Clean Code considered harmful

Powinniśmy przestać zalecać niedoświadczonym programistom czytanie Clean Code

Prawie dokładnie sześć lat temu na łamach „Programisty" pojawił się mój artykuł „Jak clean jest Clean Code", w którym polemizowałem z niektórymi postulatami wysuniętymi przez Roberta C. Martina. Upłynęło jednak trochę czasu, napisałem kilkadziesiąt, a może i kilkaset tysięcy linii kodu i moja niechęć wobec skrajnego – w niektórych przypadkach – podejścia Wujka Boba do pisania kodu jeszcze bardziej się wzmogła. Ściślej, stoję obecnie na stanowisku, że duża część proponowanych przez niego reguł jest nie tyle kontrowersyjna albo irytująco restrykcyjna, ale wręcz szkodliwa i że nie powinniśmy ich stosować ani tym bardziej proponować czeladnikom programistycznego fachu. Co ciekawe, nie jestem wcale z moją opinią odosobniony.

CLEAN CODE?

„Clean Code" jest terminem zaproponowanym przez Roberta C. Martina, zwanego w środowisku programistycznym jako „Uncle Bob", czyli „Wujek Bob". W 2009 roku wydał on książkę zawierającą szereg zasad, których – według niego – należy przestrzegać, aby projektowaną przez programistów architekturę oraz pisany przez nich kod można było określić mianem „czysty". Działo się to półtorej dekady temu, więc wróćmy na chwilę do tamtych czasów.

Pierwszą dekadę lat 2000 wspominam jako czas, w którym większość znanych mi programistów – być może poza tymi obracającymi się w środowiskach akademickich – szła swoimi ścieżkami, samodzielnie szukając drogi do dobrze napisanego kodu. Przyczyną takiego stanu rzeczy nie były jednak wcale ich złe intencje.

Przede wszystkim Internet nie był jeszcze tak szeroko rozpowszechnionym medium wymiany informacji, jakim jest dzisiaj. Popularny stackoverflow.com powstał dopiero w 2008 roku, a jego rozwój do stanu, w którym można było znaleźć na nim dobre 90% odpowiedzi na programistyczne pytania, trwał długie lata. W Sieci istniały wprawdzie programistyczne fora, ale trochę brakowało takich, które zrzeszałyby wystarczająco dużo programistów (aby zapewnić odpowiednią różnorodność opinii) oraz w których uczestniczyliby również uznani profesjonaliści – również zza granicy.

Sam korzystałem głównie z grup dyskusyjnych. Stanowiły one coś na kształt współczesnych forów, ale były one scentralizowane i przez to mające potencjał dotarcia do szerszej grupy zainteresowanych. Od czasu do czasu prowadziliśmy tam dyskusje na temat jakości kodu, pojawiały się pierwsze odpowiedniki współczesnych *code reviews*, ale podejrzewam, że mało który z uczestników umiałby jasno zdefiniować, co to znaczy, że kod jest dobrej jakości.

Z pierwszej przyczyny wynika pośrednio druga: mówimy bowiem o czasach, w których nowych programistów pojawiało się coraz więcej, ale z drugiej strony tych naprawdę doświadczonych była garstka. W praktyce więc – szczególnie wobec braku dobrej dokumentacji i łatwo dostępnych materiałów szkoleniowych – początkujący adepci programistycznej sztuki skazani byli na uczenie się na własnych błędach, również w środowiskach produkcyjnych.

Oprócz tego zaryzykuję twierdzenie, że wiele z programistycznych projektów było relatywnie niewielkich: programy powstawały jak grzyby po deszczu tylko po to, aby kompletnie zniknąć z rynku zaledwie rok później. W konsekwencji stosunkowo rzadko pojawiała się potrzeba długotrwałej konserwacji i rozwoju napisanego oprogramowania, co z kolei ograniczało prawdopodobieństwo, aby źle napisany kod ugryzł programistów w… no, w każdym razie ugryzł. Trudno więc dziwić się, że mało kto przejmował się takimi tematami, jak standaryzacja podejścia do projektowania architektury czy też wypracowanie norm kontroli jakości kodu źródłowego.

