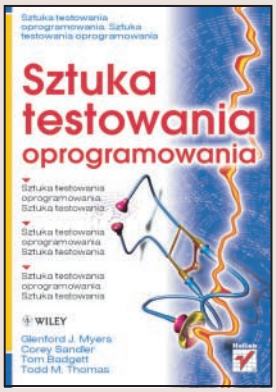
Sztuka testowania oprogramowania



Autor: Glenford J. Myers, Corey Sandler, Tom Badgett, Todd M. Thomas
Tłumaczenie: Andrzej Grażyński
ISBN: 83-7361-894-5
Tytuł oryginału: The Art of Software Testing, Second Edition
Format: B5, stron: 272

Testowanie to ostatni i niestety czasem pomijany element procesu tworzenia oprogramowania. Tymczasem ten właśnie etap powinien być niezwykle znaczącą częścią projektu. Znaczenie testowania dostrzegano już w początkowym okresie dynamicznego rozwoju technologii tworzenia oprogramowania, jednak nadal trudno jest znaleźć jasny i czytelny zbiór reguł testowania i metodyki, w oparciu o które proces ten należy przeprowadzać. Testy oprogramowania często przeprowadzane są przez jego twórców lub osoby przypadkowe, co zdecydowanie nie zdaje egzaminu.

„Sztuka testowania oprogramowania" to książka traktująca wyłącznie o testowaniu oprogramowania. Przedstawia zasady testowania kodu źródłowego, pojedynczych modułów programu oraz całej aplikacji. Zawiera cenne wskazówki dla testerów dotyczące przygotowywania przypadków testowych i metodologii testowania. Autorzy opisali w niej również metodykę testowania ekstremalnego i sposoby testowania aplikacji internetowych.

- Podstawowe zasady testowania programów
- Inspekcja kodu źródłowego
- Przypadki testowe
- Testowanie pojedynczych modułów aplikacji
- Testowanie funkcjonalne, systemowe, akceptacyjne i instalacyjne
- Usuwanie błędów
- Reguły testowania ekstremalnego
- Testowanie aplikacji internetowych

Zadbaj o to, aby tworzone przez Ciebie programy były pozbawione błędów.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa **7**

Wprowadzenie **9**

1. Samoocena zdolności testera **13**

2. Psychologiczne i ekonomiczne aspekty testowania programów **19**
Psychologia testowania 20
Ekonomika testowania 23
Test „czarnej skrzynki" 24
Test „białej skrzynki" 26
Zasady testowania programów 29
Podsumowanie 36

3. Inspekcja programów, wędrówka po kodzie źródłowym i przegląd kodu **39**
Inspekcje i wędrówki po kodzie 40
Inspekcja kodu 42
Lista kontrolna błędów programistycznych na użytek inspekcji kodu 45
Błędy w odwołaniach do danych 45
Błędy w deklaracjach danych 48
Błędy obliczeniowe 50
Błędy porównywania 51
Błędy przepływu sterowania 53
Błędy interfejsu 55
Błędy wejścia-wyjścia 56
Inne błędy 57
Wędrówki po kodzie 60
Kontrola przy biurku 61

Wzajemna ocena 62
Podsumowanie 63

4. Projektowanie przypadków testowych 65
Przypadki testowe dla testów „białej skrzynki" 67
Testowanie pokrycia kodu 67
Podział na klasy równoważności 75
Przykład 79
Analiza wartości granicznych 83
Grafy przyczynowo-skutkowe 91
Zgadywanie błędów 113
Strategia 115

5. Testowanie modułów (jednostek) 117
Projektowanie przypadków testowych 118
Testowanie przyrostowe 132
Testowanie zstępujące a testowanie wstępujące 138
Testowanie zstępujące 138
Testowanie wstępujące 145
Porównanie 147
Przeprowadzanie testów 149

6. Testowanie wysokopoziomowe 151
Testowanie funkcjonalne 157
Testowanie systemowe 158
Testowanie możliwości 161
Testowanie objętościowe 161
Testowanie przeciążeń 162
Testowanie użyteczności 164
Testowanie ochrony danych 166
Testowanie efektywności 166
Testowanie pamięci 167
Testowanie konfiguracji 167
Testowanie zgodności i konwersji 168
Testowanie procedury instalacyjnej 168
Testowanie niezawodności 168
Testowanie funkcji ratunkowych 170
Testowanie możliwości obsługi 171
Testowanie dokumentacji 171
Testowanie procedur 172
Przeprowadzanie testów 172

Testowanie akceptacyjne 173
Testowanie instalacyjne 174
Planowanie i kontrolowanie testów 175
Kryteria zakończenia testu 178
Niezależne agencje testujące 185

7. Debugowanie 187
Debugowanie „na siłę" 189
Debugowanie przez indukcję 191
Debugowanie przez dedukcję 195
Debugowanie przez nawracanie 200
Debugowanie przez testowanie 201
Reguły debugowania 201
Reguły lokalizowania błędów 202
Techniki poprawiania błędów 203
Analiza błędów 205

8. Testowanie ekstremalne 209
Podstawy programowania ekstremalnego 210
Testowanie ekstremalne — koncepcja 216
Ekstremalne testowanie jednostek 216
Testowanie akceptacyjne 218
Testowanie ekstremalne — praktyka 220
Projektowanie przypadków testowych 221
Aplikacja i jej sterownik testowy 224
Podsumowanie 225

9. Testowanie aplikacji internetowych 227
Podstawowa architektura aplikacji e-commerce 229
Wyzwania związane z testowaniem 231
Strategie testowania 235
Testowanie warstwy prezentacji 237
Testowanie warstwy biznesowej 241
Testowanie warstwy danych 244

A Przykładowa aplikacja do testowania ekstremalnego 249

B Liczby pierwsze mniejsze niż 1000 255

Słownik 257

Skorowidz 263

3

Inspekcja programów, wędrówka po kodzie źródłowym i przegląd kodu

Przez wiele lat powszechne było wśród programistów przekonanie, że program przeznaczony jest wyłącznie do wykonywania przez komputer, nie do czytania przez człowieka, a więc testowanie programu nie może odbywać się inaczej, jak tylko przez uruchamianie go na komputerze. Mniej więcej we wczesnych latach 70. ubiegłego wieku programiści zaczęli jednak stopniowo doceniać także znaczenie „bezkomputerowego" czytania kodu jako integralnej części wszechstronnego procesu testowania.

Co prawda nie wszyscy programiści zwykli studiować kody źródłowe w poszukiwaniu błędów, sama jednak koncepcja takiego czytania zyskała sobie ogólnie przychylne przyjęcie. Jej praktyczna realizacja uwarunkowana jest kilkoma czynnikami, między innymi rozmiarem i złożonością programu, rygoryzmem harmonogramów, liczebnością zespołu testującego, kwalifikacjami jego członków itp.

Przed przystąpieniem do omawiania tradycyjnych, „maszynowych" technik testowania poświęcimy nieco uwagi testowaniu bezkomputerowemu. Okazuje się ono zadziwiająco efektywne pod względem wykrywania błędów — efektywne tak dalece, że przy realizacji

dowolnego projektu informatycznego należy jego zastosowanie przynajmniej rozważyć; szczególnie polecane jest ono bezpośrednio po zakończeniu tworzenia kodu programu, jeszcze przed przystąpieniem do jego komputerowego testowania, choć może okazać się użyteczne także we wcześniejszych stadiach (etapach) kodowania (rozwinięcie tej kwestii wykraczałoby jednak poza ramy niniejszej książki).

