Znajdź błąd. Sztuka analizowania kodu

Autor: Adam Barr
Tłumaczenie: Bartłomiej Garbacz
ISBN: 83-7361-855-4
Tytuł oryginału: Find the Bug: A Book of Incorrect Programs
Format: B5, stron: 288

Wyszukiwanie błędów w kodzie to czynność, którą programiści wykonują niemal równie często, jak pisanie kodu. Narzędzia do wykrywania i poprawiania błędów tylko częściowo rozwiązują problem. W wielu przypadkach błąd nie tkwi w nieprawidłowo sformułowanym poleceniu lub źle zdefiniowanej zmiennej, ale w miejscu, którego nawet najlepsze narzędzie nie znajdzie. Programista musi się nauczyć samemu bronić przed ukrytymi pomyłkami i nieprzyjemnymi niespodziankami. Błędy trzeba znaleźć, zanim one znajdą nas.

Książka „Znajdź błąd. Sztuka analizowania kodu" to zbiór 50 programów napisanych w językach Perl, C, Java, Python i asembler x86. Każdy z nich zawiera jeden, trudny do znalezienia, ale jak najbardziej realistyczny błąd. Wykrycie go wymaga przewidzenia sposobu, w jaki program będzie wykonywany, i prześledzenia krok po kroku jego działania. Każdy przykład opatrzony jest wskazówkami pomocnymi przy wyszukiwaniu błędów. Książka przedstawia sposoby analizowania programów i przewidywania miejsc, w których może wystąpić błąd.

- Klasyfikacja błędów
- Metody analizy kodu
- Błędy w programach w języku C
- Analiza aplikacji napisanych w języku Python
- Wyszukiwanie błędów w programach w języku Java
- Programy w języku Perl i asembler x86

Wykonując zadania zawarte w tej książce, nie tylko nauczysz się odnajdywać błędy, ale także udoskonalisz swoje umiejętności w zakresie pisania aplikacji

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

Spis treści

Rozdział 2.

Wskazówki dotyczące analizy kodu

Celem niniejszej książki jest udoskonalenie umiejętności Czytelnika w zakresie znajdowania błędów w kodzie. Zanim jednak przejdziemy do konkretnych przykładów, w niniejszym rozdziale zostaną przedstawione pewne porady odnośnie do czytania kodu. W zamierzeniu nie ma to być kompletne opracowanie technik usuwania błędów z kodu, lecz wprowadzenie do zagadnienia i przedstawienie informacji, które mogą okazać się przydatne podczas lektury problemów zawartych w kolejnych rozdziałach książki.

W swoim artykule *Tales of Debugging from the Front Lines* Marc Eisenstadt omawia różne sposoby znajdowania błędów. Jednym z nich jest, jak to określa, „zbieranie danych", które polega na przejrzeniu kodu w programie uruchomieniowym, dodaniu kodu opakowującego, wstawieniu instrukcji drukujących itd. Może się to okazać przydatnym sposobem usuwania błędów z kodu i wielu przypadkach będzie to metoda poprawna.

Jednak problemy opisane w niniejszej książce nie poddają się łatwo rozwiązaniu przy użyciu techniki zbierania danych, ponieważ w ich przypadku nie ma czego zbierać. Programy mieszczą się na stronie i w zamierzeniu ich analiza ma odbywać się właśnie w ten sposób. Można by, co prawda, wprowadzić je do komputera i wykonać, jednak stałoby to w sprzeczności z celami stawianymi niniejszej książce.

Zamierzeniem autora jest zmuszenie Czytelnika do analizy programów za pomocą, jak to określa Eisenstadt, ***inspekulacji***, którą opisuje jako „połączenie »inspekcji« (inspekcji kodu), »symulacji« (symulacji ręcznej) oraz »spekulacji«… Innymi słowy, [programiści] albo na pewien czas porzucają problem i zajmują się czymś innym, albo poświęcają dużo czasu na czytanie kodu i jego przemyślenie, być może symulując ręcznie jego wykonywanie. Chodzi o to, że tego rodzaju techniki nie wiążą się z prowadzeniem żadnych eksperymentów ani zbieraniem danych, a tylko »myśleniem« o kodzie".

Mówi się, że Archimedes — matematyk żyjący w trzecim wieku p. n. e. — po uświadomieniu sobie faktu, że wyporność obiektu jest zależna od ciężaru cieczy, którą wypiera, biegał nago po ulicach krzycząc „Eureka!", co w grece oznacza „znalazłem". Archimedes wchodząc do wanny obserwował, jak woda wylewa się poza jej brzegi w momencie zanurzania jego ciała, i właśnie wtedy przyszła mu do głowy owa genialna myśl. Znajdowanie błędów w kodzie może stanowić podobne doświadczenie. Nieoczekiwanie możemy sobie coś uświadomić, doświadczając własnego odkrycia (bieganie nago po ulicach nie jest obowiązkowe).

Niniejszy rozdział prezentuje szereg działań, które można podjąć w czasie analizy kodu. Często nie jest konieczne wykonywanie ich wszystkich — w dowolnej chwili przyczyna błędu może nagle się ujawnić, nawet jeśli nie rozpatruje się bezpośrednio zawierającego go kodu. Jednak jeśli taki przebłysk intuicji nie wystąpi w ogóle, można mieć nadzieję, że do czasu wykonania ostatniego z opisywanych działań błąd zostanie odkryty.

Działania, o których mowa, to:

1. Podział kodu na sekcje o określonych celach działania.
2. Identyfikacja znaczenia każdej zmiennej.
3. Wyszukanie znanych pułapek.
4. Wybór danych wejściowych dla celów analizy działania.
5. Analiza działania każdej sekcji kodu.

Poniżej działania te opisano bardziej szczegółowo.

Podział kodu na sekcje o określonych celach działania

Pierwszym etapem działań zmierzających do zrozumienia kodu jest dokonanie jego podziału na sekcje i zidentyfikowanie celów każdej z nich.

Sekcja jest fragmentem kodu wykonującym określone zadanie. Nie da się określić konkretnej liczby wierszy kodu składających się na sekcję; zależy to całkowicie od charakteru kodu. Sekcja może składać się z jednej instrukcji lub wywołania funkcji, albo stanowić pętlę o 30 wierszach kodu. Sekcję można zdefiniować jako dowolną sekwencję instrukcji programu, które wykonują wystarczająco dużo działań, by poświęcenie czasu na zdefiniowanie ich celów było uzasadnione.

„Celem działania" sekcji kodu jest zbiór zmian, które ma wprowadzić dany kod w strukturach danych używanych przez program. Jeżeli sekcja stanowi pełną funkcję, nazwa tej funkcji zwykle wskazuje ogólnie, jakie działania są wykonywane w ramach sekcji, jednak nie na tyle szczegółowo, by mogło to pomóc w analizie kodu. Jest to bardziej punkt wyjścia do dalszych analiz celów działania funkcji.

Identyfikacja sekcji w kodzie

Jeżeli znamy kod, który jest poddawany analizie, zadanie jego podziału na sekcje może okazać się proste, ponieważ wiemy, które jego części odpowiadają różnym częściom implementowanego algorytmu. Jeżeli jednak nie znamy kodu — czy to dlatego, że napisała go inna osoba, czy dlatego, że napisaliśmy go sami, ale na tyle dawno, że nie pamiętamy już szczegółów — trzeba poświęcić nieco czasu na przemyślenie kwestii podziału kodu.

Podstawowym etapem działań jest zlokalizowanie głównej części algorytmu. Większość funkcji rozpoczyna się od kodu wprowadzającego, obsługującego przypadki szczególne, błędy i inne podobne elementy, zaś kończy kodem czyszczącym, który zwykle zwraca pewne wartości do funkcji wywołującej. Między nimi znajduje się kod implementujący algorytm główny.

Algorytm główny jest tą częścią kodu, którą należałoby omówić w przypadku opisywania jego działania innej osobie. Można powiedzieć: „funkcja wyszukuje klucz w słowniku", bez wspominania o tym, że najpierw sprawdza ona, czy słownik jest poprawny, a później zwalnia bufor tymczasowy, który przydzieliła.

Oczywiście, kod wprowadzający i czyszczący również mogą zawierać błędy i muszą zostać poddane sprawdzeniu równie dokładnie, jak każdy inny fragment kodu. Jednak bez wątpienia te partie kodu zwykle są wykonywane dla dowolnych danych wejściowych, więc są testowane bezustannie. Ukryte błędy związane z postacią danych wejściowych mogą ukrywać się w algorytmie głównym. Jest to ta część kodu, która faktycznie odpowiada implementowanemu algorytmowi matematycznemu.

Stąd przydatną rzeczą jest określenie, gdzie kończy się kod wprowadzający, a gdzie zaczyna kod czyszczący. Należy zaznaczyć obszar znajdujący się między tymi wierszami jako lokalizację algorytmu głównego:

```
int find_largest_hash(String s[]) {
    if (s.length == 0) {
        throw new InvalidParameterException();
    }

    HashCalculator hb = new HashCalculator();
    int largesthash = hb.hash(s[0]);
    int newhash;
    for (int j = 1; j < s.length; j++) {
        newhash = hb.hash(s[j]);
        if (newhash > largesthash) {
            largesthash = newhash;
        }
    }

    hb.flush();

    return largesthash;
}
```

W powyższym przykładzie kod sprawdzający `s.length == 0` oraz trzy wiersze definiujące zmienne `hb`, `largesthash` oraz `newhash` stanowią kod wprowadzający. Wywołanie metody `hb.flush()` oraz instrukcja `return` to kod czyszczący. Reszta kodu stanowi algorytm główny.

