

Pragmatyczny programista

Ta karta zawiera podsumowanie wskazówek i zadań opisanych w książce *Pragmatyczny programista*.

Ściaga

WSKAZÓWKI 1–23

1. **Należy dbać o swoje rzemiosło** 15
Jaki sens ma tworzenie oprogramowania przez całe życie, jeśli nie dbamy o jakość swojej pracy?
2. **Należy myśleć o tym, co się robi** 15
Musimy wyłączyć autopilota i przejąć kontrolę nad sterami. Powinniśmy stale krytykować i chwalić swoją pracę.
3. **Proponuj rozwiązania, zamiast posługiwać się kiepskimi wymówkami** 23
Zamiast wymówek należy proponować wyjścia z sytuacji. Nie możemy mówić, że coś jest niemożliwe — koncentrujemy się na tym, co **można** zrobić.
4. **Nie akceptuj żadnej wybitej szyby** 25
Należy naprawiać złe projekty, błędne decyzje i nieprzemyślany kod zaraz po odkryciu niedociągnięć.
5. **Bądź katalizatorem zmian** 28
Nie możemy zmuszać ludzi do zmian. Powinniśmy raczej pokazać im, że przyszłość może wyglądać nieporównanie lepiej, i pomóc w kreowaniu tej przyszłości.
6. **Pamiętaj o szerszym kontekście** 29
Szczegóły nie mogą nas pochłaniać do tego stopnia, że przestaniemy zwracać uwagę na zjawiska wokół nas.
7. **Jakość powinna być uwzględniona w wymaganiach** 31
Warto zaangażować użytkowników w proces definiowania rzeczywistych wymagań jakościowych.
8. **Regularnie inwestuj w swoje portfolio wiedzy** ... 34
Niech uczenie się będzie naszym nawykiem.
9. **Patrz krytycznym okiem na to, co czytasz i słyszysz** 37
Nie dajmy się zwieść producentom, szumowi medialnemu ani dogmatom. Warto analizować sytuację pod kątem własnych potrzeb i realizowanego projektu.
10. **Ważne jest nie tylko to, co mówimy, ale też to, jak to mówimy** 42
Nawet najlepsze pomysły są bezwartościowe, jeśli nie umiemy ich zakomunikować.
11. **Nie powtarzaj się (ang. *Don't Repeat Yourself* — DRY)** 47
Każdy element wiedzy musi mieć pojedynczą, jednoznaczную i rozstrzygającą reprezentację w systemie.
12. **Należy dbać o możliwość wielokrotnego stosowania kodu** 53
Jeśli wielokrotne stosowanie kodu jest łatwe, programiści chętnie korzystają z tej możliwości. Warto więc stworzyć odpowiednie środowisko.
13. **Należy eliminować wzajemny wpływ niepowiązanych elementów** 55
Powinniśmy projektować autonomiczne, niezależne komponenty. Każdy komponent musi realizować jedno precyzyjnie zdefiniowane zadanie.
14. **Nie istnieją ostateczne decyzje** 65
Żadna decyzja nie jest wykuwana w skale. Powinniśmy raczej traktować swoje decyzje jak rysunki na piasku — musimy być gotowi i otwarci na zmiany.
15. **Należy znajdować cel za pomocą pocisków smugowych** 68
Pociski smugowe umożliwiają wstrzeliwanie się w cel poprzez sprawdzanie różnych możliwości i badanie, jak blisko celu trafiają.
16. **Prototypy należy tworzyć z myślą o nauce** 74
Tworzenie prototypów to typowe doświadczenie poznawcze. O wartości prototypu decyduje nie utworzony kod, tylko wyciągnięte lekcje.
17. **Należy programować możliwie blisko dziedziny problemu** 78
Projekt i kod należy przygotowywać w języku użytkownika.
18. **Szacowanie pozwala unikać przykrych niespodzianek** 83
Przed przystąpieniem do pracy należy oszacować niezbędne nakłady. W ten sposób odkryjemy z wyprzedzeniem potencjalne problemy.
19. **Harmonogram i kod powinny powstawać iteracyjnie** 88
Warto wykorzystywać zbierane doświadczenie do stopniowego poprawiania harmonogramu.
20. **Wiedzę należy zapisywać zwykłym tekstem** 92
Zwykły tekst nigdy nie wyjdzie z użycia. Ułatwia dostosowywanie, diagnozowanie i testowanie systemu.
21. **Należy korzystać z potencjału poleceń powłoki** 98
Jeśli tylko graficzne interfejsy użytkownika na to pozwalają, powinniśmy stosować skrypty powłoki.
22. **Należy dobrze opanować jeden edytor** 100
Edytor powinien być przedłużeniem naszych rąk: wybrany edytor musi oferować możliwości konfiguracji, rozszerzania i programowania.
23. **Zawsze należy stosować system kontroli kodu źródłowego** 106
System kontroli kodu jest wehikułem czasu dla naszej pracy — wreszcie **możemy** cofnąć czas.