Wreszcie trzecią przyczyną takiego, a nie innego stanu rzeczy mógł być fakt, iż rozwój inżynierii oprogramowania jest ściśle związany z rozwojem języków programowania. Aby nadać nieco kontekstu, możemy posłużyć się świetnym diagramem Érica Lévéneza [1] prezentującym wycinek historii rozwoju większych z nich. W roku 2000 mamy więc pierwszą wersję C#, PHP 4.0, Pythona 1.6 i Delphi 6. Dla porównania nieco ponad 10 lat później dysponowaliśmy już C# 4.0, Javą 7, Pythonem 2.6.3 (i równolegle 3.2), Delphi 2010, PHP 5.3.6, no i na horyzoncie pojawił się wreszcie słynny C++11 (dla niezorientowanych, opublikowana w 2011 roku pierwsza duża aktualizacja C++, który praktycznie pozostawał bez żadnych zmian od 2003 r.).

Piszę o tym dlatego, że rozwój języków programowania otwierał nowe możliwości w zakresie projektowania architektury. Mało kto pewnie pamięta, że wyrażenia lambda, które wstrzyknęły do C# elementy programowania funkcyjnego, pojawiły się dopiero w wersji 3.0 tego języka (ściślej, w 2006 roku). Możemy podejrzewać że pierwsze ugruntowane wzorce korzystania z lambd pojawiły się przynajmniej rok, może dwa lata później.

I tu trzeba Wujkowi Bobowi oddać sprawiedliwość, że poprzez wydanie swojej książki oraz przeprowadzenie szeregu konferencji na

całym świecie wprowadził programistów w nową epokę, bo zaczął na temat jakości kodu mówić głośno. Od tego momentu sam fakt, że program działał, nie był już wystarczającym kryterium sukcesu. Teraz trzeba było jeszcze pokazać, że jego kod źródłowy jest czytelny, a cały projekt łatwy w konserwacji i rozwoju. Albo – co chyba zdarzało się znacznie częściej – przekonująco wytłumaczyć się, dlaczego tak nie jest.

Należy uczciwie powiedzieć, że w opisanym powyżej zakresie jego działanie z pewnością odbiło się pozytywnie na społeczności programistów. Również i jego książka „Clean Code" osiągnęła z pewnością pozytywny efekt w postaci ogólnej poprawy jakości pisanego przez programistów kodu. Niestety jednak książka ta nie zestarzała się równie dobrze, jak inne znane klasyki literatury programistycznej. Stanowi ona bowiem mieszankę zasad, z którymi możemy się w miarę zgodzić, z takimi zasadami, których stosowanie może odbić się niekorzystnie nie tylko na czytelności kodu źródłowego, ale w niektórych przypadkach również na jego wydajności.

W numerze 8/2019 (87) „Programisty" napisałem artykuł o nazwie „Jak bardzo clean jest Clean Code". Już wtedy polemizowałem z niektórymi pomysłami Wujka Boba, ale nie zdecydowałem się stanowczo im przeciwstawić. Jednak sześć lat później i dziesiątki, a może i setki tysięcy linijek kodu później moje zdanie uległo zmianie: teraz uważam, że powinniśmy przestać stosować się do reguł „Clean Code" i zacząć szukać nowego sposobu na odróżnianie słabego kodu od dobrego.

Powiedzmy sobie jasno: nie uważam „Clean Code" za zło wcielone – wiele z reguł proponowanych przez Wujka Boba ogólnie ma sens i jest (w miarę) zbieżne ze współczesnymi standardami programowania. Ale są one niestety wymieszane z takimi regułami, które są nieprzydatne, powodują wprowadzenie do kodu niepotrzebnego zamętu lub są wręcz szkodliwe, a takiego zmieszanego zestawu pretendującego do miana „Czystego Kodu" nie mogę z czystym sumieniem polecić początkującym, bo ci – z definicji – nie są jeszcze w stanie odróżnić jednych od drugich.