Powyższy przykład pokazuje również, że nie trzeba znać dokładnie kodu w celu określenia rozmieszczenia sekcji. Choć nie mamy żadnych informacji na temat klasy `Hash➥Calculator`, i tak nie mamy wątpliwości, gdzie jest ona inicjalizowana, gdzie się jej używa w algorytmie głównym oraz gdzie są wykonywane działania czyszczące.

Jeżeli algorytm główny składa się z więcej niż kilku wierszy kodu, musi zostać podzielony na mniejsze części. Ponownie należy wziąć pod uwagę, jak można by opisać algorytm innej osobie. Każda część takiego opisu prawdopodobnie określa sekcję kodu. Gdybyśmy opisali algorytm następująco: „najpierw wczytujemy dane, potem rozmieszczamy je według wartości klucza, a następnie przekazujemy na wyjście", należałoby podjąć próbę podziału kodu na adekwatne trzy sekcje.

Identyfikacja celów działania każdej sekcji

Po dokonaniu podziału kodu na sekcje należy zidentyfikować cele działania każdej z nich. Jakie zmienne i w jaki sposób powinny zostać zmodyfikowane na końcu każdej sekcji? Jakie warunki niezmiennicze powinny być spełnione? W jaki sposób należy skonfigurować struktury danych?

Kiedy zakończymy etap podziału kodu na sekcje z określeniem ich celów, sprawdzamy, czy każdy z tych celów został uwzględniony: kod rozpoczynający działania związane z kolejnym celem zanim zakończy się przetwarzanie poprzedniego może być podatny na powstawanie błędów. Niektóre języki dopuszczają użycie instrukcji ***asercji*** (ang. *assertion*) które stanowią wyrażenia logiczne (zwykle testowane tylko w wersji uruchomieniowej kodu), powodujące zatrzymanie programu w przypadku, gdyż ich wartością okaże się fałsz. Luki między sekcjami często stanowią dobre miejsce na wstawienie instrukcji asercji, które weryfikują poprawne osiągnięcie celów stawianych danej sekcji, co pokazuje poniższy kod:

```
public class MyArray {
    public boolean isSorted() {
        for (int j = 0; j < data.length-1; j++) {
            if (data[j] > data[j+1]) {
                return false;
            }
        }
        return true;
    }
}

MyArray ma;

// Sortowanie tablicy
ma.sort();
assert (ma.isSorted());
```

Jeżeli sekcja kodu jest pętlą, należy określić ogólny cel jej działania. Jednak należy również postarać się określić cel działania pętli w każdym jej przebiegu. Przykładowo, w przypadku pętli sortującej tablicę celem jej pierwszego przebiegu może być: „pierwszy element w tablicy ma zawierać najmniejszą wartość".

W przypadku instrukcji `if` należy postarać się określić cel samego warunku `if`, na przykład: „blok `if()` zostanie wykonany wówczas, gdy użytkownik nie został jeszcze zweryfikowany".

Komentarze

Komentarze stanowią istotny element procesu określania celów działania fragmentów kodu. Stanowią one jedyną możliwość sformułowania w języku naturalnym i zakomunikowania przez programistę informacji o jego działaniach.

Wielu programistów pisze komentarze jako wskazówki, pomocne w momencie powtórnego przeglądania kodu. W wielu przypadkach komentarze, szczególnie te długie, wskazują obszary kodu, które w odczuciu autora były skomplikowane, niejasne lub w pewien inny sposób niełatwe w odbiorze w czasie późniejszej analizy. Obecność takich komentarzy zazwyczaj określa kluczowe części algorytmu.

Komentarze mogą również pomóc w identyfikacji przydatnych sekcji kodu, ponieważ często wielowierszowe komentarze objaśniające poprzedzają blok kodu warty zgrupowania w ramach jednej sekcji i stanowią próbę wyjaśnienia celów działania takiego fragmentu kodu.

Jednakże istotną rzeczą jest, by nie dać się zwieść komentarzom. Kompilator i (lub) interpreter ignoruje komentarze i niekiedy podobnie powinien postąpić czytający kod. Komentarze mogą nie być dostosowane do zaktualizowanej wersji kodu lub mogą w ogóle być błędne. Choć reprezentują punkt wyjścia do prób zrozumienia kodu, należy zweryfikować ich poprawność w stosunku do faktycznego kodu.

Niektóre komentarze są wstawiane bez namysłu, w przekonaniu, że wszystkie operacje wymagają opatrzenia komentarzem, jak w poniższym przykładzie:

```
// dodanie this_price do total
total += this_price
```

Tego rodzaju komentarze raczej nie pozwolą na ujawnienie obszarów kodu zawierających błędy. Z drugiej strony, prosty komentarz podobny do prezentowanego poniżej, który jest bez wątpienia błędny, stanowi sygnał, że w kodzie mogły być wprowadzone znaczące zmiany od czasu jego oryginalnego utworzenia:

```
// aktualizacja współrzędnej x
y_coord += delta;
```

Najprawdopodobniej ktoś zmienił ten kod w pośpiechu, być może wklejając go z innego dokumentu, a następnie zmienił nazwy zmiennych przy użyciu funkcji edytora automatycznego wyszukiwania i zastępowania ciągów znaków. W procesie tym mogło zostać wypaczone znaczenie i cele działania kodu.

Identyfikacja znaczenia każdej zmiennej

Po zidentyfikowaniu wszystkich sekcji należy przyjrzeć się zmiennym używanym w kodzie i określić „znaczenie" każdej z nich.

Znaczenie zmiennej oznacza wartość, jaką koncepcyjnie powinna ona przyjmować.

Nazwy zmiennych

Nazwy zmiennych, podobnie jak komentarze, mogą być albo przydatne, albo mylące.

W przeciwieństwie do sekcji kodu, wszystkie zmienne posiadają nazwy, co zwykle można wykorzystać, uzyskując pewne wskazówki co do znaczenia zmiennych. Nazwa zmiennej stanowi niejako mini-komentarz programisty, wstawiany w każdym miejscu użycia zmiennej. Jednak, podobnie jak w przypadku komentarzy, należy się upewnić, że zmienne rzeczywiście są używane w sposób, jaki sugerują ich nazwy. Ponadto niektóre zmienne, nawet te o istotnym znaczeniu, posiadają nazwy jednoliterowe lub inne mało znaczące:

```
float srednie_saldo // prawidłowo
string nazwa;       // OK, ale nazwa czego?
int k;              // niejasne, może oznaczać cokolwiek
```

W przeciwieństwie do komentarzy, kompilator lub interpreter **nie** ignoruje nazw zmiennych, ponieważ odwołują się one do określonych obszarów pamięci. Jednak kompilator czy interpreter nie stawia żadnych wymagań co do faktycznych nazw. Nazwanie zmiennej `a`, `suma` lub `wzyx` nie wpływa na sposób jej obsługi przez kompilator. To, co ma znaczenie, to wymóg poprawnego zadeklarowania, zdefiniowania i używania zmiennej w programie.

Jeżeli zmienna posiada niejasną nazwę lub nazwa ta nie odpowiada jej prawdziwemu znaczeniu, należy spróbować określić nową nazwę lub przynajmniej słowną definicję jej znaczenia. Przykładowo, w przypadku zmiennej o nazwie `i` można zapisać uwagę, że jest ona używana tylko jako licznik pętli lub że przechowuje identyfikator bieżącego użytkownika, albo że zawiera wskaźnik na kolejny wiersz danych wejściowych.

Określenie sposobów użycia każdej zmiennej

W przypadku każdej zmiennej wykorzystywanej w funkcji lub w bloku kodu należy sprawdzić, gdzie jest używana. Pierwszym krokiem jest odróżnienie miejsca użycia zmiennej w wyrażeniu (gdzie nie podlega modyfikacjom) od miejsca, gdzie przyjmuje nową wartość. Nie zawsze jest to oczywiste. Niektóre zmienne, szczególnie chodzi tu o struktury danych, mogą być modyfikowane w funkcjach, do których są przekazywane jako argumenty. Niektóre języki zapewniają sposoby określenia, że zmienna nie podlega zmianom w ramach funkcji (na przykład poprzez użycie słowa kluczowego `const` w językach C i C++), ale nie zawsze są one wykorzystywane:

```
suma += dane[j];   // zmienna suma jest modyfikowana, zmienne dane oraz j są używane
print(licznik);    // zmienna licznik jest używana
update(mystruct); // zmienna mystruct może być modyfikowana
```

Po określeniu, **gdzie** jest modyfikowana zmienna, można przejść do etapu podjęcia próby zrozumienia, **w jaki sposób** jest używana. Czy jej wartość jest stała w ramach całej funkcji? Czy jest stała w pojedynczej sekcji kodu? Czy jest używana tylko w jednej części kodu, czy wszędzie? Jeżeli jest używana w więcej niż jednej części, czy wykorzystuje się ją ponownie tylko w celu uniknięcia deklarowania dodatkowej zmiennej (liczniki pętli są często używane właśnie w ten sposób), czy może jej wartość określona na końcu jednej sekcji ma znaczenie na początku kolejnej?