Na początek jednak istotne spostrzeżenie. Jako że testowanie programu przez człowieka odbywa się zwykle za pomocą metod mniej formalnych niż matematyczna weryfikacja kodu (przez komputer), naturalne stają się obawy, czy rezultaty tak prostego i mało rygorystycznego postępowania mogą mieć jakąkolwiek wartość praktyczną. Otóż mogą, i choć takie nieformalne postępowanie nie może stanowić głównej metody testowania, to jednak może testowanie to uczynić bardziej produktywnym i wiarygodnym — z dwóch powodów.

Po pierwsze, im wcześniej wykryte zostaną błędy, tym mniejszy będzie koszt ich naprawienia i mniejsze prawdopodobieństwo pomyłki przy naprawianiu. Po drugie, z chwilą rozpoczęcia testowania maszynowego programiści ulegają swoistej presji mentalnej, ukierunkowanej na szybkie osiągnięcie sukcesu („jak najszybciej uporać się z tym nieznośnym błędem!"), co zwykle zwiększa zagrożenie wprowadzania nowych błędów przy usuwaniu istniejących. Mniej formalne postępowanie „bezkomputerowe" jest od tego typu presji zazwyczaj wolne.

Inspekcje i wędrówki po kodzie

Wędrówki po kodzie (*walkthrougs*) oraz jego inspekcje (*inspections*) to dwie główne metody bezkomputerowego testowania. Ponieważ są one pod wieloma względami podobne do siebie, zajmiemy się najpierw tymi podobieństwami, odkładając na nieco później omówienie różnic między nimi.

Zarówno wędrówki po kodzie, jak i inspekcja kodu polegają na wizualnym studiowaniu kodu przez zespół testowy. Celem tego jest jedynie wykrywanie błędów, nie ich poprawianie — wszak mamy do czynienia z testowaniem, nie z debugowaniem. Zespół powinien pracować w klimacie porozumienia, w dążeniu do wspólnego celu, powinien też być odpowiednio przygotowany do działania. Powodzenie (albo niepowodzenie) wspólnego wysiłku uwarunkowane jest wieloma czynnikami, których większość omówiliśmy w rozdziale 2.

Wędrówka po kodzie sprowadza się do jego uważnego przeglądu, dokonywanego przez zespół (optymalnie) trzech lub czterech testerów. Tylko jeden z nich jest autorem (lub współautorem) programu, pozostali nie są związani z jego tworzeniem — zgodnie z regułą nr 2 z poprzedniego rozdziału. Jest to sytuacja znacznie korzystniejsza (z perspektywy powodzenia testów) w porównaniu z samodzielnym studiowaniem kodu przez jego autora (tzw. „kontrola na biurku" — *desk-checking*).

Inną zaletą wędrówek po kodzie, przekładającą się bezpośrednio na niższe koszty poprawiania błędów, jest możliwość precyzyjnego umiejscawiania wykrywanych błędów w kodzie źródłowym. Zazwyczaj wykrywa się wówczas całe grupy powiązanych ze sobą błędów, które później mogą być w sposób grupowy eliminowane. Nie ma tej zalety testowanie maszynowe, pozwalające jedynie na obserwację *symptomów* błędów (program „zawiesza się" bądź na wydruku pojawiają się bezsensowne wartości) i umożliwiające wykrywanie tylko jednego błędu na raz.

Inspekcja i wędrówki kodu umożliwiają wykrycie średnio 30 – 70 procent błędów związanych z logiką programu i kodowaniem, są jednak zwykle nieefektywne w odniesieniu do poważniejszych błędów projektowych, jak błędy w procesie analizy wymagań (*requirement analysis*). Nie oznacza to oczywiście, że można w ten sposób wykryć do 70% *wszystkich* błędów, bo — jak wyjaśnialiśmy w rozdziale 2. — liczba wszystkich błędów tkwiących w programie zawsze pozostaje niewiadoma; owe 70% odnosi się raczej do *wszystkich błędów wykrytych* w całym procesie testowania.

Swoją drogą warto zachować pewien krytycyzm (sceptycyzm?) co do przedstawionej statystyki, bezkomputerowe testowanie umożliwia bowiem wykrywanie raczej „prostych" błędów — te bardziej subtelne, zakamuflowane ujawniają się przeważnie tylko w testowaniu maszynowym. To poniekąd prawda, jak jednak wskazuje doświadczenie wielu testerów, testowanie bezkomputerowe okazuje się skuteczniejsze od maszynowego w stosunku do *pewnych szczególnych rodzajów* błędów, powinno więc być traktowane jako jego wartościowe uzupełnienie.

Chociaż testowanie bezkomputerowe okazuje się użyteczne w odniesieniu do nowo tworzonych programów, może być równie (jeśli nie bardziej) praktyczne w stosunku do modyfikacji programów istniejących. Dokonywanie zmian w istniejących (i przetestowanych)

programach jest czynnością znacznie bardziej podatną na błędy (w sensie średniej ilości błędów przypadających na jedną dodawaną lub zmienianą instrukcję) niż tworzenie nowych programów. Testowanie bezkomputerowe okazuje się w tym kontekście wartościowym uzupełnieniem (maszynowego) testowania regresywnego.

Inspekcja kodu

Inspekcja kodu to zestaw procedur i technik wykrywania błędów w ramach czytania kodu przez grupę ludzi. Większość dyskusji związanych z inspekcją kodu koncentruje się wokół procedur, formularzy do wypełnienia itp.; po krótkim wprowadzeniu w ogólne zagadnienia inspekcji zajmiemy się więc szczegółowo poszczególnymi technikami wykrywania błędów.

Zespół dokonujący inspekcji składa się zazwyczaj z czterech osób. Jedną z nich jest moderator, z założenia doświadczony programista, niezwiązany jednak z analizowanym (testowanym) programem i niewtajemniczony w jego szczegóły. Do podstawowych obowiązków moderatora należą między innymi:

- organizacja sesji inspekcyjnych i dystrybucja niezbędnych materiałów,
- kierowanie sesjami inspekcyjnymi,
- rejestrowanie wszystkich stwierdzonych błędów,
- zapewnienie, że wykrywane błędy są konsekwentnie poprawiane.

Moderator jest więc kimś w rodzaju inżyniera kontroli jakości. Pozostali członkowie zespołu to (zwykle) programista (autor programu), projektant programu (nie mylić z programistą) oraz specjalista od testów.

Listing programu i jego specyfikacja projektowa dostarczane są przez moderatora pozostałym członkom zespołu na kilka dni przed rozpoczęciem sesji. Członkowie zespołu powinni zapoznać się dokładnie ze wspomnianymi materiałami, zaś sam przebieg sesji sprowadza się głównie do dwojakiego rodzaju czynności:

1. Programista — jako narrator — odczytuje kolejne instrukcje programu, zaznajamiając z jego logiką pozostałych członków zespołu. Ci ostatni mogą w tym czasie zadawać pytania,

powinni też na bieżąco śledzić tok narracji w celu zauważenia ewentualnych niejasności w logice programu (i wykrycia w ten sposób kolejnego błędu). Faktem jest jednak, iż to głównie programista wykrywa najwięcej błędów, w rezultacie (jedynie) głośnego odczytywania treści programu wobec uważnego audytorium — czyli w wyniku zastosowania techniki zgoła nieskomplikowanej, acz niewątpliwie użytecznej.

2. Program analizowany jest pod kątem zgodności z tzw. listą kontrolną (*checklist*) powszechnie popełnianych błędów, tworzoną i aktualizowaną w następstwie kolejnych sesji testowych. Przykład takiej listy przedstawimy w dalszej części rozdziału.