- 24. Należy rozwiązywać problemy, nie szukać winowajcy** 108
To, czy problem powstał z naszej winy, czy z winy kogoś innego, nie ma znaczenia — problem wciąż istnieje i wymaga rozwiązania.
- 25. Nie wolno panikować** 109
Weź głęboki oddech i MYŚL, co mogło spowodować błąd.
- 26. Wywołanie select działa** 114
Błędy rzadko występują w systemach operacyjnych, kompilatorach czy nawet zewnętrznych produktach lub bibliotekach. Najbardziej prawdopodobnym winowajcą jest nasz kod.
- 27. Nie należy niczego zakładać — należy to udowodnić** 115
Swoje założenia powinniśmy potwierdzać w docelowym środowisku, na rzeczywistych danych i w warunkach granicznych.
- 28. Należy opanować jeden język operujący na tekście** 117
Każdego dnia sporo czasu poświęcamy pracy z tekstem. Dlatego nasz komputer nie miałby przejąć części tych zadań?
- 29. Należy pisać kod, który pisze kod** 120
Generatory kodu zwiększają naszą produktywność i ułatwiają unikanie powielania.
- 30. Pisanie doskonałego oprogramowania jest niemożliwe** 125
Oprogramowanie z natury rzeczy nie może być doskonałe. Warto więc chronić kod i użytkowników przed nieuniknionymi błędami.
- 31. Należy projektować z uwzględnieniem kontraktów** 129
Powinniśmy używać kontraktów do dokumentowania i sprawdzania, czy nasz kod nie robi mniej ani więcej, niż powinien.
- 32. Awarie powinny następować możliwie wcześniej** 138
Martwy program zwykle powoduje mniej szkód niż upośledzony program.
- 33. Jeśli coś nie może się zdarzyć, należy użyć asercji do zagwarantowania, że rzeczywiście się nie zdarzy** 140
Asercje weryfikują nasze założenia. Warto ich używać do ochrony kodu przed nieznanym.
- 34. Wyjątki należy stosować dla wyjątkowych problemów** 145
Wyjątki są narażone na wszystkie negatywne zjawiska ograniczające czytelność i możliwości konserwacji kodu (określane mianem kodu spaghetti). Wyjątki należy stosować tylko w wyjątkowych sytuacjach.
- 35. Należy kończyć to, co się zaczyna** 147
Tam, gdzie to możliwe, funkcja lub obiekt, które przydzieliły jakiś zasób, powinny odpowiadać za jego zwolnienie.
- 36. Należy minimalizować związki pomiędzy modułami** 159
Powinniśmy unikać związków, pisząc „skromny” kod i konsekwentnie stosując prawo Demeter.
- 37. Należy konfigurować, nie integrować** 162
Wybór technologii na potrzeby aplikacji powinien mieć postać opcji konfiguracyjnych, nie trwałego skutku integracji czy inżynierii.
- 38. W kodzie należy umieszczać abstrakcje; szczegóły należy wyrażać w metadanych** ... 163
Powinniśmy programować ogólne przypadki, a konkretne scenariusze opisywać poza kompilowaną bazą kodu.
- 39. Warto analizować przepływ pracy, aby na tej podstawie poprawiać współbieżność** 168
Należy identyfikować współbieżność w przepływie pracy użytkownika.
- 40. Należy projektować przy użyciu usług** 171
Powinniśmy projektować systemy złożone z usług — niezależnych, współbieżnie działających obiektów ukrytych za dobrze zdefiniowanymi, spójnymi interfejsami.
- 41. Zawsze należy projektować z myślą o współbieżności** 173
Jeśli dopuścimy współbieżność, nasze interfejsy będą bardziej czytelne i mniej ograniczone zbędnymi założeniami.
- 42. Należy oddzielać widoki od modeli** 178
Elastyczność można uzyskać stosunkowo niewielkim kosztem — wystarczy projektować aplikacje złożone z modeli i widoków.
- 43. Należy koordynować przepływ pracy za pomocą tablic** 185
Warto używać tablic do koordynowania odrębnych, niezwiązanych bezpośrednio zdarzeń i agentów z zachowaniem niezależności i izolacji uczestników.
- 44. Nie należy programować przez koincydencję** ... 191
Powinniśmy polegać na tym, co niezawodne. Musimy unikać przypadkowej złożoności. Nie należy mylić szczęśliwego trafu z przemyślanym planem.
- 45. Należy szacować rzędy wielkości algorytmów** .. 197
Warto zastanowić się, ile czasu będą zajmowały poszczególne zadania, **przed** przystąpieniem do pisania odpowiedniego kodu.
- 46. Należy testować swoje szacunki** 198
Matematyczna analiza kodu to nie wszystko. Warto sprawdzić czas wykonywania kodu w jego docelowym środowisku.
- 47. Refaktoryzację należy przeprowadzać możliwie wcześniej i jak najczęściej** 202
Tak jak ogród wymaga pielęgnowania chwastów i sadzenia nowych roślin, tak kod nie może się obejść bez przepisywania, przeprojektowywania i zmiany architektury pewnych elementów. Musimy eliminować źródła problemów.