Przeglądając pętle należy przeanalizować stan każdej zmiennej po zakończeniu pętli. Zmienne należy podzielić na te, które w czasie działania pętli nie zmieniały wartości, te, które były używane tylko w pętli (na przykład zmienne przechowujące wartości tymczasowe), oraz te, które będą używane po zakończeniu pętli z uwzględnieniem pewnych oczekiwań co do ich wartości (w oparciu o działania wykonane w pętli). Licznik pętli może należeć do jednej z dwóch ostatnich kategorii; często jest używany tylko w celu sterowania pętlą, jednak niekiedy używa się go po jej zakończeniu w celu ułatwienia określenia, co wydarzyło się w czasie jej wykonywania (szczególnie wówczas, gdy została zakończona przedwcześnie):

```
for (j = 0; j < total_records; j++) {
    if (end_of_file) {
        break;
    }
}
if (j == total_records) {
    // pętla nie została dokończona ze względu na wystąpienie końca pliku (end_of_file)
}
```

Ze względu na fakt, że wartość zwracana przez funkcję ma znaczenie, należy określić, czy zmienna jest używana tymczasowo w ramach funkcji, czy może stanowi część danych zwracanych do podprogramu wywołującego daną funkcję:

```
def sum_array( arr );
    tot = 0
    for j in arr:
        tot = tot + j
    return tot
```

Zmienna `arr` jest używana wewnątrz funkcji, jednak nie podlega modyfikacjom. Zmienna `j` jest modyfikowana, ale pod koniec funkcji jest niszczona. Z kolei zmienna `tot` jest modyfikowana, a następnie zwracana do podprogramu wywołującego.

Należy się upewnić, że wszystkie zmienne są inicjalizowane przed ich użyciem (niektóre kompilatory i interpretery ostrzegają użytkownika, jeżeli tak nie jest). Wielu zmiennym nie przypisuje się wartości początkowej w momencie ich deklarowania, więc istotne znaczenie ma to, aby przypisano im pewną wartość w ramach wszystkich możliwych ścieżek wykonania kodu, zanim zostaną użyte w wyrażeniu.

Zmienne ograniczone

Zmienne ograniczone (ang. *restricted variables*) mogą zawierać tylko określony podzbiór wartości, które mogłyby przechowywać w normalnej sytuacji w oparciu o swój typ. Przykładowo, w przypadku kodu symulacji toru wyścigowego o ośmiu torach można zdefiniować zmienną całkowitą o nazwie `lane`. W normalnym przypadku zmienna całkowita może przyjmować wartości z dużego przedziału, jednak w tym przypadku należy ograniczyć ten zakres do przedziału od 1 do 8 lub od 0 do 7. Należy to traktować jako element definicji znaczeniowej zmiennej.

Niektóre języki dopuszczają jawne określanie takich ograniczeń na zmiennych, jednak często programiści nie wykorzystują tej możliwości nawet tam, gdzie to możliwe. Przykładowo, programista może zdefiniować zbiór wyliczeniowych stałych `ONE`, `TWO`, `THREE`, `FOUR`, `FIVE`, `SIX`, `SEVEN` i `EIGHT`, a następnie może określić, że zmienna `lane` może przyjmować tylko wartości tych stałych. Jednak często istnieje konieczność dokonania wyboru między ścisłym sprawdzaniem typów (kompilator lub interpreter zapewnia wówczas, że zmiennej `lane` będzie przypisywana tylko jedna z ośmiu wymienionych wartości) a łatwością programowania (pozwalającą na przykład na wykonywanie operacji arytmetycznych na zmiennej `lane`, takich jak dodawanie wartości 1).

W idealnej sytuacji każda zmienna ograniczona jest definiowana jako taka — przynajmniej w komentarzu w miejscu jej definicji — być może poprzez samą nazwę zmiennej. Zmienne ograniczone są często używane na sposoby powodujące błędy, jeżeli ich wartość okaże się nie należeć do zakładanego przedziału. Tak więc istotną rzeczą jest określenie, czy i w jaki sposób zmienna podlega ograniczeniu:

```
char * get_lane_name(lane) {
    static char * lane_names = { "one", "two, "three",
                                 "four", "five", "six",
                                 "seven", "eight" };
    return lane_names[lane];
}
```

Wykonanie powyższego kodu zakończy się niepowodzeniem, jeżeli wartość parametru `lane` przekazanego do funkcji `get_lane_name()` nie będzie należała do przedziału od `0` do `7`.

Indeks tablicy (ang. *array index*) stanowi rodzaj zmiennej ograniczonej, ponieważ jego prawidłowe wartości determinuje rozmiar tablicy. Niektóre języki sprawdzają operacje dostępu do tablic w czasie uruchomienia i w razie potrzeby generują błędy. Inne języki bez żadnych problemów pozwalają na uzyskiwanie dostępu do dowolnych obszarów pamięci, na które wskazuje indeks. Wykorzystanie błędów czasu uruchomienia jest preferowanym rozwiązaniem, ponieważ w widoczny sposób pokazuje, że coś jest nie w porządku, jednak oba błędy mogą wystąpić z tego samego powodu.

Niestety, rozmiar tablicy również może podlegać dynamicznym zmianom i bywa trudny do określenia w danym miejscu kodu. Ponadto tablica może być indeksowana przy użyciu skomplikowanych wyrażeń. Weźmy pod uwagę poniższy przykład:

```
int array[100];
y = array[x];
```

W tym momencie jest rzeczą oczywistą, że zmienna `x` jest ograniczona do wartości z przedziału od `0` do `99` włącznie (zakładając, że wykorzystywany jest język indeksujący tablice od 0). Jeżeli instrukcja dostępu będzie jednak miała postać:

```
y = array[x-2];
```

to wartość zmiennej `x` będzie ograniczona do przedziału od `2` do `101`. W przypadku instrukcji podobnej do poniższej:

```
y = array[somefunction(x) / 3];
```

określenie poprawnych wartości dla zmiennej `x` może okazać się trudnym zadaniem, szczególnie wówczas, gdy liczba elementów w tablicy `array[]` została określona w czasie działania programu.

Warunki niezmiennicze

Warunki niezmiennicze stanowią uogólnioną formę zmiennych ograniczonych. Warunek niezmienniczy jest wyrażeniem, uwzględniającym jedną lub więcej zmiennych, co do którego przyjmuje się, że powinno mieć wartość prawdy przez czas wykonywania programu oprócz krótkich chwil związanych z aktualizacją wartości zmiennych. Warunek niezmienniczy jest zwykle ustaleniem określonym przez programistę w oparciu o to, w jaki sposób chce zarządzać strukturami danych używanymi przez program.

Biorąc pod uwagę zmienną będącą niebanalną strukturą danych, należy postarać się określić wszelkie warunki niezmiennicze, które zachowują wartość prawdy w przypadku, gdy struktura danych pozostaje w spójnym stanie. Przykładowo, struktura danych przechowująca ciąg znaków i długość może wymagać, aby owa długość zawsze uwzględniała długość ciągu. Należy się upewnić, czy wszystkie adekwatne elementy struktury danych są inicjalizowane w razie potrzeby. Kiedy struktura danych ulega modyfikacji, należy zapewnić, aby warunki niezmiennicze wciąż były spełnione.

W przedstawionym wcześniej przykładzie ze zmienną `lane` warunek niezmienniczy można by określić następująco:

```
(lane >= 1) && (lane <= 8)
```

Kolejny przykład, dotyczący listy jednokierunkowej, może mieć następującą postać:

```
if ((list_head != NULL) && (list_head->next != NULL))
    (list_head->next->previous == list_head)
```

Warunki niezmiennicze należy określać w miejscu, w którym występują, ponieważ stanowią one niejawny cel działań przed rozpoczęciem i po zakończeniu każdego bloku kodu programu. Ze względu na fakt, że cele działań stanowią teoretyczne idee, ignorowane przez kompilator i interpreter, warunki niezmiennicze są również dobrymi kandydatami do wykorzystania instrukcji `assert` w przypadku języków, które ją obsługują.

Występujące wcześniej zdanie, które określało, że warunki niezmiennicze zachowują wartość prawdy „oprócz krótkich chwil związanych z aktualizacją wartości zmiennych”, ma istotne znaczenie. W przypadku programów wielowątkowych trzeba pamiętać, że owe „krótkie chwile” są synchronizowane, więc inny wątek nie znajduje zmiennych w stanie, w którym warunek niezmienniczy ma wartość fałszu.

Śledzenie zmian zmiennych ograniczonych

Jak wcześniej wspomniano, pewne zmienne są ograniczone o tyle, że powinny zawierać tylko podzbiór możliwych wartości. Przykładowo, wartość całkowita używana jako wartość logiczna może zostać ograniczona do wartości 0 i 1. Ze względu na fakt, że takie ograniczenia mają zwykle charakter logiczny i nie są wymuszane przez kompilator lub interpreter, istotną rzeczą jest sprawdzenie modyfikacji zmiennych w celu upewnienia się, że ich wartości pozostają w ograniczonym przedziale.