Moderator jest odpowiedzialny za rzeczowość dyskusji oraz za to, by jej uczestnicy koncentrowali swą uwagę na wykrywaniu błędów, nie zaś na ich poprawianiu (poprawianiem błędów zajmuje się programista po zakończeniu sesji inspekcyjnej).

Jeśli efektem sesji jest wykrycie dużej liczby błędów bądź wykryte zostają błędy wymagające znaczącej ingerencji w kod programu, moderator może zarządzić ponowną inspekcję po poprawieniu tych błędów przez programistę. W każdym przypadku wykryte błędy są analizowane, klasyfikowane i stanowią podstawę do aktualizacji listy kontrolnej, której ulepszona treść powinna przyczynić się do większej efektywności następnych sesji inspekcyjnych.

Jak wcześniej stwierdziliśmy, celem inspekcji kodu jest zwykle wykrywanie błędów, nie ich poprawianie; w przypadku jednakże błędów oczywistych, wymagających nieznacznych zmian w kodzie, dokonuje się niekiedy ich „kolektywnego" poprawiania na bieżąco w czasie sesji. Ubocznym efektem takiego postępowania jest zwrócenie szczególnej uwagi zespołu na pewne wybrane obszary projektu — w czasie dyskusji nad sposobem poprawienia drobnego błędu ktoś może mianowicie zauważyć inny błąd, związany z tym samym aspektem projektu, co w efekcie już po kilku minutach skoncentrowanej dyskusji może doprowadzić do wykrycia innego, tym razem poważnego błędu.

Czas i miejsce sesji inspekcyjnych powinny być tak wybierane, by sesje te mogły odbywać się bez jakichkolwiek zakłóceń. Optymalny czas sesji zawiera się w granicach 90 – 120 minut; ze względu na dość duży wysiłek intelektualny uczestników sesji nadmierne jej przedłużanie z konieczności skutkować musi spadkiem produktywności. Średnia produktywność sesji inspekcyjnej oscyluje wokół 150 analizowanych instrukcji w ciągu godziny, tak więc inspekcja dużych programów powinna być podzielona między kilka sesji, w ramach których analizuje się poszczególne moduły programu (lub powiązane grupy modułów).

Nie należy zapominać, iż warunkiem powodzenia sesji inspekcyjnej jest odpowiednie nastawienie jej uczestników oraz odpowiednia atmosfera w zespole. Jeśli programista postrzega inspekcję swego programu jako atak na swe kwalifikacje, przyjmuje postawę obronną, z oczywistych względów stawiającą pod znakiem zapytania efektywność samej inspekcji. Tymczasem powinien on zdawać sobie sprawę, że ostatecznym celem inspekcji jest przecież ulepszanie jego dzieła drogą eliminowania tkwiących w nim z konieczności błędów. Z tego też względu zalecane jest, by wyniki sesji inspekcyjnych pozostawały *poufną* sprawą zespołów testowych; gdyby z wyników tych chcieli zrobić jakiś użytek np. menedżerowie, ryzykują oni pojawienie się wspomnianych postaw defensywnych.

Mimo iż głównym celem inspekcji programu jest wykrywanie błędów tkwiących w programach, to zwykle ma ona także kilka pożytecznych efektów ubocznych. Programista będący członkiem zespołu inspekcyjnego może na przykład na bieżąco weryfikować poszczególne elementy swego stylu programowania, swój repertuar technik programistycznych, swe preferencje wyboru algorytmów itp. Inni członkowie zespołu mogą wzbogacać swe doświadczenia w zakresie możliwych rodzajów błędów, wykrywanych w *rzeczywistych* programach. Co więcej, sesje inspekcyjne są znakomitym środkiem do wczesnego wykrywania obszarów kodu szczególnie podatnych na błędy (*vide* reguła nr 9 testowania programów w poprzednim rozdziale), dzięki czemu obszarom tym poświęcić można szczególną uwagę podczas testowania maszynowego.

Lista kontrolna błędów programistycznych na użytek inspekcji kodu

Istotną częścią procesu inspekcji jest lista kontrolna najczęściej popełnianych błędów programistycznych (*error checklist*). Niestety, treść większości takich list bądź to koncentruje się na zagadnieniach stylistycznych zamiast merytorycznych („czy komentarze są treściwe i adekwatne?", „czy bloki kodu, warianty instrukcji `if` i zagnieżdżane ciała pętli są poprawnie akapitowane?"), bądź też same opisy błędów formułowane są dość mgliście („czy kod należycie spełnia wymagania projektowe?"). Prezentowana poniżej przykładowa lista kontrolna jest efektem wieloletnich studiów nad najczęściej popełnianymi błędami programistycznymi. Jest ona w dużym stopniu niezależna od konkretnego języka programowania — wymienione na niej błędy wystąpić mogą niemal w każdym języku. Pouczającym dla Czytelników ćwiczeniem może być wzbogacenie jej o błędy charakterystyczne dla niektórych tylko języków programowania oraz o błędy wykrywane w ramach własnych sesji inspekcyjnych.

Błędy w odwołaniach do danych

1. Czy zmienna, do której następuje odwołanie, jest zainicjowana, czy też ma przypadkową wartość? Niezainicjowane zmienne są prawdopodobnie najczęstszą przyczyną niewłaściwego funkcjonowania programów, a przyczyny braku inicjowania mogą być rozmaite. Dla każdej danej (zmiennej, elementu tablicy, pola w strukturze), do której w danym miejscu kodu następuje odwołanie, należy podjąć próbę przeprowadzenia nieformalnego „dowodu", iż w momencie tego odwołania ma ona określoną wartość.
2. Czy w odwołaniach do elementów tablic wartości indeksów mieszczą się w zadeklarowanych granicach?
3. Czy w odwołaniach do elementów tablicy indeksy mają wartości całkowite? Niecałkowita wartość indeksu nie jest błędem w niektórych językach programowania, mimo to zawsze wygląda podejrzanie, a jej zamierzone stosowanie nie jest bezpieczną praktyką.

4. Czy obszar pamięci, wskazywany przez wskaźnik, nadal pozostaje przydzielony? Zwolnienie obszaru pamięci wskazywanego przez jakiś wskaźnik czyni ten ostatni tzw. wiszącym wskaźnikiem (*dangling pointer*); błąd wiszącego wskaźnika występuje zawsze wtedy, gdy sam wskaźnik jest obiektem o dłuższym czasie życia niż obiekt przezeń wskazywany. Przykładem takiej sytuacji jest przypisanie zmiennej globalnej (lub parametrowi wyjściowemu procedury) wskazania na tymczasową zmienną lokalną procedury; po zakończeniu realizacji takiej procedury jej zmienne lokalne przestają istnieć, lecz wspomniane wskaźniki zachowują swą wartość, stając się wskaźnikami wiszącymi. Podobnie jak w punkcie 1., dla każdego wskaźnika, do którego następuje odwołanie, należy spróbować nieformalnie dowieść, że w momencie tego odwołania nie jest on wskaźnikiem wiszącym.
5. Czy to samo miejsce w pamięci zajmowane jest przez dwie lub większą liczbę zmiennych o odmiennych atrybutach? Współdzielenie pamięci przez różne zmienne ma miejsce w przypadku zastosowania np. dyrektywy `EQUIVALENCE` w języku FORTRAN czy klauzuli `REDEFINES` w języku COBOL[1]. Jeśli na przykład w fortranowskim programie dyrektywa `EQUIVALENCE` utożsamia zmienną rzeczywistą `A` ze zmienną całkowitą `B`, to przypisanie jakiejkolwiek wartości zmiennej `A` powoduje automatycznie, że zmienna `B` zyskuje wartość przypadkową, zależną od konkretnej implementacji (odwołanie do zmiennej `B` spowoduje zinterpretowanie jako liczby całkowitej wzorca bitowego odzwierciedlającego liczbę rzeczywistą).
6. Czy atrybut wartości przypisywanej zmiennej zgodny jest z deklarowanym atrybutem tej zmiennej? Niezgodność taka może wystąpić np. w języku C++ lub COBOL w sytuacji, gdy zmiennej strukturalnej przypisywany jest odczytany z pliku rekord o strukturze niezgodnej ze strukturą tej zmiennej[2].

