zwracanych elementów. Podobnie jak w przypadku dodawania elementu do tablicy nie powinniśmy sprawdzać czy zwiększył się rozmiar tablicy lecz zweryfikować czy element faktycznie istnieje w tablicy.

82) Testuj podczas snu i weekendów

Marnujemy dużo mocy obliczeniowej nie wykorzystując naszych serwerów testujących w nocy i weekendy, a to świetna okazja aby sprawdzić chociażby wydajność naszych aplikacji i znaleźć potencjalne wycieki pamięci.

Podczas normalnej pracy nie mamy czasu odpalać długich testów przed każdym commitem, nie mówiąc już o sytuacji kiedy zbliża się termin oddania projektu. Dlatego uruchamiane są tylko unit testy, które są szybkie, a noc wydaje się idealnym momentem do dłuższych testów.

Poddawanie aplikacji długotrwałym testom może przynieść wiele korzyści, dlaczego by nie odpalać serwerów testujących w piątki o 18:00 tak aby działały do poniedziałku 6:00? To idealny moment na wykrycie błędów, które ukrywają się podczas pobieżnego testowania.

83) Testowanie jest inżynierskim rygorem wytwarzania oprogramowania

Programiści próbując wytłumaczyć czym jest ich praca często porównują się do innych zawodów inżynierskich takich jak budowniczych mostów. Jest to jednak nieprecyzyjne bo trudno porównać programowanie do innych zawodów.

Można jednak porównywać jeden istotny aspekt chociażby do budowania mostów – testowanie. Mamy zwyczaj pomijać testy aby szybciej wykonać zadanie. Podczas projektowania mostu menedżer nie ponagla inżynierów aby zrezygnowali z analizy strukturalnej bo są wąskie terminy.

84) Naucz się myśleć „stanowo”

Wyodrębnienie wyrażeń do znaczących metod to dobry początek, ale najważniejsze jest zrozumienie stanu maszyny. Stosuj Design by Contract Testuj twój kod aby zapewnić poprawny stan, bo jego błędy mogą spowodować utratę danych.

85) Co dwie głowy to nie jedna

Pracując w zespole nie ograniczaj się do wymiany wiedzy na poziomie zadawania pytań lub na ich odpowiadanie. Spróbuj kilkukrotnie programowania w parach aby wyrobić sobie zdanie na temat tej techniki. Jeśli jesteś nową osobą w zespole znajdź do współpracy kogoś kto posiada dużą wiedzę domenową.

86) Kod nigdy nie kłamie, ale może przeczyć sam sobie

Zdarza się, że istnieje błąd w aplikacji, który „przykrywany” jest przez inny błąd. Kiedy ktoś naprawi jeden z nich drugi może okazać się problemem. Nie ma prostych technik na unikanie takich sytuacji, ale warto zawsze podchodzić do programowania z „czystą głową” i próbować przewidzieć wszystkie możliwości, które wynikają z pisania kodu.

87) Programista rozwija się dzięki innym programistom

Jesteś częścią zespołu, ale pracujesz w izolacji rozwiązując problemy poprzez Twoją interpretację. Zacznij rozwijać swoje umiejętności dzięki wymianie wiedzy z innymi członkami zespołu.

88) Zaprzyjaźnij się z Unixowymi narzędziami

Nauka narzędzi unixowych spłaci się bardzo szybko, gdyż są uniwersalne, a do działania potrzebują one tylko linii komend. Większość tych narzędzi zostało opracowanych w czasach gdy zasoby komputerowe były małe, przez co przy dzisiejszej mocy obliczeniowej nadają się do przetwarzania dużych zbiorów danych.

89) Dobieraj odpowiednie algorytmy i struktury danych pod zadanie, które jest do zrealizowania

90) Zbyt obszerne logowanie może być równie bezużyteczne jak jego brak

91) Stosowanie zasady DRY pozwala na szybsze znalezienie i wyeliminowanie wąskiego gardła w aplikacji

92) Programiści i testerzy powinni ze sobą ściśle współpracować

Kiedy testerzy i programiści pracują ze sobą nad rozbudową aplikacji jakość oprogramowania jest dużo wyższa. Testerzy posiadają wiedzę gdzie mogą występować potencjalne błędy, a programiści posiadają wiedzę jak pisać dobry kod – dzięki temu można tworzyć automaty, które będą testować aplikację. Testerzy powinni przestać myśleć, że ich jedyną pracą jest znalezienie błędów w tworzonym kodzie przez programistów, wtedy programiści przestaną traktować testerów jako osoby, które tylko czekają na ich potknięcia.

93) Pisz kod w taki sposób jakbyś miał go rozwijać i utrzymywać przez resztę życia

94) Zamiast natywnych typów, używaj typów, które są bardziej związane z domeną

95) Pisz testy dla ludzi

Testy powinny nie tylko sprawdzać poprawność działania aplikacji, ale powinny stanowić też swego rodzaju dokumentację projektu. Jeśli nie wiesz czy Twoje testy są wystarczające czytelne – poproś osobę z zewnątrz aby na nie spojrzała i zweryfikuj czy jest w stanie dowiedzieć się co te testy sprawdzają.

96) Dobre programowanie polega na profesjonalnym podejściu i chęci napisania go najlepiej jak się da

- Unikaj „hackowania", które daje szybkie wyniki i pozwala myśleć, że to działa. Zadbaj o elegancki kod.
- Pisz kod, który jest łatwy do zrozumienia przez innych programistów, utrzymywalny (aby w przyszłości Ty albo ktoś inny mógł go szybko zmodyfikować), poprawny (upewnij się, że na pewno wszystko działa).
- Współpracuj z innymi programistami
- Staraj się pozostawić fragment kodu lepszym niż był początkowo
- Dbając o rozwój wciąż poznawaj nowe języki programowania, narzędzia i techniki. Ale stosuj je tylko gdy jest to potrzebne.

97) Twoi klienci na początku mogą nie widzieć czego rzeczywiście potrzebują

Za każdym razem gdy ustalasz z klientem zakres Twojej pracy upewniaj się co właściwie jest do zrobienia i czy dobrze rozumiesz czego chce klient. Często on sam początkowo nie wie czego potrzebuje – posiada szerszy obraz, ale to Ty powinieneś pomóc mu określić szczegóły.

//DEV:ENV

O PROGRAMOWANIU BEZ KACA

Jeśli interesują Cię podobne tematy i chciałbyś rozwijać swoje umiejętności związane z programowaniem to odwiedź serwis http://devenv.pl

Wspólną misją autorów DevEnv jest propagowanie dobrych praktyk oraz rozwiązań. Na co dzień autorzy są aktywnymi programistami z kilkuletnim doświadczeniem.

Masz pytania? Napisz!
kontakt@devenv.pl