Modyfikacje zmiennych ograniczonych można sprawdzić w toku procesu indukcyjnego. Oznacza to, że zanim zmienna zostanie zmodyfikowana, jeżeli założy się, że jej wartość bieżąca jest poprawnie ograniczona, istnieje możliwość udowodnienia, że owa wartość po dokonaniu modyfikacji również jest poprawnie ograniczona. Jeżeli można wykazać, że zmienna jest inicjalizowana poprawną wartością oraz że każda modyfikacja zachowuje jej poprawne ograniczenie, o ile tylko przed jej wykonaniem tak było, stanowi to dowód na to, że zmienna jest zawsze poprawnie ograniczona.

Przykładowo, jeżeli zmienna `grade` powinna zawierać wartość z przedziału od `1` do `4`, to poniższa instrukcja:

```
grade = 3;
```

zawsze zachowuje wartość zmiennej `grade` w ramach poprawnego ograniczenia. Jednak w przypadku zapisu takiego jak poniżej:

```
grade = 5 - grade;
```

nie jest jasne, czy wartość zmiennej `grade` wciąż będzie się znajdować w odpowiednim przedziale. Jeżeli jednak założymy, że zmienna `grade` była wcześniej poprawnie ograniczona do wartości od `1` do `4`, to w tym momencie wiemy, że wyrażenie `5 - grade` zachowuje wartość `grade` w odpowiednim przedziale.

Wyszukanie znanych pułapek

Jeżeli kod podzielono na sekcje i określono ich cele oraz zidentyfikowano prawdziwe znaczenie każdej zmiennej i przy wykonywaniu tych czynności nie znaleziono żadnego błędu, można kontynuować działania wybierając pewne dane wejściowe i analizując działanie kodu. Najpierw jednak należy szybko przejrzeć kod w poszukiwaniu kilku znanych, często występujących pułapek, bez wnikania w szczegóły.

Liczniki pętli

Liczniki pętli (ang. *loop counters*) są często używane w celu indeksowania tablic. W przypadku języków stosujących indeksowanie od zera należy sprawdzić, czy sprawdzenie warunku wyjścia z pętli wykorzystuje warunek <=, czy <. Kod podobny do przedstawionego poniżej:

```
for (index = 0; index <= MAX_COUNT; index++) {
    j = array[index];
}
```

może być poprawny, ale porównanie `index <= MAX_COUNT` jest podejrzane. W normalnej sytuacji, w przypadku tablic indeksowanych od zera, zapis ten powinien mieć postać `index < MAX_COUNT`, tak aby pętla nie wykonała przebiegu dla licznika `index` o wartości `MAX_COUNT`.

Jak wcześniej wspomniano, niektóre pętle z logicznego punktu widzenia posiadają wiele liczników, co można wyrazić w oczywisty sposób:

```
for (j = 0, k = 0; j < MAX_SIZE; j++, k += 2) {
    // treść pętli
}
```

lub w mniej zwarty sposób:

```
k = 0;
for (j = 0; j < MAX_SIZE; j++) {
    // treść pętli
    k += 2;
}
```

czy też całkowicie samodzielnie zajmując się obsługą warunków:

```
j = 0;
k = 0;
while (true) {
    if (j >= MAX_SIZE)
        break;
    // treść pętli
    j++;
    k += 2;
}
```

Powyższe trzy przykłady kodu wyglądają tak samo, jednak różnica polega na tym, że w drugim i trzecim przykładzie, gdyby do sekcji oznaczonej komentarzem `treść pętli` dodano instrukcję `continue`, spowodowałoby to pominięcie kodu modyfikującego wartości liczników pętli. W drugim przykładzie wartość `j` zostałaby zaktualizowana, jednak wartość `k` — nie. W trzecim przykładzie ani `j`, ani `k` nie zostałyby zaktualizowane.

W trzecim przykładzie wartość `k` jest zwiększana o `2` w każdym przebiegu pętli. Zwykle samo w sobie nie jest to błędem, jednak w normalnym przypadku zwiększanie jest wykonywane o `1`, więc jeżeli ktoś wykorzystał zwiększanie o `2`, prawdopodobnie miał w tym jakiś cel. Należy jednak zapamiętać, że wartość zmiennej `k` jest zwiększana w niestandardowy sposób.

Należy wystrzegać się kodu modyfikującego licznik pętli w niej samej. Zwykle jest to robione celowo i (przynajmniej tak powinno być) operacja taka jest opatrywana komentarzem, jednak znacznie utrudnia to analizę wykonywanych operacji w trakcie działania pętli — szczególnie wówczas, gdy modyfikacje są wykonywane tylko w pewnych przypadkach (w zależności od wartości danych podlegających przetwarzaniu w pętli):

```
for (p = 0; p < buffer_size; p++) {
    if (buffer[p] == '\') {
        // znak sterujący, więc pomijamy następny
        p++;
    }
    // treść pętli
}
```

W powyższym przykładzie po instrukcji `p++` nie występuje instrukcja `continue`, więc treść pętli głównej jest wykonywana bez przeszkód.

Wyrażenia występujące po lewej oraz po prawej stronie instrukcji przypisania

Ta sama zmienna lub wyrażenie czasem występuje po lewej, a czasem po prawej stronie instrukcji przypisania, które znajdują się blisko siebie. Może tak się zdarzyć w sytuacji, gdy wartość zmiennej jest używana w celu obliczenia wartości innej zmiennej, a następnie pierwsza z tych zmiennych jest oznaczana jako pusta, usuwana itp. Zazwyczaj można wyróżnić krok, w którym zmienna jest używana, oraz krok, w którym podlega modyfikacji (w poniższym przykładzie polega to na wyzerowaniu jej wartości):

```
total += array[m];
array[m] = 0;
```

W takiej sytuacji przekazanie zmiennej do funkcji może być logicznie równoważne jej wystąpieniu po prawej stronie instrukcji przypisania — jest to krok, w którym zmienna jest używana:

```
dump_contents(current_record);  // użycie
current_record.valid = -1;      // wyczyszczenie
```

Błąd występuje wówczas, gdy takie dwie instrukcje zostaną zamienione, to znaczy wartość zmiennej zostanie wyczyszczona **zanim** będzie użyta:

```
array[m] = 0;
total += array[m];  // wartość elementu już wynosi 0!!!
```

Inny przypadek to kod używany do zamiany wartości dwóch zmiennych, którego standardowy zapis to:

```
temp = var1;
var1 = var2;
var2 = temp;
```

Można z łatwością popełnić błąd w zapisie tych wierszy kodu — czy to pod względem ich kolejności, czy rozmieszczenia zmiennych.

Sprawdzenie operacji sprzężonych

Wiele operacji wykonujących pewne działania w programie posiada analogiczne operacje „wycofania” i muszą one być ze sobą odpowiednio sprzęgnięte.

Jednym z przykładów może tu być operacja przydzielania pamięci, a szczególnie pamięci tymczasowej przydzielanej przez funkcję. Całość takiej pamięci musi zostać zwolniona przed wyjściem z funkcji bez względu na warunek takiego wyjścia.

Niektóre języki nie pozwalają na jawne przydzielanie i zwalnianie pamięci, ale pewne operacje i tak muszą być sprzęgane: zakładanie i zdejmowanie blokad, zwiększanie i zmniejszanie wartości liczników odwołań itd. Kod podobny do przedstawionego poniżej:

```
process_record(record * rec) {
    acquire_lock(rec);
    if (somethingabout(rec)) {
        return 1;
    // reszta kodu
    release_lock(rec);
    return ret_val;
}
```

nie zawsze w poprawny sposób sprzęga wywołanie funkcji `acquire_lock(rec)` z wywołaniem `release_lock(rec)`. Ogólnie rzecz biorąc, w każdym miejscu, gdzie jest wykonywana pierwsza część operacji sprzężonej, należy się upewnić, czy także druga jej część jest zawsze wykonywana bez względu na wybraną ścieżkę wykonania kodu.

Wywołania funkcji

Wywołania funkcji (ang. *function calls*) mogą być trudne do analizy, ponieważ kod zawarty w funkcji nie znajduje się bezpośrednio przed czytającym treść programu. W najlepszym przypadku ma się do niego dostęp, ale zwykle trzeba polegać tylko na dokumentacji.

Poprawnie napisana funkcja modyfikuje tylko te zmienne, które powinna modyfikować. Wywołanie funkcji można traktować jako pojedynczą instrukcję przypisania, aczkolwiek może ona modyfikować wiele zmiennych i wykonywać bardziej skomplikowane działania na tablicach i strukturach.

W przypadku czytania kodu wywołania funkcji główną rzeczą wartą sprawdzenia jest to, czy parametry są do niej przekazywane w sposób poprawny. Większość kompilatorów i interpreterów wychwytuje argumenty o błędnym typie, jednak nie błędne argumenty o poprawnym typie.