---

[1] W Turbo Pascalu i Delphi zadanie to spełnia klauzula `absolute` — *przyp. tłum.*

[2] W Turbo Pascalu i Delphi może się tak zdarzyć w przypadku wczytywania danych z plików amorficznych za pomocą procedury `BlockRead` — *przyp. tłum.*

7. Czy w programie nie występują jawne lub wtórne błędy adresowania? Jeżeli na przykład w konkretnej implementacji jakiegoś języka programowania logiczne jednostki alokacji pamięci są mniejsze od jednostek fizycznie adresowalnych przez maszynę, błędy adresowania są bardzo prawdopodobne. Przykładem takiej sytuacji jest alokowanie *łańcuchów bitowych* niewyrównanych na granicach bajtu w maszynie o adresowaniu bajtowym; jeśli w programie obliczany jest adres łańcucha bitowego i następnie adres ten przypisywany jest wskaźnikowi, to w przypadku niewyrównania wspomnianego łańcucha na granicy bajtu odwołanie się do rzeczonego wskaźnika da efekty inne od oczekiwanych[3]. Podobna sytuacja wystąpić może w przypadku przekazywania niewyrównanego łańcucha bitowego jako parametru procedury.

8. Czy obszar pamięci wskazywany przez wskaźnik (lub referencję) ma właściwe atrybuty? W języku C++ niezgodność taka może wystąpić, gdy wskaźnik i wskazywany przez niego obszar różnią się od siebie deklaracją[4].

---

[3] Ani w Turbo Pascalu, ani w Delphi nie jest możliwe deklarowanie ani alokowanie danych niewyrównanych do granicy bajtu — *przyp. tłum.*

[4] Oto przykład takiej sytuacji w Turbo Pascalu:

```
Type
   PRecord1 = ^TRecord1;
   TRecord1 = record
                NrId: Longint;
                Nazwisko: String[30];
                Imię: string[15]
              end;
   Precord2 = ^TRecord2;
   TRecord2 = record
                NrRef: integer;
                Stanowisko: String[35];
                Lokal: string[8]
              end;

   var
     X1 : Precord1;
     X2 : Precord2;
     P: Pointer;
...
New(X1);
P := X1;
...
X2 := P;
Writeln(X2^.NrRef);
```

— *przyp. tłum.*

9. Czy dana, do której odwołuje się kilka procedur, jest „widziana" (pod względem struktury) w identyczny sposób przez każdą z tych procedur?
10. Czy w indeksowaniu elementów tablic lub poszczególnych znaków łańcucha nie popełniono błędu „pomyłki o jeden"?[5]
11. W językach obiektowych — czy w implementacji klasy pochodnej uwzględniono wszystkie wymogi wynikające z dziedziczenia klas?[6]

Błędy w deklaracjach danych

1. Czy wszystkie używane zmienne zadeklarowane zostały w sposób jawny? W niektórych językach (np. w FORTRAN-ie) nie ma wymogu deklarowania zmiennych, lecz brak jawnej deklaracji zmiennej może być przyczyną poważnych kłopotów. Jeśli na przykład parametr formalny (nazwijmy go `A`) podprogramu fortranowskiego miał być w zamierzeniu tablicą, lecz pominięto jego deklarację (np. w postaci dyrektywy `DIMENSION A(10)`), wszelkie odwołania do elementów tej tablicy (np. `C = A(I)`) zinterpretowane zostaną jako odwołania do *funkcji* reprezentowanej przez parametr `A`. W efekcie podjęta zostanie próba wykonania… zawartości tablicy (przekazanej jako parametr aktualny), postrzeganej jako kod programu. W językach programowania dopuszczających zagnieżdżanie deklaracji procedur brak deklaracji zmiennej używanej

[5] Błąd „pomyłki o jeden" (*off-by-one error*) polega na „rozminięciu się" o 1 faktycznego wyniku z prawidłowym. Przykładem takiego błędu jest niepoprawna odpowiedź na pytanie, ile słupów trzeba postawić na dziesięciokilometrowej linii energetycznej, jeśli odległość między słupami wynosi 50 metrów? Udzielana często nieprzemyślana odpowiedź (200) różni się od poprawnej (201) o jeden — *przyp. tłum.*

[6] W C++ i Delphi jednym z takich wymogów jest zaimplementowanie wszystkich metod abstrakcyjnych klasy macierzystej (chyba że nie chcemy tworzyć instancji obiektu). Również często spotykanym w Delphi uchybieniem takiemu wymogowi jest pomylenie klauzul `virtual` i `override` — *przyp. tłum.*

w ciele procedury powoduje zinterpretowanie jej jako zmiennej globalnej, nie zawsze zgodnie z intencją programisty[7].

2. Jeśli atrybuty zmiennej nie zostały jawnie wymienione w jej deklaracji, to czy domyślne ustalenia atrybutów zostały właściwie zrozumiane przez programistę? Błędne założenia co do domyślnych atrybutów w języku Java są częstym źródłem niespodzianek.
3. Czy inicjowanie zmiennej w jej deklaracji jest prawidłowe? W wielu językach inicjowanie deklarowanych tablic i łańcuchów jest skomplikowane i podatne na błędy.
4. Czy każdej zmiennej przypisano właściwy typ i rozmiar?
5. Czy sposób inicjowania zmiennej jest spójny z rodzajem zajmowanej przez nią pamięci? Przykładowo w języku FORTRAN, jeżeli jakaś zmienna ma być inicjowana przy każdym wejściu do podprogramu, to inicjowanie

---

[7] Oto przykład w Turbo Pascalu — składniowo poprawny, lecz w procedurze `B` zapomniano zadeklarować zmienną lokalną `i`:

```
procedure A;
const
  LL = 10;
type
  LLArr = array [ 1..LL ] of integer;
var
  i: integer;
  M: LLArr;
  N: array[1..LL] of LLArr;

  procedure B (var X: array of integer; var K:integer);
  begin {B}
    i := Low(X);
    while (i <= High(X)) and (X[i] = 0) do
      inc(i);
    ....
  end; {B}
begin {A}
  ...
  for i := Low(M) to High(M) do
  begin
    B(N[i], M[i])
  end;
  ....
end {A}
```

— *przyp. tłum.*

to powinno odbywać się w instrukcji przypisania, a nie w dyrektywie DATA[8].

6. Czy w programie występują zmienne o podobnych nazwach (np. VOLT i VOLTS)? Nie musi to być błąd, lecz sytuacja taka może być wynikiem pomyłki programisty[9].

Błędy obliczeniowe

1. Czy w programie prowadzone są jakieś obliczenia na zmiennych nienumerycznych?
2. Czy w programie występuje mieszanie typów zmiennych w wyrażeniach, na przykład mieszanie wartości całkowitych i zmiennopozycyjnych? Jeżeli tak, należy upewnić się, że określone przez semantykę języka reguły konwersji typów danych zostały właściwie zrozumiane. Oto fragment kodu w języku Java prezentujący zaokrąglanie niezgodne z oczekiwanym:

```
int x = 1;
int y = 2;
int z = 0;
z = x/y;
System.out.println("z = " + z);
```

   W wyniku wykonania powyższego kodu wypisana zostanie wartość 0.
3. Czy w jakimś obliczeniu biorą udział dwie zmienne tego samego typu, lecz różnej długości?