W niniejszym artykule zaprezentuję więc moje argumenty przeciw regułom „Clean Code". Nie lubię jednak poprzestawać na krytyce, dlatego postaram się później zaproponować szkic nowych zasad czystego kodu, z których być może wraz z kolektywem wszystkich zainteresowanych programistów uda nam się wypracować nowy i lepszy standard kodowania.

WYSOKI POZIOM

Zacznijmy od kilku ogólnych obserwacji dotyczących „Clean Code", które rzucają cień na dzieło Roberta C. Martina.

Nie wiem, czy zauważyliście, ale Wujek Bob zastosował ciekawy zabieg psychologiczny. Wielu naukowców pisząc swoje prace naukowe, ogranicza się do stwierdzeń „nowa, proponowana metoda" albo na przykład „propozycja w trakcie ewaluacji". Tymczasem Robert C. Martin nazwał zbiór proponowanych przez siebie reguł po prostu „Clean Code", czyli „Czysty Kod". Tym samym wytworzył ciekawy konstrukt: jeżeli stosujesz się do jego zasad, twój kod jest czysty. Jeżeli zaś nie – wtedy nie możesz już o nim w ten sposób powiedzieć. Warto na przykład zwrócić uwagę na hasło znajdujące się na samym początku książki w jej angielskojęzycznej wersji:

> *Writing clean code is what you must do in order to call yourself a professional. There is no reasonable excuse for doing anything less than your best.*

Powiedzmy sobie szczerze: opisany wcześniej ciąg myślowy nie jest prawdziwy. „Czysty Kod" to nie to samo co czysty kod i śmiem twierdzić, że jest wiele sytuacji, w których oba terminy – ten w cudzysłowach i ten bez – stoją wręcz ze sobą w sprzeczności. Ale o tym za chwilę.

Wujek Bob okazuje się też być mistrzem figur retorycznych. W jego książce można znaleźć więcej wypowiedzi, które z jednej strony kreują obraz „Clean Code" jako jedynego słusznego podejścia do programowania, ale z drugiej strony zostawiają wystarczająco dużo miejsca na „plausible deniability", czyli możliwość wykazania, że wcale o to autorowi nie chodziło. W polskiej edycji znajdziemy na przykład takie stwierdzenie (tekst sformatowany przeze mnie):

> *W istocie wiele zaleceń zamieszczonych w tej książce jest kontrowersyjnych. Prawdopodobnie nie zgodzisz się ze wszystkimi. Możesz je kwestionować. Świetnie. Nie uważamy się za ostateczny autorytet. Z drugiej strony, zalecenia zamieszczone w tej książce dotyczą technik, których poznanie zajęło nam dużo czasu. Nauczyliśmy się ich przez dekady doświadczenia oraz w wyniku wielokrotnych prób i błędów.* ***Tak więc niezależnie od tego, czy się zgadzasz, czy nie, byłoby błędem, gdybyś nie przyjął i nie docenił naszego punktu widzenia****.*

Dodajmy tu jasno, że w żadnej mierze nie próbuję wygrać dysputy poprzez zdyskredytowanie konkurenta. Moje zarzuty są merytoryczne i zaraz do nich przejdziemy; chciałbym jednak zachęcić do podchodzenia do wypowiedzi Roberta C. Martina z odpowiednim dystansem, ponieważ jego zdolności retoryczne znajdują się przynajmniej na poziomie polityka. I nie jest to tylko moje zdanie – różnego rodzaju manipulacjom w jego wypowiedziach jest poświęcony dosyć duży materiał w serwisie YouTube [2].

Drugim ogólnym spostrzeżeniem jest to, że „Clean Code" w postaci zaproponowanej przez Wujka Boba próbuje przekonać wszystkich do tego, że czysty kod obiektowy to taki kod, który jest *czysto obiektowy*. Mam tu na myśli rozwiązywanie wszystkich problemów architekturalno-algorytmicznych poprzez zastosowanie mechanik polimorfizmu. Prostym przykładem może być choćby postulat, aby rezygnować ze stosowania instrukcji `if` oraz `switch` na rzecz dziedziczenia i abstrakcji.