Jedną z możliwości przekazania błędnego argumentu jest przypadek indeksu do tablicy. Ze względu na fakt, że każdy element tablicy posiada ten sam typ, można przekazać błędny argument o poprawnym typie po prostu myląc wartość indeksu. Typ indeksu jest zwykle jednym z typów podstawowych (umożliwiającym przechowywanie wartości całkowitej), więc nietrudno popełnić taki błąd. Przykładowo, w przypadku poniższego kodu:

```
call_func (struct_a, pointer_b, array[q])
```

istnieje prawdopodobieństwo, że jeżeli zmienne `struct_a` lub `pointer_b` mają błędny typ, kompilator zgłosi błąd. Jednak jeśli zmienna `q` jest wartością całkowitą, a zapis `array[q]` tak naprawdę miał mieć postać `array[r]` lub `array[s]`, kompilator nie zauważy niczego podejrzanego.

Wartości zwracane

Chociaż wiele funkcji manipuluje na przekazywanych do nich strukturach, w przypadku wielu innych ważna jest jedynie ***wartość zwracana*** (ang. *return value*) — jedyny stały wynik wykonania funkcji. Stąd całość kodu zapisanego z dużą ostrożnością i przeanalizowanego zda się na nic, jeżeli funkcja zwraca niepoprawną wartość.

Podstawowym błędem jest zwrócenie wartości nieodpowiedniej zmiennej, na przykład zwrócenie wskaźnika tymczasowego zamiast oczekiwanego, jak w poniższym kodzie:

```
record * find_largest(record list[]) {
    record * current_record;
    record * largest_record;
    // kod znajdujący largest_record
    return current_record;
}
```

Kod ten najprawdopodobniej powinien zwrócić wartość `largest_record`. Ze względu na fakt, że obie zmienne mają ten sam typ, kompilator nie ma żadnej możliwości stwierdzenia, że z semantycznego punktu widzenia kod jest niepoprawny.

Niektóre funkcje posiadają wiele instrukcji `return`. Powrót z funkcji w momencie, gdy znaleziono wynik, jest często o wiele łatwiejszy niż sprawdzanie, czy wciąż należy wykonywać jakieś działania, co pokazuje poniższy przykład:

```
def is_word(s):
    done = 0
    return_value = 0
    if len(s) == 0:
        return_value = 0;
        done = 1
    if done == 0:
        # pewien kod, który może ustawiać wartość return_value na 0 lub 1
    if done == 0:
        # pewien dodatkowy kod, który może ustawiać wartość return_value
    return return_value
```

Bardziej przejrzystym sposobem zapisu tego kodu może być zawarcie instrukcji `return` w każdym miejscu, gdzie ustawiana jest wartość `return_value`, zamiast używania zmiennej `done` w celu uniknięcia wykonywania pozostałego kodu. Tak więc pierwsza część funkcji miałaby następującą postać:

```
def is_word(s):
    if len(s) == 0:
        return 0
    # ciąg dalszy funkcji...
```

Jeżeli występuje wiele instrukcji `return`, należy się upewnić, czy każda ścieżka wykonania kodu pozwala na dotarcie do jednej z nich. Na pewno nie chcemy, aby kod miał postać podobną do poniższej:

```
def calculate_average(l):
    if len(l) == 0:
        return 0
    # dodatkowy kod
    if count > 0
        return total/count
```

Problem związany z powyższym kodem polega na tym, że może powodować wyjście z funkcji bez wykonania żadnej instrukcji `return`. Wiele języków nie pozwala na występowanie takiej sytuacji w przypadku funkcji zdefiniowanych jako zwracające określony typ danych, jednak w powyższym przykładzie, zapisanym w języku Python, funkcja zwróci wewnętrzną wartość `None`, która zapewne nie jest tym, czego oczekuje użytkownik.

Wreszcie, należy się upewnić, czy zwracane dane są wciąż poprawne. Nie należy zwracać wskaźnika na obszar pamięci, który został już zwolniony!

Kod podobny do istniejącego błędu

Jeżeli znajdzie się określony błąd, co do którego można podejrzewać, że może się powtórzyć w innym miejscu, należy wyszukać inne lokalizacje, w których tak jest. Błędy są powtarzane i może tak być dlatego, że kod został powielony lub autor kodu miał tendencję do popełniania danego błędu, czy też opacznie zrozumiał sposób działania kodu (programista konsekwentnie próbował zrobić coś poprawnie, lecz w rzeczywistości konsekwentnie popełniał błąd).

Przykładowo, jeżeli napotka się kod podobny do poniższego:

```
for (j = 0; k < MAX; k++)
```

należy poszukać innych pętli `for`, które pasowałyby do tego samego wzorca zapisu w celu zapewnienia, aby ten sam błąd nie był powielany gdzie indziej (szczególnie wówczas, gdy wygląda, jakby fragmenty kodu były kopiowane i wklejane w programie).

Podobnie, jeżeli odkryje się, że kod wywołuje funkcję z argumentami podanymi w niepoprawnej kolejności, należy sprawdzić inne miejsca jej wywołania. Jeżeli odkryje się błąd zakresu dla operacji dostępu do tablicy, należy sprawdzić inne miejsca, gdzie uzyskiwany jest dostęp do tej samej tablicy.

Wybór danych wejściowych dla celów analizy działania

Jeżeli po wykonaniu opisanych powyżej działań wciąż nie udało się zlokalizować błędu, najprawdopodobniej konieczne jest przeanalizowanie kodu „na piechotę”. W pewnym sensie taka analiza działania kodu nie jest rozwiązaniem najlepszym. W idealnej sytuacji pozwoliłoby to na udowodnienie, że każda sekcja wykonuje stawiane jej cele, każda zmienna jest wykorzystywana zgodnie z jej przeznaczeniem oraz że są zwracane i wyświetlane poprawne wartości, co nie pozostawia miejsca na żadne wątpliwości co do poprawności funkcji dla wszystkich danych wejściowych. Analiza działania kodu wprowadza element niepewności, ponieważ bez względu na ilość wypróbowywanych zestawów danych wejściowych błąd może nie zostać odkryty przy wykorzystaniu żadnego z nich.

Jednak w wielu przypadkach jedynym sposobem na znalezienie błędu jest właśnie dokonanie analizy działania kodu. W tym celu należy wybrać pewne dane wejściowe. Oprócz krótkich niezależnych programów, które obliczają określoną wartość (lub zbiór wartości), wszystkie sekcje kodu — czy to programu, czy funkcji, czy po prostu fragment większej sekcji kodu — zachowują się odmiennie w zależności od pobranych danych wejściowych.

W przypadkach, w których próbuje się wyśledzić błąd zgłoszony przez inną osobę, osoba ta może niekiedy określić dane wejściowe powodujące występowanie problemu. Stanowią one wówczas pierwszy zestaw danych wybieranych do analizy. Jednak niekiedy należy wybrać własne serie danych w celu znalezienia trudnego do powtórzenia lub niedostatecznie opisanego błędu, sprawdzając nowy kod przed jego opublikowaniem. Może to również dotyczyć sytuacji, w których zgłoszone dane wejściowe są zbyt skomplikowane, by z nich skorzystać. Analiza działania kodu jest czasochłonna. Nie można po prostu sprawdzić jego działania dla wszystkich danych wejściowych. Na szczęście często można wykorzystać małą próbkę takich danych, która jednak jest na tyle reprezentatywna, że pozwala na wykrycie wszystkich możliwych błędów.

Określając dane wejściowe należy pamiętać o tym, że nie jest się ograniczonym do wyboru tylko danych wejściowych funkcji zewnętrznych czy całego niezależnego programu. W rzeczywistości często łatwiej jest podzielić kod na mniejsze grupy i przeanalizować je w pierwszej kolejności. Po uzyskaniu pewności, że owe mniejsze grupy obsługują różne dane wejściowe w sposób poprawny, można się cofnąć i przeanalizować większe partie kodu bez konieczności powtórnego analizowania szczegółów sprawdzanych sekcji.

Z najłatwiejszym przypadkiem podziału kodu mamy do czynienia wówczas, gdy funkcje mają charakter warstwowy — jedna wykorzystuje drugą. Rozpoczynamy od funkcji znajdującej się na najniższym poziomie, czyli takiej, która nie zawiera żadnych wywołań innych funkcji w ramach sprawdzanego kodu. Następnie przechodzimy w górę hierarchii, sprawdzając po kolei każdą funkcję zewnętrzną.

Podobnie można postąpić w ramach pojedynczej funkcji po dokonaniu jej podziału na logiczne sekcje. Wybieramy sekcję, którą chcemy sprawdzić, a następnie określamy jej dane wejściowe. W tym przypadku na „dane wejściowe" składają się wartości wszystkich zmiennych, które są używane w badanej sekcji kodu. Jeżeli określiliśmy znaczenie każdej zmiennej, będziemy wiedzieć, które z nich są tu istotne.

Jeżeli program przechowuje pewne dane o swoim stanie między kolejnymi uruchomieniami kodu, należy również postarać się określić możliwe wartości takich danych. Przykładowo, w przypadku języków obiektowych funkcja, którą się bada, może być metodą klasy. Wówczas bieżący stan zmiennych składowych klasy (używanych w funkcji) z logicznego punktu widzenia stanowi część danych wejściowych takiej funkcji.

Wreszcie, powinno być rzeczą oczywistą, że wybierając testowe dane wejściowe należy znać oczekiwane dane wyjściowe. W przeciwnym razie określenie, czy program działa poprawnie, stanowi ogromną trudność.