---

[8] W Turbo Pascalu odpowiednikiem fortranowskiej dyrektywy DATA są tzw. stałe typowane, na przykład:

```
const
  MonthLenghts : array[1..12] of byte =
(31,28,31,30,31,30,31,31,30,31,30,31);
```

— *przyp. tłum.*

[9] Podobieństwo nazw może mieć inną jeszcze konsekwencję: otóż wiele kompilatorów uwzględnia jedynie określoną liczbę początkowych znaków identyfikatora; przykładowo niektóre wersje języka FORTRAN rozróżniają tylko *sześć* pierwszych znaków nazwy, tak więc np. identyfikatory IPOWERX1, IPOWERX2, IPOWERY1 i IPOWERY2 orz IPOWER oznaczają wówczas *tę samą* zmienną! W Turbo Pascalu znacząca część identyfikatora także jest ograniczona — choć znacznie dłuższa, bo kompilator rozróżnia 63 początkowe znaki — *przyp. tłum.*

4. Czy w którymś obliczeniu typ zmiennej wynikowej ma „węższy" zakres niż typ przypisywanego wyniku?
5. Czy w którymś obliczeniu może wystąpić nadmiar lub niedomiar? Czy jest możliwe, by mimo poprawnej wartości ostatecznego wyniku wyniki pośrednie przekraczały zakres dozwolony w danym języku?
6. Czy jest możliwe, by w którejś operacji dzielenia dzielnik miał wartość zerową?
7. Jeżeli arytmetyka zmiennopozycyjna komputera zrealizowana jest w układzie dwójkowym, to czy program wolny jest od lawinowo kumulujących się błędów zaokrągleń z tego tytułu? W takim komputerze wynik obliczenia[10] `10 * 0.1` może nie być dokładnie równy `1`.
8. Czy możliwe jest, że dana zmienna ma wartość wykraczającą poza zakres wynikający z jej roli w programie? Jeśli na przykład jakaś zmienna rzeczywista reprezentuje prawdopodobieństwo zdarzenia, czy możliwe jest, że przypisuje się jej wartość ujemną albo większą od `1`?
9. Czy w konstrukcji wyrażenia zawierającego kilka operatorów należycie zrozumiane zostały reguły pierwszeństwa operatorów obowiązujące w danym języku?
10. Czy w programie występują przypadki niewłaściwego użycia arytmetyki całkowitoliczbowej? Jeżeli na przykład `i` jest zmienną całkowitą, czy warunek `2*i/2 == 1` prawdziwy jest zarówno dla parzystej, jak i nieparzystej wartości `i`? (Odpowiedź na to pytanie zależna jest od tego, które z działań — mnożenie czy dzielenie — wykonane zostanie najpierw).

Błędy porównywania

1. Czy w programie występują porównania zmiennych różnych typów, na przykład porównanie łańcucha ze wskaźnikiem, datą lub liczbą? Jeśli tak, to czy reguły

[10] Ułamek zapisywany w układzie dziesiętnym jako `0,1` w pozycyjnym układzie dwójkowym jest *nieskończonym* ułamkiem okresowym `0,0001100110011001100110011..` — *przyp. tłum.*

konwersji między typami danych zostały należycie zrozumiane?

2. Czy w programie występują porównania zmiennych o różnych długościach? Jak traktowane są one w danym języku programowania?[11]
3. Czy wybrano właściwy operator porównania? Programiści często błędnie interpretują warunki „co najwyżej", „co najmniej", „większy niż…", „nie większy niż…", „mniejszy lub równy" itp.
4. Czy składnia wyrażenia boolowskiego jest zgodna z intencją programisty? Czy właściwie zrozumiał on znaczenie i reguły pierwszeństwa operatorów `and`, `or`, `not` itp.
5. Czy argumenty operatorów boolowskich są wyrażeniami boolowskimi? Czy kombinacja operatorów boolowskich i operatorów porównania jest właściwa? Oto kilka przykładów błędów tego rodzaju. Warunku, iż wartość `i` zawiera się pomiędzy `2` a `10`, nie można zapisać jako `2 < i < 10`, lecz należy go zapisać w postaci `(2 < i) && (i < 10)`. Badanie, czy `i` większe jest od `x` lub `y`, nie może być zapisane jako `i > x||y`, lecz trzeba je zapisać w postaci `(i > x) || (i > y)`. Podobnie wyrażenie `if (a==b==c)` *nie* jest poprawnym testem na równość zmiennych `a`, `b` i `c`, zaś matematyczna relacja `x > y > z` musi być zapisana w postaci `(x > y) && (y > z)`.
6. Czy w kodzie programu, realizowanego na maszynie o architekturze binarnej, występują porównania liczb ułamkowych lub zmiennopozycyjnych? Mogą być one źródłem nieoczekiwanych błędów, bowiem niektórych skończonych ułamków dziesiętnych nie da się dokładnie zapisać w reprezentacji dwójkowej.
7. Czy przy konstrukcji złożonego wyrażenia boolowskiego należycie zrozumiano zasady pierwszeństwa operatorów boolowskich, obowiązujące w danym języku? Przykładowo, która operacja — alternatywa czy koniunkcja — wykonana

[11] W popularnym niegdyś języku CLIPPER łańcuchy `ALA` i `ALABASTER` mogły (w pewnych warunkach — ustawienie `SET EXACT OFF`) zostać uznane za równe, bowiem porównywanie kończyło się w momencie wyczerpania krótszego łańcucha. — *przyp. tłum.*

zostanie najpierw w wyrażeniu `if((a==2 && (b==2) || (c==3))`?

8. Czy sposób wartościowania wyrażeń boolowskich (wartościowanie częściowe albo kompletne) może mieć wpływ na poprawność programu? W wyrażeniu

```
if ((y==0) || (x/y)>z)
```

w sytuacji, gdy `y` równe jest zero, ostateczny wynik znany jest już po obliczeniu pierwszego członu. Obliczanie drugiego członu[12] nie jest już konieczne, lecz jeśli mimo to zostanie wykonane, wystąpi błąd dzielenia przez `0`.

Błędy przepływu sterowania

1. Jeżeli w programie występuje rozgałęzienie wielokierunkowe, na przykład „liczone `GO TO`" w języku FORTRAN, czy wartość zmiennej sterującej może przekroczyć liczbę wariantów skoku? Przykładowo, czy w instrukcji

```
GO TO (200, 300, 400), K
```

zmienna `K` może mieć wartość inną niż `1`, `2` lub `3`?

2. Czy każda pętla w programie ostatecznie się kończy? Spróbuj przeprowadzić nieformalny dowód tego dla każdej z nich.
3. Czy każda procedura, funkcja, podprogram i sam program kończą się ostatecznie?
4. Czy możliwe jest, że któraś pętla dla pewnych danych wejściowych nie wykona się ani razu? Jeśli tak, to czy jest to efekt zamierzony, czy przeoczenie? Przykładowo co stanie się w poniższym fragmencie

```
for (i==x ; i <= z ; i++) {
  ...
}
```

[12] W Turbo Pascalu i Delphi możliwe jest sterowanie sposobem wartościowania wyrażeń boolowskich za pomocą opcji kompilacji `$B`: gdy opcja ta jest włączona (`$B+`), każde wyrażenie boolowskie obliczane jest w sposób kompletny. Wyłączenie opcji powoduje zakończenie obliczeń wtedy, gdy wartość wyrażenia staje się przesądzona — *przyp. tłum.*

```
while (NOTFOUND) {
  ...
}
```

jeśli zmienna `NOTFOUND` będzie miała początkową wartość `false` lub `x` będzie większe niż `z`?