- 48. Należy projektować z myślą o testach208**
O testach warto pomyśleć jeszcze przed napisaniem pierwszego wiersza kodu.
- 49. Należy testować swoje oprogramowania; w przeciwnym razie zrobią to nasi użytkownicy213**
Testy muszą być bezlitosne. Nie możemy pozwolić, aby użytkownicy znajdowali błędy za nas.
- 50. Nie należy używać kreatorów do tworzenia kodu, którego nie rozumiemy215**
Kreatory mogą błyskawicznie utworzyć mnóstwo kodu. Warto dokładnie zapoznać się z **całym** tym kodem, zanim włączymy go do projektu.
- 51. Nie należy zbierać wymagań — należy je wydobywać z ukrycia218**
Wymagania rzadko są na wyciągnięcie ręki. Są raczej ukryte głęboko pod warstwami założeń, nieporozumień i decyzji politycznych.
- 52. Aby myśleć jak użytkownik, należy z nim popracować220**
W ten sposób można najlepiej i najszybciej zrozumieć, jak dany system **rzeczywiście** będzie używany.
- 53. Abstrakcje żyją dłużej niż szczegóły224**
Powinniśmy inwestować raczej w abstrakcję, nie w implementację. Tylko abstrakcje mogą przetrwać w burzliwych czasach zmian implementacji i technologii.
- 54. Należy stosować glosariusz projektu226**
Warto utworzyć i utrzymywać pojedyncze źródło wszystkich pojęć i terminów stosowanych na potrzeby projektu.
- 55. Nie należy wykraczać myślami poza schemat — należy raczej znaleźć ten schemat229**
W razie napotkania problemu niemożliwego do rozwiązania należy zidentyfikować **faktyczne** ograniczenia. Warto zadać sobie pytania: „Czy to rzeczywiście trzeba robić w ten sposób? Czy w ogóle trzeba to robić?”.
- 56. Należy słuchać uporczywych wątpliwości — nie wolno zaczynać pracy, dopóki nie jest się gotowym231**
Programista zbiera doświadczenie przez całe życie. Nie wolno nam ignorować uporczywych wątpliwości.
- 57. Niektóre rzeczy lepiej robić, niż o nich mówić233**
Nie powinniśmy wpadać w pułapkę specyfikacji. Kiedyś trzeba przystąpić do kodowania.
- 58. Nie możemy być niewolnikami formalnych metod235**
Nie powinniśmy ślepo wdrażać jakiejkolwiek techniki bez uprzedniego sprawdzenia jej przydatności w kontekście obecnych praktyk programistycznych i własnych możliwości.
- 59. Drogie narzędzia nie generują lepszych projektów237**