Pokrycie kodu

Określając dane wejściowe dla znanego kodu ma się przewagę nad osobami, które przeprowadzają testy na kodzie stanowiącym „czarną skrzynkę" — mogą one jedynie go wykonywać i nie mają dostępu do źródeł. Przewaga wynika z tego, że w tym pierwszym przypadku można tak dobrać dane wejściowe, że zapewni się sprawdzenie całości kodu. Przykładowo, jeżeli w pewnym miejscu w kodzie występuje warunek `if()`, którego wartością może być prawda lub fałsz, można zapewnić wystąpienie co najmniej jednego zestawu danych wejściowych powodujących wystąpienie wartości prawdy i jednego powodującego wystąpienie wartości fałszu.

Kuszącą perspektywą może wydawać się przyjęcie założenia, że każdy obszerny lub różnorodny zestaw danych wejściowych w naturalny sposób zapewnia pokrycie całości kodu — szczególnie, jeśli będzie można go używać codziennie przez pewien czas. Niestety, jest to mało prawdopodobne. W rzeczywistości jest bardziej prawdopodobne, że poprawny będzie kod wykonywany dla wielu danych wejściowych, a nie kod wykonywany rzadko. Wynika to z faktu, że błędy występujące w często używanym kodzie z dużo większym prawdopodobieństwem mogły zostać wychwycone na etapie początkowych działań programistycznych i testowych.

Weźmy pod uwagę interesującą historię autorstwa Donalda Knutha dotyczącą kwestii przyjmowania założeń co do pokrycia kodu. Pochodzi ona z eseju *The Errors of TeX* (więcej informacji na jego temat Czytelnik znajdzie w dodatku A *Klasyfikacja błędów*):

> W czasie jednego ze swoich pierwszych eksperymentów napisałem niewielki kompilator dla firmy Burroughs Corporation wykorzystując język interpretowany opracowany specjalnie w tym celu. Rozbudowałem interpreter tak, aby zliczał ilość operacji interpretowania każdej instrukcji. Następnie przetestowałem nowy system kompilując dużą aplikację użytkową. Ku swojemu zaskoczeniu odkryłem, że ów duży test tak naprawdę pozwalał sprawdzić bardzo niewiele: ponad połowa liczników częstości wykonania zachowała wartości zerowe! Większość mojego kodu mogła zawierać całe mnóstwo błędów, a i tak aplikacja działałaby poprawnie. Napisałem więc nieeleganckie, sztucznie spreparowany program… i oczywiście odkryłem jednocześnie wiele nowych błędów. Mimo to okazało się, że wciąż 10% kodu nie zostało wykonane przez ów nowy test. Przyjrzałem się pozostałym wartościom zerowym i stwierdziłem, że mój kod źródłowy [pod względem danych testowych, a nie samego kompilatora] nie był dostatecznie nieeleganckie, nie uwzględniał pewnych przypadków szczególnych, o których zapomniałem. Nie sprawiło mi większej trudności dodanie kilku dodatkowych instrukcji, aż skonstruowałem procedurę testową, która wykonywała wszystkie instrukcje kompilatora oprócz jednej (później udowodniłem, że i tak nie mogłaby ona zostać wykonana w żadnej sytuacji, więc usunąłem ją).

Nie można zakładać, że przeprowadzone testy pozwolą na sprawdzenie całości kodu. Zamiast tego należy wybrać takie dane wejściowe, które pozwolą to zapewnić.

Jednym z aspektów procesu badania kodu, o którym należy pamiętać, jest „wywnioskowany przypadek przeciwny", co sprowadza się do tego, że jeżeli wszystkie działania wykonywane w przypadku, gdy wartością warunku `if()` jest prawda, **nie** są wykonywane,

jeżeli wartością tą jest fałsz. Najbardziej oczywistym przypadkiem „wywnioskowanego przypadku przeciwnego" jest sytuacja, gdy gałąź `else` nie występuje w ogóle, jak w poniższym przykładzie:

```
if (x == 5) {
    y = 7;
}
```

W tym przypadku „wywnioskowany przypadek przeciwny" polega na tym, że jeżeli `x` nie jest równe `5`, to `y` zachowuje swoją bieżącą wartość. Jednak nawet jeśli istnieje jawnie określona klauzula `else`, często można wywnioskować pewne informacje:

```
if (total > 20) {
    total = 0;
    carry = 1;
} else {
    total = total + 1;
}
```

W tym przypadku „wywnioskowany przypadek przeciwny" polega na tym, że wartość zmiennej `carry` pozostaje niezmieniona.

Oczywiście, instrukcje warunkowe mogą być odwracane (poprzez logiczne zanegowanie znaczenia warunku `if()` i zamianę miejscami klauzul `if` i `else`). Poniższy fragment kodu stanowi odwrócenie kodu przedstawionego powyżej:

```
if (total <= 20) {
    total = total + 1;
} else {
    total = 0;
    carry = 1;
}
```

Oznacza to, że klauzula `else` również posiada „wywnioskowany przypadek przeciwny".

W zakresie wyboru danych wejściowych należy również pokryć „wywnioskowany przypadek przeciwny". Jeżeli mamy do czynienia z kodem podobnym do poniższego:

```
if (tax > 0) {
    price += price * tax;
}
```

można by uznać, że pokrycie całości kodu zapewnią dane wejściowe ze zmienną `tax` większą od `0`, ponieważ zostanie wykonany każdy wiersz. Jednak należy również przewidzieć pokrycie „wywnioskowanego przypadku przeciwnego" poprzez użycie danych wejściowych ze zmienną `tax` równą `0`, co sprawi, że wartością warunku `if()` będzie fałsz.

Puste dane wejściowe

Puste dane wejściowe to sytuacja, w której nie istnieją dane, na których miałoby się odbywać przetwarzanie. Przykładowo, chodzi tu o program sortujący tablicę, do którego zostanie przekazana tablica licząca zero elementów lub program operujący na ciągach znaków, do którego zostanie przekazany ciąg pusty. W typowej sytuacji program obsługuje taki przypadek na jeden z dwóch sposobów — jawnie sprawdza występowanie takiej sytuacji na początku kodu:

```
void sort_array(int arr[], int count) {
    if (count == 0) {
        return;
    }
    // kod sortowania tablicy
}
```

lub obsługuje przypadek pusty w ramach głównego algorytmu:

```
void sort_array(int arr[], int count) {
    for (int i = 0; i < count; i++) {
        // kod sortowania tablicy
    }
}
```

Jeżeli wartością `count` jest `0`, sprawdzenie `i < count` od razu zwraca wartość fałszu, więc pętla główna nigdy nie jest wykonywana i kod nie przeprowadza żadnych działań.

Bez względu na sposób obsługi przypadku pustego należy określić postać odpowiednich danych pustych i przeanalizować działanie kodu dla takich danych wejściowych.

Banalne dane wejściowe

Banalne dane wejściowe stanowią kolejny krok w stosunku do danych pustych: chodzi tu o listę możliwych elementów zawierającą tylko jeden element, co sprawia, że konieczne do wykonania działania są banalne lub w ogóle nie trzeba podejmować żadnych działań. Przykładem banalnych danych może być program drukujący pierwsze `n` liczb pierwszych, kiedy nakaże mu się wyświetlenie pierwszej z nich, lub program usuwający powtarzające się elementy z tablicy, kiedy przekaże się do niego tablicę zawierającą jeden element.

Podobnie jak w przypadku danych pustych, dane banalne mogą być obsługiwane poprzez wykonywanie specjalnego sprawdzenia na początku, łączącego się często ze sprawdzeniem przypadku pustego:

```
void remove_dups(int arr[], int count) {
    if (count < 2) {
        return;
    }
    // reszta kodu funkcji remove_dups
}
```

albo też banalne dane wejściowe mogą być obsługiwane w ramach algorytmu głównego.

Ponownie, żaden z tych sposobów nie jest poprawny lub niepoprawny. Nadrzędnym celem jest zapewnienie, aby kod działał poprawnie, kiedy podda się go analizie z użyciem danych banalnych. Szczególnie w przypadkach, gdy dane banalne są obsługiwane przez algorytm główny, analiza działania kodu — nawet jeśli obsługa przypadku banalnego jest poprawna — może pozwolić na odkrycie sytuacji, w których obsługa przypadku niebanalnego byłaby niepoprawna.

Gotowe dane wejściowe

Gotowe dane wejściowe mają znaczenie w przypadku funkcji modyfikujących. Termin ten odnosi się do sytuacji, w której nie jest konieczne dokonanie żadnych zmian. Przykładem gotowych danych wejściowych jest funkcja zmieniająca litery w ciągu znaków na postać wielką, kiedy okaże się, że wszystkie litery ciągu są już wielkie.