5. Co stanie się w przypadku pętli sterowanej zarówno przez iterację, jak i warunek boolowski (na przykład pętli realizującej przeszukiwanie tablicy), gdy warunek boolowski nie będzie nigdy spełniony? Przykładowo jak zachowa się pętla

```
DO I=1 to TABLESIZE WHILE (NOTFOUND)
```

gdy `NOTFOUND` nigdy nie osiągnie wartości `false`?

6. Czy program wolny jest od błędów „pomyłki o jeden”? Czy żadna z pętli nie wykonuje zbyt mało lub zbyt wiele iteracji? Wspomniany błąd charakterystyczny jest dla pętli rozpoczynających się od zerowej wartości zmiennej sterującej, bowiem wartość `0` często zostaje przeoczona. W poniższej pętli:

```
for (int i=0; i<=10;i++) {
  System.out.println(i);
}
```

wypisanych zostanie 11, a nie 10 wartości. Poprawna pętla zliczająca do dziesięciu powinna wyglądać następująco:

```
for (int i=0; i<=9;i++) {
  System.out.println(i);
}
```

7. Jeżeli składnia języka umożliwia grupowanie instrukcji, np. w bloki ujęte w nawiasy `{...}` czy też w pętle `do while`, to czy grupowanie to jest prawidłowe? Czy każdemu otwierającemu nawiasowi `{` odpowiada nawias zamykający `}`? Czy każde ciało pętli `while` jest prawidłowo ograniczone przez frazy `while` i `do`? Większość współczesnych kompilatorów automatycznie wykrywa tego typu nieprawidłowości.
8. Czy w każdej instrukcji warunkowej uwzględniono wszystkie możliwe sytuacje? Jeżeli na przykład przewidziano konkretne działania w sytuacjach, gdy testowana zmienna ma wartość `1`, `2` lub `3`, to czy można być pewnym, że nie może ona przyjmować żadnych innych wartości?

Błędy interfejsu

1. Czy liczba parametrów aktualnych w wywołaniach modułu (procedury, funkcji, podprogramu) równa jest[13] liczbie deklarowanych przez niego parametrów formalnych?
2. Czy atrybut (typ i rozmiar) każdego parametru aktualnego zgodny jest[14] z atrybutem odpowiadającego mu parametru formalnego?
3. Czy jednostka, w jakiej wyrażana jest wartość parametru aktualnego, zgodna jest z jednostką założoną przy deklarowaniu parametru formalnego? Czy na przykład do funkcji, której parametr jest z założenia wielkością kąta mierzonego w stopniach, przekazywana jest wielkość kąta mierzona w radianach?
4. Czy liczba parametrów w dynamicznym wywołaniu modułu jest równa liczbie parametrów spodziewanych w przypadku tego modułu?
5. Czy atrybut każdego parametru w dynamicznym wywołaniu modułu zgodny jest z atrybutem oczekiwanym dla tego parametru przez wywoływany moduł?
6. Czy jednostki, w których mierzone są wartości parametrów przekazywanych w dynamicznym wywołaniu modułu, zgodne są z jednostkami oczekiwanymi przez ten moduł?
7. Czy parametry w wywołaniach funkcji wbudowanych prawidłowe są co do liczby, atrybutów i kolejności?
8. Czy — w przypadku klasy lub modułu mających kilka punktów wejścia — w którymś z tych punktów występują odwołania do parametrów, które nie zostały zainicjowane? Sytuacja taka występuje w poniższym programie (w języku PL/I) — w punkcie wejścia `B` zmienna `X` nie ma określonej wartości.

---

[13] W niektórych językach (m. in. w C/C++) możliwe jest deklarowanie modułów (funkcji) dopuszczających zmienną liczbę parametrów wywołania. W Delphi istnieje namiastka tego mechanizmu w postaci tablic `array of const` — *przyp. tłum.*

[14] W niektórych językach (Delphi, Visual Basic, C++) typ zmiennej może przeobrażać się dynamicznie w czasie wykonywania programu — są to tzw. zmienne wariantowe. Jeżeli parametr formalny deklarowany jest jako wariantowy, to parametr aktualny może mieć *dowolny* typ zgodny z typem wariantowym — *przyp. tłum.*

```
A:   PROCEDURE(W, X);
     W=X+1;
     RETURN
B:   ENTRY(Y,Z);
     Y=X+Z;
     END;
```

9. Czy wykonanie podprogramu powoduje zmianę parametru, który z założenia powinien pozostać niezmienny?
10. Czy w przypadku użycia zmiennych globalnych są one „widziane" w identyczny sposób (pod względem definicji) przez wszystkie odwołujące się do nich moduły?
11. Czy w wywołaniach podprogramów (procedur, funkcji) używane są stałe (w roli parametrów aktualnych)? W niektórych implementacjach języka FORTRAN wywołanie

    ```
    CALL SUB(J,3)
    ```

    może być niebezpieczne, jeżeli bowiem wykonanie podprogramu `SUB` zmienia wartość drugiego parametru, to może ulec zniszczeniu obszar pamięci, w którym program przechowuje stałą `3`, w efekcie czego stała przestanie być stałą.

Błędy wejścia-wyjścia

1. Czy atrybuty jawnie deklarowanych plików są prawidłowe?
2. Czy atrybuty parametrów wywołania procedury `OPEN` są poprawne?
3. Czy specyfikacja formatu wejścia-wyjścia zgodna jest z zestawem zmiennych (wyrażeń) w instrukcji wejścia-wyjścia powołującej się na ten format? W języku FORTRAN istnieje ścisła zależność między listą wejścia-wyjścia w instrukcjach `READ` i `WRITE` a instrukcjami `FORMAT`, na które instrukcje te się powołują?
4. Czy dla programu dostępna jest wystarczająca ilość pamięci dla danych wczytywanych z plików?
5. Czy każdy używany plik zostaje otwarty przed wykonaniem na nim operacji odczytu lub zapisu?

6. Czy każdy otwierany plik jest zamykany po użyciu?
7. Czy wystąpienie końca pliku (`eof`) jest prawidłowo wykrywane i obsługiwane?
8. Czy wyjątki i błędy wejścia-wyjścia są prawidłowo wykrywane i obsługiwane?
9. Czy w tekstach drukowanych (wyświetlanych) przez program nie występują błędy literowe lub gramatyczne?

Inne błędy

1. Czy w (tworzonym przez kompilator) listingu odwołań do obiektów (*cross-reference listing*) występują zmienne, do których nie ma odwołania lub do których występują tylko odwołania jednokrotne? Jednokrotne odwołanie do zmiennej może świadczyć o tym, iż w rzeczywistości jest to zniekształcona nazwa innej zmiennej.
2. Jeśli kompilator tworzy listing atrybutów zmiennych, należy sprawdzić, czy którejś zmiennej nie został błędnie przypisany atrybut domyślny.
3. Jeżeli w wyniku kompilacji generowane są komunikaty ostrzegawcze (*warnings*) lub informacyjne (*hints*, *tips*, *infos*), należy każdy z nich dokładnie przeanalizować. Komunikaty ostrzegawcze sygnalizują sytuacje, które kompilatorowi „wydają się" podejrzane i które mogą oznaczać rozminięcie się programisty z założeniami. Komunikaty informacyjne dotyczą natomiast rozmaitych aspektów kodu: niezadeklarowanych zmiennych, konstrukcji utrudniających lub uniemożliwiających optymalizację itd.
4. Czy moduł (program) dokonuje dostatecznej kontroli poprawności danych wejściowych?
5. Czy w programie pominięto jakąś funkcję?