Realia pokazują jednak, że już dawno przestaliśmy programować obiektowo. Teraz programujemy, mniej więcej, obiektowo-funkcjonalno-deklaratywnie, a pewnie i ten opis też nie jest wcale wyczerpujący. Poza tym programowanie czysto obiektowe możemy rozpatrywać tylko w kategorii akademickiej analizy, bo pociąga ono za sobą zbyt wiele wad, aby traktować je na poważnie (choćby marna wydajność kodu w porównaniu do alternatywnych rozwiązań).

Po trzecie, Robert C. Martin proponuje jednolity zestaw bardzo ścisłych i konkretnych reguł, ignorując jednocześnie kwestię kontekstu, w jakim dany kod jest pisany.

Bądźmy realistami: zupełnie inaczej będzie wyglądał czysty kod programu, którego kluczowym aspektem jest jego niezawodność (na przykład w systemach life-critical), a inaczej, jeżeli zależy nam przede

wszystkim na jego wysokiej wydajności (na przykład w systemach przetwarzających duże ilości danych). Skonstruowanie jednego zbioru zasad, które bez żadnych modyfikacji możemy zastosować w obu przypadkach, jest najzwyczajniej w świecie nierealne.

To tyle, jeśli chodzi o ogólne spostrzeżenia, teraz przejdźmy do konkretów.

CASE STUDY

Zanim przejdziemy do reguł jako takich, weźmy na początku na warsztat jeden z konkretnych przykładów zaprezentowanych przez Wujka Boba jako „czysty kod".

W rozdziale na temat klas prezentuje on kod źródłowy algorytmu generującego liczby pierwsze, umieszczony przez Donalda Knutha w jego książce o nazwie „Literate Programming". W tym artykule skupię się tylko na jednej metodzie, która realizuje generowanie liczb pierwszych (reszta kodu odpowiada za paginację wyników). Oryginalny kod wygląda więc następująco:

Listing 1. Fragment oryginalnego kodu (listing 10.5 w książce Czysty Kod)

```
package literatePrimes;

public class PrintPrimes {
    public static void main(String[] args) {
        final int M = 1000;
        final int RR = 50;
        final int CC = 4;
        final int WW = 10;
        final int ORDMAX = 30;
        int P[] = new int[M + 1];
        int PAGENUMBER;
        int PAGEOFFSET;
        int ROWOFFSET;
        int C;
        int J;
        int K;
        boolean JPRIME;
        int ORD;
        int SQUARE;
        int N;
        int MULT[] = new int[ORDMAX + 1];
        J = 1;
        K = 1;
        P[1] = 2;
        ORD = 2;
        SQUARE = 9;
        while (K < M) {
            do {
                J = J + 2;
                if (J == SQUARE) {
                    ORD = ORD + 1;
                    SQUARE = P[ORD] * P[ORD];
                    MULT[ORD - 1] = J;
                }
                N = 2;
                JPRIME = true;
                while (N < ORD && JPRIME) {
                    while (MULT[N] < J)
                        MULT[N] = MULT[N] + P[N] + P[N];
                    if (MULT[N] == J)
                        JPRIME = false;
                    N = N + 1;
                }
            } while (!JPRIME);
        K = K + 1;
        P[K] = J;
    }
}
```

Nie ulega dyskusji, że powyższy kod zdecydowanie nie należy do najczytelniejszych. Poza niestosowaniem konwencji nazewnictwa (ang. *camel case*) dla zmiennych, te mają krótkie i kompletnie nieczytelne nazwy, zaś cały kod jest jedną wielką ścianą tekstu.

Zobaczmy zatem, jaki jest pomysł Wujka Boba na uczynienie powyższego kodu czytelnym (zob. Listing 2).