Gotowe dane wejściowe powodują wykonanie kodu sprawdzającego, czy należy wykonywać jakieś działania, i nie powodują wykonania kodu odpowiedzialnego za faktyczne operacje:

```
void upper_case(char * s, int len) {
    for (int j = 0; j < len; j++) {
        if ((s[j] >= 'a') && (s[j] <= 'z')) {
            // kod zamiany litery s[j] na postać wielką
        }
    }
}
```

W przeciwieństwie do danych pustych i banalnych, zwykle nie jest możliwe (albo nie jest warte podejmowania wysiłków) określanie w kodzie w ramach początkowych sprawdzeń, czy dane wejściowe są od razu gotowe do przekazania na wyjście. W przedstawionym powyżej kodzie ciąg wejściowy `s` zawierający same wielkie litery i tak powoduje wykonanie całego przebiegu pętli `for()`. Jednak gdyby warunek `if()` w kolejnym wierszu zawsze przyjmował wartość fałszu, nie zostałyby znalezione żadne błędy w kodzie oznaczonym komentarzem *kod zamiany litery s[j] na postać wielką.*

Przy opracowywaniu danych wejściowych dla przypadku od razu rozwiązanego, jedną z kwestii, które należy uwzględnić, jest to, jak obszerne muszą być dane wejściowe. Przykładowo, dla przedstawionego powyżej kodu oznacza to udzielenie odpowiedzi na pytanie, ile znaków musi liczyć ciąg, aby można było odpowiednio przetestować kod. Odpowiedź na tak postawione pytanie jest względna. Ogólnie rzecz biorąc, wykorzystanie danych wejściowych o trzech do pięciu „elementach" (gdzie elementem jest pozycja tablicy, znak w ciągu znaków itd.) stanowi właściwy kompromis, gdyż oznacza dane na tyle krótkie, że można praktycznie przeanalizować kod przetwarzający te dane i jednocześnie na tyle obszerne, że zapewnia znalezienie błędów, których występowanie jest uzależnione od faktu, że określona liczba elementów występuje w danych wejściowych.

Należy zwrócić uwagę na przypadki, w których kod wydaje się wykonywać zbyt wiele działań na danych, które w rzeczywistości są już gotowe do przekazania na wyjście. Niepotrzebne przenoszenie danych, nawet jeśli są one z powrotem umieszczane na swoich oryginalnych pozycjach, bez wątpienia stanowi problem natury wydajnościowej i może wskazywać na występowanie błędu, który może się ujawnić w przypadku niegotowych danych wejściowych.

Błędne dane wejściowe

Błędne dane wejściowe to dane, które po prostu są niepoprawne. Przykładem może tu być funkcja pobierająca ciąg cyfr, do której przekaże się ciąg innych znaków lub funkcja pobierająca wskaźnik, do której przekaże się wskaźnik `NULL`.

W przypadku błędnych danych wejściowych oprócz konieczności upewnienia się, że funkcja obsługuje je bez powodowania awarii programu, należy również sprawdzić, czy zachowuje się ona poprawnie. W wielu przypadkach błędne dane wejściowe powinny być obsługiwane odmiennie od, na przykład, danych pustych, poprzez zwrócenie określonej wartości błędu lub zgłoszenie wyjątku.

W innych sytuacjach, gdy funkcja jest zagnieżdżona w innym kodzie stanowiącym część tego samego modułu, błędne dane wejściowe mogą być traktowane jako wystąpienie błędu po stronie funkcji wywołującej i w zamierzeniu mają nie być obsługiwane. Oczywiście, w przypadku niektórych funkcji nie występują żadne dane wejściowe, które można by traktować jako błędne, jednak w większości przypadków powinno być możliwe określenie błędnych danych wejściowych i poddanie kodu programu analizie z ich użyciem.

Pętle

Podobnie jak nie można przeanalizować kodu dla wszystkich możliwych danych wejściowych, tak zwykle nie można również poddać analizie każdego przebiegu pętli. W pewnych przypadkach można sterować liczbą przebiegów pętli ograniczając rozmiar danych wejściowych. W przypadku kodu podobnego do przedstawionego poniżej:

```
int sum_array(int arr[], int count) {
    int j;
    for (j = 0; j < count; j++) {
        // kod sumowania elementów tablicy
    }
    // zwrócenie sumy
}
```

dane wejściowe funkcji bezpośrednio sterują liczbą przebiegów pętli. Przedstawione wcześniej wskazówki odnośnie do liczby elementów w danych wejściowych mają zastosowanie także i tu. Najpierw należy wykonać kod dla zmiennej `count` równej `0` (przypadek pusty), następnie dla `count` równej `1` (przypadek banalny), a w końcu dla `count` należącej do przedziału od `3` do `5`.

Liczby losowe

Niektóre funkcje wykorzystują w swoich obliczeniach liczby generowane losowo. Funkcje takie zwykle wykorzystują pakiet obsługi liczb losowych napisany przez inną osobę, stanowiący część języka, systemu operacyjnego lub odrębnej biblioteki.

Głównym zmartwieniem związanym z liczbami losowymi jest konieczność sprawdzenia ścisłego zakresu zwracanych liczb losowych. Niekiedy zwracana jest wartość z przedziału od 0 do określonej wartości, a w innych przypadkach wartość z przedziału od 0 do 1. W innych sytuacjach kres górny wartości liczby losowej jest mniejszy od zadanej wartości, więc nigdy nie są one sobie równe. Przykładowo, język Python posiada standardową bibliotekę o nazwie `random`:

```
import random
index = int (len(my_array) * random.random())
```

Wywołanie funkcji `random.random()` powoduje zwrócenie liczby z przedziału od `0` do `1`, ale nie równej `1`, więc wywołanie to stanowi poprawny sposób losowego wybrania elementu z tablicy. Funkcja `random.random()` nigdy nie zwraca wartości 1, więc obliczany `index` nigdy nie będzie równy `len(my_array)` (co byłoby zbyt dużą wartością).

Wartość zwracana przez generator liczb losowych stanowi kolejną porcję danych wejściowych, nawet jeśli pojawia się nagle w środku kodu. Jako taka, musi zostać odpowiednio określona w toku procesu analizy działania kodu.

Najlepszym rozwiązaniem jest wybranie takich wartości, które znajdują się blisko kresu dolnego i górnego. W przedstawionym powyżej przykładzie wartościami tymi są `0` oraz wartość poprzedzająca `1`. Wybranie innych danych wejściowych zwykle jest uzależnione od tego, jakie działania są później wykonywane na wartości losowej. Jeżeli na przykład kod wykonuje jedną z trzech operacji w zależności od wybranej wartości losowej, należy wybrać trzy wartości odpowiadające owym trzem wyborom (jest bardzo prawdopodobne, że dwa z nich uwzględniają wartości 0 oraz „poprzedzającą 1”):

```
// Określamy, czy wynikiem uderzenia w rozgrywce baseballowej było ball, strike, czy foul
rnd = random.random()
if rnd < 0.3:
    ball()
elif rnd < 0.75:
    strike()
else:
    foul()
```

W tym przypadku chcemy wybrać jedną wartość mniejszą od 0,3, jedną z przedziału od 0,3 do 0,75 i jedną większą od 0,75.

W przypadku liczb losowych używanych w obliczeniach, a nie jako konkretne wartości wyboru, zazwyczaj odpowiednim rozwiązaniem jest wybranie wartości znajdującej się dokładnie pośrodku między dolnym a górnym kresem dziedziny jej wartości.

Analiza działania każdej sekcji kodu

W celu dokonania analizy działania kodu należy nauczyć się „myślenia jak komputer”, to znaczy tego, w jaki sposób należy badać kod źródłowy śledząc dokładnie stan, w jakim znajduje się komputer, a dzięki temu być może sprowokować bodziec „Eureka”, kiedy nagle zdamy sobie sprawę, w którym miejscu spodziewany stan różni się od stanu faktycznego. Innymi słowy, oznacza to znalezienie błędu.

Emulowanie działania komputera wydaje się oczywiste, jednak w praktyce może okazać się dość trudne.

Może okazać się trudne, szczególnie po przeczytaniu dużych ilości kodu, uniknięcie szybkiego przeglądania fragmentów kodu, który wydaje się prawidłowy. Należy pamiętać, że komputer każdą instrukcję przetwarza z równą uwagą i analizując kod musimy postępować tak samo. Bez względu na to, czy instrukcja wydaje się oczywista, czy definicja stałej wygląda na banalną, czy wyrażenie wydaje się być poprawne na pierw-

szy rzut oka, trzeba się zmusić do skupienia na faktycznej postaci kodu, a nie tym, jaka powinna ona być lub jaka chcielibyśmy, żeby była. Wiąże się to z analizą działania kodu dla określonych danych wejściowych. Nie dokonujemy analizy próbując śledzić szereg możliwości w oparciu o różne dane wejściowe, na przykład „ta zmienna będzie miała wartość 0, chyba że wysokość była większa od 100, gdyż wówczas będzie to wartość 1". Każda porcja danych wejściowych ma określoną wartość, która ściśle determinuje wartości innych zmiennych.

Śledzenie wartości zmiennych

Analizując działanie kodu należy śledzić wartości wszystkich zmiennych, chyba że określiliśmy, iż zmienna nie jest już istotna dla działania funkcji (jednak nawet wówczas może się okazać, że takie założenie było nieuzasadnione).