Powyższa lista kontrolna zestawiona została w skrócie w tabelach 3.1 i 3.2.

Tabela 3.1. *Lista kontrolna inspekcji programu, część pierwsza*

Odwołania do danych	**Obliczenia**
1. Użycie niezainicjowanych zmiennych. 2. Indeksy poza dopuszczalnym zakresem. 3. Niecałkowita wartość indeksu. 4. „Wiszące" wskaźniki i referencje. 5. Niezgodność atrybutów zmiennych współdzielących obszar pamięci. 6. Niezgodność atrybutów w strukturze lub rekordzie. 7. Obliczanie adresów danych niewyrównanych na granicy adresowania maszynowego. 8. Błędna struktura wskazywanego obszaru. 9. Niezgodność deklaracji struktury w procedurach. 10. Błąd „pomyłki o jeden" w indeksowaniu tablic lub łańcuchów. 11. Niespełnienie wymogów dziedziczenia klasy.	1. Obliczenia z udziałem zmiennych nienumerycznych. 2. Obliczenia z udziałem „mieszanych" typów danych. 3. Obliczenia z udziałem zmiennych o zróżnicowanej długości. 4. Zakres typu zmiennej wynikowej niewystarczający do pomieszczenia wyniku obliczeń. 5. Nadmiar lub niedomiar pośrednich wyników obliczeń. 6. Dzielenie przez zero. 7. Błędy zaokrągleń wynikające z binarnej arytmetyki zmiennopozycyjnej. 8. Wartość zmiennej poza dopuszczalnym zakresem. 9. Niewłaściwe założenie co do pierwszeństwa operatorów. 10. Nieprawidłowe dzielenie zmiennych całkowitych.
Deklaracje danych	**Porównania**
1. Użycie niezadeklarowanej zmiennej. 2. Błędne założenia co do ustawień domyślnych. 3. Niepoprawna inicjacja tablic i łańcuchów. 4. Niepoprawność długości, typu i zakresu widoczności zmiennej. 5. Niezgodność sposobu inicjowania zmiennej z jej klasą pamięciową. 6. Zmienne o bardzo podobnych nazwach.	1. Porównywanie niezgodnych zmiennych. 2. Porównywanie danych mieszanych typów. 3. Błędne operatory relacyjne. 4. Błędne wyrażenia boolowskie. 5. Błędna kombinacja operatorów boolowskich i operatorów relacyjnych. 6. Porównywanie wartości ułamkowych lub zmiennopozycyjnych w binarnej arytmetyce zmiennopozycyjnej. 7. Niewłaściwe założenie co do pierwszeństwa operatorów (porównania i boolowskich). 8. Błędna specyfikacja wartościowania wyrażeń boolowskich (wartościowanie skrócone albo kompletne).

Tabela 3.2. *Lista kontrolna inspekcji programu, część druga*

Przepływ sterowania	**Wejście–wyjście**
1. Niekompletne rozgałęzienia wielokierunkowe. 2. Niekończące się pętle. 3. Niekończący się program. 4. Pętle niewykonywane ani razu z powodu błędnych warunków wejściowych. 5. Błędne zakończenia pętli sterowanych dodatkowym warunkiem. 6. Błąd „pomyłki o jeden" w ustalaniu liczby iteracji pętli. 7. Niedopasowane ograniczniki DO/END. 8. Nieuwzględnienie pewnych sytuacji w wyrażeniu warunkowym.	1. Niepoprawne atrybuty plików. 2. Niepoprawne instrukcje OPEN. 3. Niezgodność formatów z instrukcjami wejścia-wyjścia. 4. Bufory zbyt małe dla wczytywanych danych. 5. Próba odczytu lub zapisu z (do) nieotwartego pliku. 6. Niezamknięcie pliku po użyciu. 7. Niepoprawna obsługa sytuacji końca pliku. 8. Niepoprawna obsługa błędów wejścia-wyjścia. 9. Błędy literowe i gramatyczne w drukowanych lub wyświetlanych komunikatach.
Interfejs	**Inne**
1. Niezgodna liczba parametrów formalnych i aktualnych. 2. Niezgodność atrybutów parametrów formalnych i aktualnych. 3. Niezgodne jednostki miar w wartościach parametrów formalnych i aktualnych. 4. Niewłaściwa liczba parametrów dynamicznego wywołania modułu. 5. Nieprawidłowe atrybuty parametrów dynamicznego wywołania modułu. 6. Niezgodne jednostki miar w wartościach parametrów dynamicznego wywołania modułu. 7. Niewłaściwa liczba, kolejność lub atrybuty parametrów w wywołaniu funkcji wbudowanej. 8. Odwołania do parametrów niezainicjowanych w danym punkcie wejściowym. 9. Niespójne odwołania do zmiennych globalnych w poszczególnych modułach. 10. Stałe w roli parametrów aktualnych.	1. Brak odwołań do niektórych zmiennych (wykazywany na listingu odwołań tworzonym przez kompilator). 2. Błędne przyporządkowanie zmiennym domyślnych atrybutów. 3. Ostrzegawcze i (lub) informacyjne komunikaty kompilatora. 4. Niedostateczna kontrola poprawności danych wejściowych modułu. 5. Przeoczenie jakiejś funkcji programu.

Wędrówki po kodzie

Idea „wędrówek po kodzie" (*walkthrougs*) zbliżona jest w swej istocie do inspekcji programu, bowiem podobnie jak ona sprowadza się do grupowego czytania kodu. Jej szczegółowa realizacja jest jednak nieco inna, inne są też techniki wykrywania błędów.

Podobnie jak inspekcja kodu, także wędrówki po kodzie odbywają się w ramach godzinnego lub dwugodzinnego, nieprzerwanego spotkania. Zespół „wędrowców" liczy od trzech do pięciu osób: jedna z nich odgrywa rolę podobną do roli moderatora w procesie inspekcji, druga (sekretarka) dokonuje rejestracji wykrytych błędów, trzecia jest natomiast testerem. Co do kwalifikacji członków zespołu zdania są podzielone: niewątpliwie pożądane jest, by był wśród nich programista, wskazana jest też obecność (1) wysoko kwalifikowanego programisty, (2) eksperta w zakresie konkretnego języka programowania, (3) programisty-nowicjusza, o świeżym i nieskażonym spojrzeniu na proces programowania, (4) osoby, która zajmuje się (będzie się zajmować) konserwacją programu, (5) uczestnika innego projektu i (6) innej osoby z zespołu realizującego projekt.

Procedura wędrówek po kodzie rozpoczyna się podobnie jak inspekcja: członkowie zespołu otrzymują z kilkudniowym wyprzedzeniem niezbędne materiały. Dalej jest już jednak całkiem inaczej, bowiem uczestnicy, zamiast wsłuchiwać się w narrację prowadzoną w kontekście listy kontrolnej, uskuteczniają swoistą „zabawę w komputer". Osoba, której wyznaczono rolę testera, przybywa na zebranie uzbrojona w niewielki zbiór zapisanych na papierze przypadków testowych — każdy z nich zawiera reprezentatywny zestaw danych wejściowych i oczywiście oczekiwany wynik dla tego zestawu. Zespól przystępuje następnie do *mentalnego wykonywania programu* na podstawie jego kodu źródłowego, z użyciem wspomnianych przypadków testowych jako danych wejściowych. Stan programu po wykonaniu każdej instrukcji zapisywany jest na kartce papieru lub (co wygodniejsze) na tablicy.