Należy z dystansem traktować szum wywoływany przez producentów, dogmaty obowiązujące w branży i aurę otaczającą metkę z ceną. Narzędzia powinniśmy oceniać wyłącznie według oferowanych możliwości.
- 60. Pracę należy organizować wokół implementowanych funkcji, nie zajmowanych stanowisk243**
Nie należy oddzielać projektantów od koderów czy testerów od specjalistów odpowiedzialnych za modelowanie danych. Zespół powinniśmy budować tak, jak budujemy swój kod.
- 61. Nie należy stosować ręcznych procedur247**
Skrypt powłoki lub plik wsadowy zawsze, niezależnie od okoliczności, wykonuje te same polecenia w tej samej kolejności.
- 62. Należy testować wcześniej. Należy testować często. Należy testować automatycznie253**
Testy wykonywane przy okazji każdej kompilacji są nieporównanie bardziej efektywne od planów testów na półce.
- 63. Kodowanie nie jest skończone, dopóki nie zostaną wykonane wszystkie testy253**
Kropka.
- 64. Do testowania testów należy stosować techniki sabotażu259**
Powinniśmy celowo umieszczać błędy w osobnej kopii kodu źródłowego, aby sprawdzać, czy stosowane testy wystarczą do ich wykrywania.
- 65. Należy testować pokrycie stanów, nie pokrycie kodu260**
Musimy identyfikować i testować najważniejsze stany programu. Samo testowanie wierszy kodu nie wystarczy.
- 66. Każdy błąd należy znajdować tylko raz261**
Pierwsze odnalezienie błędu przez testera powinno być jednocześnie ostatnim takim przypadkiem. Od tej pory błąd powinien być odnajdywany przez automatyczne testy.
- 67. Język polski należy traktować jako jeszcze jeden język programowania263**
Dokumenty powinniśmy pisać tak, jak piszemy kod — musimy przestrzegać zasady DRY, używać meta-danych, stosować model MVC, automatycznie generować treść itp.
- 68. Dokumentacja jest częścią produktu, nie dodatkiem263**
Dokumentacja tworzona w oderwaniu od kodu jest bardziej narażona na błędy i dezaktualizację.
- 69. Należy nieznacznie przekraczać oczekiwania użytkowników270**
Warto dobrze zrozumieć oczekiwania użytkowników, po czym dać im trochę więcej.
- 70. Podpisuj efekty swojej pracy272**
Rzemieślnicy zawsze byli dumni ze swojej pracy. My też powinniśmy być.