Prawdę mówiąc, nie wiem nawet, gdzie zacząć.

Przede wszystkim obdarzyłem Roberta C. Martina pewnym kredytem zaufania i podjąłem próbę zrozumienia, co dzieje się w tym kodzie. Na pierwszy rzut oka nie implementuje on żadnego znanego mi algorytmu generowania liczb pierwszych, więc zacząłem go analizować linijka po linijce (lub metoda po metodzie, co zresztą prawie na jedno wychodzi). Po kilkunastu minutach jednak poddałem się – nawet szerokie, opisowe nazwy typu `smallestOddNthMultipleNotLessThanCandidate` nie przekazały mi żadnych pożytecznych informacji.

Sam algorytm zrozumiałem dopiero wówczas, gdy przeniosłem oryginalny kod (ten z książki Knutha) do pustego projektu, przerobiłem na C# (to tylko dla mojej wygody) i zacząłem refaktoryzować samodzielnie. I dopiero uruchomienie programu, a potem późniejsze przedebugowanie go połączone z inspekcją wartości tablic i zmiennych pozwoliło mi dojść do tego, w jaki sposób generowane są liczby pierwsze (na marginesie, zdesperowany, zdążyłem już zadać pytanie na mathematics.stackexchange.com, ale udało mi się jednak samodzielnie rozpracować problem jeszcze przed otrzymaniem pierwszej odpowiedzi).

Oczywiście jest całkiem możliwe, że to ja po prostu jestem zbyt mało doświadczony, żeby zrozumieć matematyczny algorytm. Decyzję pozostawiam czytelnikowi: jeżeli nie miał wcześniej do czynienia z cytowanym kodem, to zachęcam, aby tak jak ja przeprowadził jego statyczną analizę i spróbował wydedukować działanie algorytmu samodzielnie.

Swoją drogą, taka forma „inżynierii wstecznej" jest fantastycznym miernikiem jakości kodu.

Projektanci interfejsów graficznych znają na pewno metodę o nazwie „hallway usability testing", która polega na złapaniu losowej osoby z korytarza (stąd „hallway"), posadzeniu jej przed projektowanym interfejsem użytkownika, a potem poleceniu rozwiązania za pomocą tegoż interfejsu jakiegoś problemu (na przykład „posortuj wszystkie elementy rosnąco"). Jeżeli ów korytarzowy pechowiec nie będzie w stanie zrealizować takiego polecenia, oznacza to, że zaprojektowany interfejs jest do kitu i trzeba go poprawić.

Mechanizm ten można z powodzeniem zastosować również w przypadku kodu źródłowego – oczywiście z małą modyfikacją, bo tym razem zamiast losowej osoby z korytarza musimy „złowić" losowego programistę. Pozostała część jednak działa w sposób analogiczny.

Cóż, w moim przypadku kod Wujka Boba egzaminu nie zdał, a śmiem twierdzić, że z moim, niemałym mimo wszystko doświadczeniem programistycznym, miał nawet pewne fory.

Roberta C. Martina można jednak wciąż bronić – w końcu taki „hallway usability testing" stanowi mimo wszystko miarę bardzo subiektywną, zależną od złowionej osoby. Zobaczmy więc, jakie *konkretne* zarzuty mogę postawić powyższemu rozwiązaniu (kolejność nie gra roli).