Zważywszy na znikomą „moc obliczeniową" owego symulowanego komputera, nie ma wątpliwości co do tego, że wykorzystywane przypadki testowe powinny być proste na tyle, aby nie wymagały skomplikowanych obliczeń. Nie odgrywają one zresztą krytycznej roli w całym procesie wędrówki — każdy z nich stanowi raczej wy god-

ny punkt startowy do symulowanego wykonywania kodu, podczas którego programista odpowiedzieć musi na szereg pytań (zadawanych przez innych członków zespołu) dotyczących przyjętych założeń, szczegółowych rozwiązań w zakresie kodowania itp.; większość błędów wykrywana jest właśnie w wyniku takiej konwersacji, a nie w rezultacie ostatecznego porównania oczekiwanego wyniku z rzeczywiście otrzymanym.

Podobnie jak w przypadku inspekcji kodu, tak i tym razem krytyczne znaczenie dla powodzenia całego przedsięwzięcia ma odpowiednia atmosfera i nastawienie członków zespołu. Wszelkie komentarze powinny być ukierunkowane raczej na sam program niż na jego autora; innymi słowy, wykrywane w programie błędy nie powinny być uważane za przejaw słabości programisty, lecz raczej za pewien przyrodzony atrybut samego procesu programowania.

Wszelkie zagadnienia pokrewne związane z wędrowaniem po kodzie źródłowym programu zbliżone są do tych związanych z inspekcją kodu. W szczególności wynikające z inspekcji pożyteczne efekty uboczne — jak identyfikacja sekcji szczególnie podatnych na błędy czy też rozmaite walory edukacyjne — pozostają w pełni aktualne także w procesie symulowanego wykonywania kodu.

Kontrola przy biurku

Kolejną techniką bezkomputerowego testowania kodu jest samodzielna analiza kodu programu, stanowiąca połączenie jednoosobowej inspekcji z wędrówką po kodzie źródłowym, zwana potocznie „kontrolą przy biurku" (*desk checking*). Osoba analizująca program weryfikuje go pod kątem obecności błędów figurujących na liście kontrolnej, jak również dokonuje symulacji jego wykonywania na podstawie własnoręcznie sporządzonych przypadków testowych.

W większości sytuacji postępowanie takie jest stosunkowo mało produktywne, z oczywistego powodu: osobą analizującą program jest zazwyczaj jego autor, co — jak wykazaliśmy w rozdziale 2. — nie jest czynnikiem sprzyjającym powodzeniu testowania. Wynika stąd dość interesujący pomysł, by dwaj programiści dokonujący analizy własnych programów po prostu wymienili się tymi programami; pomysł ten nie usuwa jednak kolejnego mankamentu, jakim jest

brak współzawodnictwa charakterystycznego dla pracy zespołowej — każdy uczestnik inspekcji czy wędrowania po kodzie ma bowiem ambicję wykrycia jak największej liczby błędów. Samotna analiza pozbawiona jest tego efektu synergii i jakkolwiek jest ona lepsza niż zupełna bezczynność (na polu testowania programu), to jednak jest znacznie mniej produktywna w porównaniu z wędrówką po kodzie czy jego inspekcją.

Wzajemna ocena

Ostatnia z technik bezkomputerowego wykorzystywania kodu programu, którą chcemy tu omówić, nie jest związana z testowaniem programów — jej celem *nie* jest wykrywanie błędów. Wspominamy o niej w tym miejscu dlatego, iż bezpośrednio związana jest z czytaniem kodu źródłowego. Celem tego czytania jest ocena (anonimowego) programu pod względem ogólnej jakości, łatwości konserwacji, możliwości rozbudowy, użyteczności, czytelności itp.; te cechy programu stanowią następnie podstawę do oceny jego autora jako programisty.

Administrator procesu, którym jest zwykle doświadczony programista, wybiera od 6 do 20 uczestników (6 stanowi absolutne minimum ze względu na zachowanie anonimowości). Wszyscy uczestnicy powinni mieć zbliżone kwalifikacje i profil programistyczny — nie zaleca się na przykład łączenia internetowych programistów korzystających z języka Java z programistami systemowymi wykorzystującymi głównie język asemblera. Każdy uczestnik proszony jest następnie o dostarczenie administratorowi dwóch własnych programów: najlepszego i najgorszego (wedle własnej oceny).

Administrator dokonuje następnie rozprowadzenia — w sposób losowy — otrzymanych programów wśród uczestników sesji. Każdy z uczestników otrzymuje cztery (anonimowe) programy; dwa z nich należą do kategorii „najlepszy" (wedle samooceny autora), a dwa pozostałe do kategorii „najgorszy", lecz osoba oceniająca nie zna przyporządkowania ocenianych przez siebie programów do poszczególnych kategorii. Osoba oceniająca ma następnie 30 minut na analizę wszystkich czterech programów, po czym powinna sklasyfikować każdy z nich według poniższych kryteriów, wartościując każde kryterium w skali od 1 do 7 (1 oznacza definitywne „tak", 7 — definitywne „nie"):

- Czy program jest łatwy do zrozumienia?
- Czy wysokopoziomowe aspekty projektu są sensowne i zauważalne?
- Czy niskopoziomowe aspekty projektu są sensowne i zauważalne?
- Czy modyfikacja programu byłaby zadaniem łatwym dla osoby oceniającej?
- Czy osoba oceniająca byłaby dumna ze stworzenia takiego programu?

Osoba oceniająca proszona jest także o inne komentarze i sugestie związane z programem, możliwościami jego ulepszeń itd.

Po zakończeniu sesji każdy z uczestników otrzymuje formularz z oceną dostarczonych przez siebie programów — oceną zarówno sumaryczną, jak i ocenami wystawionymi przez poszczególnych oceniających (ci ostatni nadal pozostają dla autora programów anonimowi). Autor programów otrzymuje także informację o względnej pozycji swych programów w ogólnym rankingu (pod kątem poszczególnych kryteriów), jak również porównanie własnej oceny innych programów z oceną własnych programów przez innych uczestników. W efekcie każdy uczestnik dysponuje informacją pozwalającą mu wyciągnąć odpowiednie wnioski odnośnie własnych kwalifikacji programistycznych. Ze względu na prostotę opisany proces da się łatwo zrealizować zarówno w szkolnej klasie, jak i w warunkach przemysłowych.

Podsumowanie

Niniejszy rozdział poświęcony jest testowaniu programów bez użycia komputera — które przez programistów wciąż nie jest należycie doceniane. Wielu z nich skłania się ku opinii, że program stworzony jest do wykonania przez maszynę i za pomocą tejże maszyny powinien być testowany. Założenie to jest z gruntu błędne, bowiem program, jako twór myśli ludzkiej, nadaje się także do analizy bezkomputerowej, w ramach której wykrywanie błędów może być równie efektywne, jak w procesie testowania maszynowego (a niekiedy nawet bardziej). Integralnym elementem realizacji projektu informatycznego

powinny być następujące techniki bezkomputerowego, „ludzkiego" testowania kodu:

- Inspekcja z wykorzystaniem listy kontrolnej.
- Wędrówka po kodzie, czyli grupowa „zabawa w komputer" polegająca na symulowanym wykonywaniu kodu źródłowego.
- Samodzielna analiza kodu „przy biurku".
- Wzajemna ocena umiejętności programistycznych w ramach zespołu.