✓ **Nauka języków** s. 37.

Zmęczony językami C, C++ i Java? Spróbuj opanować CLOS, Dylan, Eiffel, Objective C, Prolog, Smalltalk lub TOM. Każdy z tych języków oferuje inne możliwości i nieco inaczej „smakuje”. Zrealizuj — choćby w domu — prosty projekt przy użyciu jednego lub kilku spośród tych języków.

✓ **Akrostych Wiedza** s. 39.

Warto wiedzieć, czego chcemy ich nauczyć.

Co ich zainteresuje w tym, co mamy do powiedzenia?

Na ile obeznani z tematem są nasi słuchacze?

Jak **dużo** szczegółów oczekują rozmówcy?

Kto powinien dysponować naszymi informacjami?

Jak zmotywować uczestników spotkania do wysłuchania naszych propozycji?

✓ **Jak utrzymywać ortogonalność** s. 53.

- Projektuj niezależne, dobrze zdefiniowane komponenty.
- Wystrzegaj się związków w kodzie.
- Unikaj danych globalnych.
- Stosuj refaktoryzację dla podobnych funkcji.

✓ **Co może być przedmiotem prototypu** s. 73.

- architektura;
- nowe funkcje w istniejącym systemie;
- struktura lub treść danych zewnętrznych;
- narzędzia lub komponenty zewnętrznych producentów;
- problemy związane z wydajnością;
- projekt interfejsu użytkownika.

✓ **Pytania o architekturę** s. 75.

- Czy dobrze zdefiniowano zakres odpowiedzialności?
- Czy odpowiednio zdefiniowano zasady współpracy?
- Czy zminimalizowano powiązania?
- Czy można zidentyfikować potencjalne powielenia?
- Czy definicje i ograniczenia interfejsu są możliwe do zaakceptowania?
- Czy moduły mają dostęp do potrzebnych danych, **kiedy** ich potrzebują?

✓ **Lista kontrolna diagnozowania** s. 115.

- Czy zgłoszony problem ma postać bezpośredniego wyniku jakiegoś błędu, czy tylko symptomu?
- Czy błąd **rzeczywiście** występuje w kompilatorze? Czy błąd występuje w systemie operacyjnym? A może problem tkwi w naszym kodzie?
- Gdybyśmy mieli szczegółowo wyjaśnić ten problem współpracownikowi, co byśmy powiedzieli?
- Jeśli podejrzany kod przechodzi swoje testy jednostkowe, czy te testy są dostatecznie kompletne? Co dzieje się, kiedy te same testy jednostkowe są wykonywane dla **tych samych** danych?
- Czy warunki, które spowodowały ten błąd, występują w jakimś innym miejscu systemu?

✓ **Prawo Demeter dla funkcji** s. 158.

- Metoda obiektu może wywoływać tylko metody należące:
 - do niej samej;
 - do parametrów przekazanych na jej wejściu;
 - utworzonych przez nią obiektów;
 - obiektów komponentu.

✓ **Jak programować celowo** s. 191.

- Zawsze musisz wiedzieć, co robisz.
- Nie koduj po omacku.
- Postępuj według planu.
- Opieraj się tylko na tym, co niezawodne.
- Dokumentuj swoje założenia.
- Testuj założenia równie wnikliwie jak kod.
- Nadawaj priorytety swoim wysiłkom.
- Nie bądź niewolnikiem historii.

✓ **Kiedy refaktoryzować** s. 201.

- Odkrywasz naruszenie zasady DRY.
- Stwierdzasz, że rozwiązania mogą być bardziej ortogonalne.
- Dysponujesz lepszą wiedzą.
- Zmieniły się wymagania.
- Musisz poprawić wydajność.

✓ **Przecinanie węzła gordyjskiego** s. 229.

Podczas rozwiązywania **nierozwiązywalnych** problemów zadaj sobie następujące pytania:

- Czy istnieje prostszy sposób?
- Czy rozwiązuję właściwy problem?
- **Dlaczego** ta kwestia jest problemem?
- Co sprawia, że jego rozwiązanie jest trudne?
- Czy nie ma innego rozwiązania?
- Czy w ogóle musimy to robić?

✓ **Aspekty testowania** s. 254.

- testy jednostkowe;
- testy integracyjne;
- sprawdzanie poprawności i weryfikacja;
- wyczerpywanie zasobów, błędy i odzyskiwanie;
- testy wydajnościowe;
- testy użyteczności;
- testy samych testów.