Istnieją dwa sposoby śledzenia zmiennych:

- Rozpoczynając badanie kolejnych instrukcji mówimy sobie „zmienna `x` ma wartość 12, więc w tym miejscu `subtotal` będzie równe 32…". Podejście takie sprawdza się w prostych przypadkach.
- Zapisujemy wszystkie zmienne na kartce papieru. Ten sposób jest lepszym rozwiązaniem wówczas, gdy istnieje wiele zmiennych lub instrukcje zawierają skomplikowane wyrażenia, gdzie analiza składniowa wymaga zbyt wiele uwagi, aby jednocześnie pamiętać wartości wszystkich zmiennych.

Weźmy pod uwagę poniższy przykład:

```
userid = get_userid();
access = (privilege > 3) ?
            max_access(userid) : (privilege << 2) + 1;
```

Hmm… jaką to wartość miała zmienna `userid`?

Należy pamiętać, że w przypadku każdej zmiennej każda instrukcja programu albo ją modyfikuje, albo nie. Komputer nigdy o niczym nie zapomina, także o zmodyfikowaniu zmiennej, jeżeli to konieczne. Zapisanie wszystkiego na papierze pomaga nam zapobiec zapomnieniu o czymś. Pomaga również w odkryciu, które zmienne zmieniają się w czasie wykonywania danej sekcji, a które pozostają niezmienione.

Jeżeli odkryje się, że wybrane dane wejściowe utrudniają śledzenie wartości zmiennych — na przykład, kiedy wybrana tablica jest zbyt obszerna — można wrócić i wybrać inne dane wejściowe. Trzeba jednak pamiętać, że określone błędy mogą wystąpić tylko w przypadku dostatecznie obszernych danych wejściowych.

Układ kodu

Układ kodu w przypadku większości języków ma służyć jako wskazówka dla osoby czytającej go, ale zwykle informacje te nie są wykorzystywane przez kompilator lub interpreter przy określaniu sposobu wykonania programu. O ile reguły języka wyraźnie

tego nie wymagają, wcięcia i rozmieszczenie nawiasów klamrowych nie powinny być wykorzystywane do wyciągania jakichkolwiek wniosków co do znaczenia kodu. Należy samodzielnie sprawdzić, czy aspekt semantyczny jest poprawny. Kod podobny do poniższego:

```
if (a == b)
    function_A();
    function_B();
```

zwykle oznacza coś zupełnie innego niż kod:

```
if (a == b) {
    function_A();
    function_B();
}
```

Może okazać się konieczne bardzo ostrożne czytanie kodu w celu zauważenia takiej różnicy.

Z drugiej strony, w przypadku pewnych języków układ kodu, na przykład po względem stosowanych wcięć lub tego, w której kolumnie występuje znak, **ma** znaczenie i może powodować odwrotne problemy, kiedy pomijamy znaczenie wcięć. W języku Python poniższy kod:

```
if a == b:
    function_A()
    function_B()
```

różni się od kodu

```
if a == b:
    function_A()
function_B()
```

Niepoprawnie zakończone komentarze również mogą zaciemnić prawdziwy charakter kodu. W przypadku poniższego fragmentu kodu w języku C:

```
/*
 * Dodanie x
 *

tot += x;

/*
 * teraz dodanie y
 */

tot += y;
```

instrukcja

```
tot += x;
```

nie jest wykonywana poprawnie, ponieważ stanowi część komentarza. Jeżeli wyszukujemy błędy w kodzie i udało nam się ograniczyć problem do niewielkiej sekcji kodu, ale nie możemy po prostu określić dokładnego miejsca występowania błędu, niektóre języki pozwalają na usunięcie komentarzy (na przykład poddając kod przetworzeniu przez preprocesor języka C) w celu sprawdzenia, czy błąd nie jest związany z usunięciem pewnych instrukcji przez błędnie wstawiony komentarz.

Ponadto należy zachować ostrożność odczytując skomplikowane wyrażenia arytmetyczne, szczególnie takie, które nie zawierają nawiasów jasno określających kolejność wykonywania działań. Jeżeli nie jest się pewnym, w jaki sposób dane wyrażenie zostanie poddane rozbiorowi składniowemu, można dodać nawiasy samodzielnie w sposób, który uważa się za słuszny, a następnie sprawdzić, czy zmienia to sposób działania programu.

Pętle

Pętle mogą być szczególnie kłopotliwe pod względem analizy, ponieważ zwykle nie da się zasymulować wszystkich ich przebiegów.

W przypadku kodu wykonywanego liniowo bez użycia pętli często można z łatwością znaleźć błędy badając każdy wiersz po kolei. Jednak w przypadku pętli zazwyczaj nie jest możliwe przejrzenie całego zestawu instrukcji, które zostaną wykonane w toku wszystkich przebiegów pętli.

W przypadku każdej pętli należy zwrócić uwagę na to, gdzie następuje wyjście z niej i dokąd jest przenoszone wówczas sterowanie. W normalnej sytuacji wyjście z pętli następuje na jej końcu, kiedy warunek zakończenia przyjmuje wartość fałszu, jednak wyjście może nastąpić również poprzez instrukcję `break` lub instrukcję `return` z poziomu funkcji. Należy określić, czy pętla zawiera instrukcję `break` i gdzie powoduje ona przeskok. Niektóre języki oferują możliwość określania kodu, który jest zawsze wykonywany w momencie zakończenia pętli, tak jak klauzula `else`, którą w języku Python można dodać do pętli (jest ona wykonywana, jeżeli pętla zakończy działanie w sposób normalny — kiedy pętla `for` dojdzie do końca lub warunek pętli `while` przyjmie wartość fałszu — ale nie kiedy wyjście z pętli następuje poprzez instrukcję `break`).

Oczywiście, należy pamiętać, że warunek zakończenia pętli jest jawnie sprawdzany jedynie na końcu pętli. W przypadku kodu podobnego do poniższego:

```
while x > 0:
    # blok kodu A
    if (pewien_warunek):
        x = 0
    # blok kodu B
```

fragment `blok kodu B` zostanie wykonany po ustawieniu wartości zmiennej `x` na `0`, chyba że po instrukcji `x = 0` jawnie doda się instrukcję `break`. Człowiek może stale wyliczać w pamięci warunek zakończenia pętli, ale komputer nie postępuje w ten sposób. Oznacza to, że jeżeli gdzieś we fragmencie *kod bloku B* zostanie przyjęte założenie, że `x` jest zawsze większe od `0`, wówczas program może ulec awarii.

Po zakończeniu pętli istotną rzeczą w takich przypadkach jest znajomość tego, w jakim stanie dany język pozostawia licznik pętli. W szczególności, czy zostanie ustawiony na wartość, którą posiadał w ostatnim przebiegu, czy może o jeden większą? Poniższa instrukcja pętli języka Python:

```
for i in range(3, 10):
```

oraz pętla języka C:

```
for (i = 3; i < 10; i++)
```

wydają się wykonywać te same działania: zmienna `i` przyjmuje wartości 3, 4, 5, 6, 7, 8 i 9. Jednak po wykonaniu pętli w języku Python wartością `i` będzie 9, natomiast w przypadku pętli języka C będzie to 10.

Kiedy mamy do czynienia z pętlą, która musi być wykonywana wielokrotnie, należy wybrać do analizy określone przebiegi. Dobrym wyborem początkowym będzie pierwszy, drugi, przedostatni oraz ostatni przebieg. Przykładowo, w przypadku kodu podobnego do poniższego:

```
for (k = 0; k < MAX_COUNT; k++) {
    // treść pętli
}
```

należy dokonać analizy dla `k` równego 0, 1, `MAX_COUNT-2` oraz `MAX_COUNT-1`. Oczywiście nie pozwoli to na wychwycenie wszystkich błędów, ale ogólnie rzecz biorąc, jeżeli pętla jest dla tych wartości wykonywana poprawnie, prawdopodobnie jest tak również w przypadku pozostałych wartości, których nie analizujemy.

W przypadkach, w których wynik jednego przebiegu działania pętli jest uzależniony od poprzedniego, często można skorzystać z procesu indukcji w celu udowodnienia, że pętla jest poprawna: zakładamy, że pętla działała poprawnie dla poprzedniego przebiegu, a następnie sprawdzamy, czy implikuje to, że będzie działała poprawnie także dla bieżącego przypadku.

Podsumowanie

Poniżej wymieniono działania, jakie należy podjąć badając kod. Trzeba pamiętać, że często nie jest konieczne wykonywanie ich wszystkich.

1. **Podział kodu na sekcje o określonych celach działania**. Dzielimy kod na mniejsze sekcje i określamy, jakie zmiany każda z nich powinna wprowadzać wśród zmiennych programu.
2. **Identyfikacja znaczenia każdej zmiennej**. Określamy logiczne znaczenie każdej zmiennej i zaznaczamy miejsce jej użycia i modyfikacji.
3. **Wyszukanie znanych pułapek**. Wykonujemy pewne szybkie sprawdzenia kodu w celu wyszukania podstawowych błędów, które można szybko zlokalizować.
4. **Wybór danych wejściowych dla celów analizy działania**. Wybieramy odpowiedni zestaw danych wejściowych do użycia w trakcie analizy działania kodu.
5. **Analiza działania każdej sekcji kodu**. Dokładnie badamy działanie kodu, emulując w swoim umyśle każdą instrukcję i śledząc zmiany dokonywane przez nie wśród zmiennych programu.