PAIR PROGRAMMING – CZY TO DZIAŁA? · STREFA CVE

INDEX: 285358

www.programistamag.pl

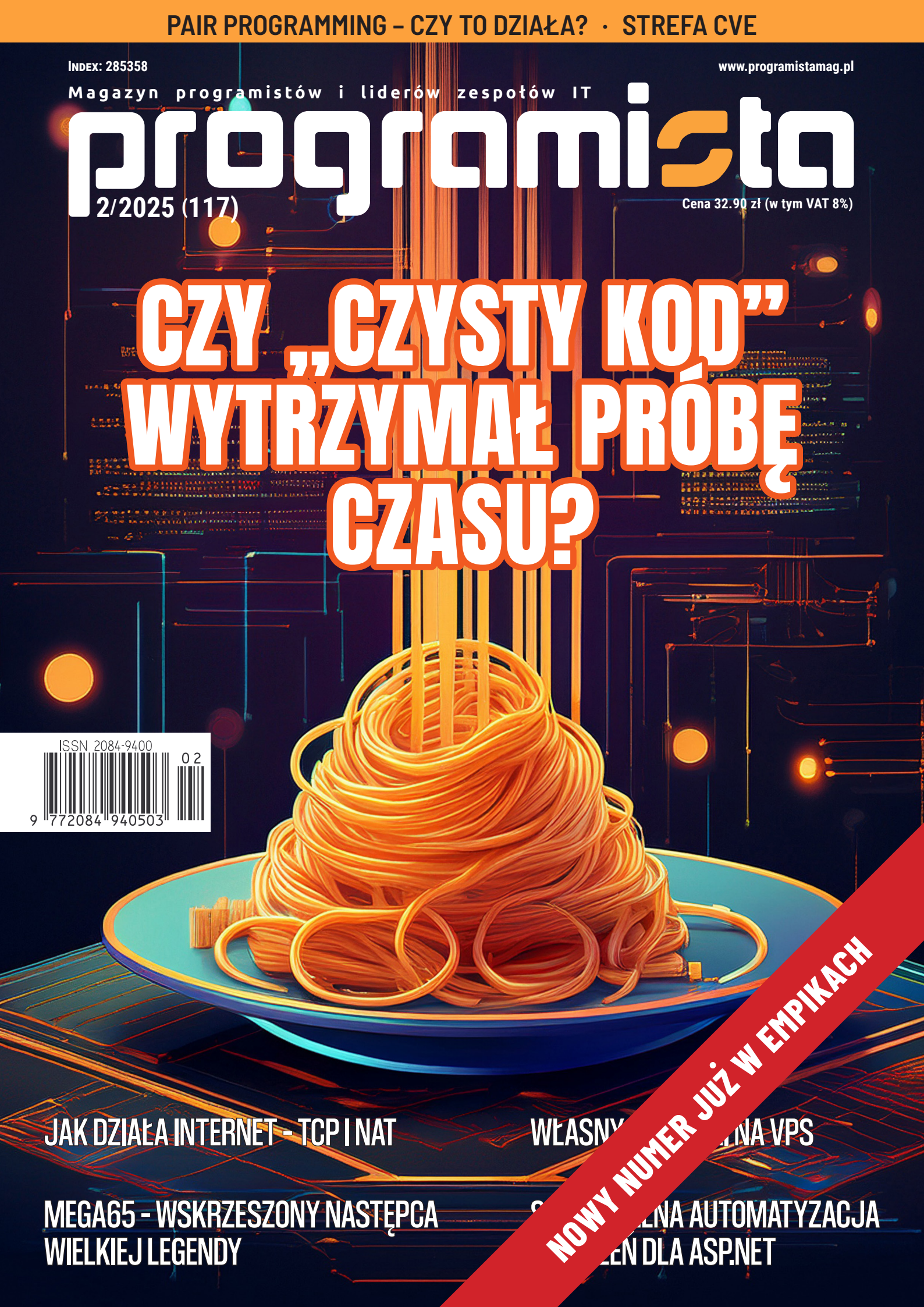
Magazyn programistów i liderów zespołów IT

programista

2/2025 (117)

Cena 32.90 zł (w tym VAT 8%)

CZY „CZYSTY KOD” WYTRZYMAŁ PRÓBĘ CZASU?



ISSN 2084-9400

02

9 772084 940503

JAK DZIAŁA INTERNET - TCP I NAT

WŁASNY [illegible] NA VPS

MEGA65 - WSKRZESZONY NASTĘPCA WIELKIEJ LEGENDY

[illegible]ENA AUTOMATYZACJA [illegible]EŃ DLA ASP.NET

NOWY NUMER JUŻ W EMPIKACH