Baza programisty

Pragmatic Bookshelf

Debugowanie

Jak wyszukiwać i naprawiać błędy w kodzie oraz im zapobiegać

Paul Butcher

Mistrz debugowania w akcji!

Jak tworzyć oprogramowanie, które łatwo się debuguje?

Jak wykrywać potencjalne przyczyny problemów?

Jak ominąć pułapki czyhające na programistów?

Helion

helion.pl

Debugowanie. Jak wyszukiwać i naprawiać błędy w kodzie oraz im zapobiegać

Autor: Paul Butcher
Tłumaczenie: Andrzej Grażyński
ISBN: 978-83-246-2760-8
Tytuł oryginału: Debug It!: Find, Repair, and Prevent Bugs in Your Code
Format: 158×235, stron: 240

Mistrz debugowania w akcji!

- Jak tworzyć oprogramowanie, które łatwo się debuguje?
- Jak wykrywać potencjalne przyczyny problemów?
- Jak ominąć pułapki czyhające na programistów?

Zapewne niejednokrotnie podczas pracy przy komputerze musiałeś użerać się z wadliwymi aplikacjami. Doskonale wiesz, jak to jest, kiedy kolejne łatki usuwają stare błędy, równocześnie generując nowe, programiści zaś nie kwapią się do zmiany niewłaściwych założeń. A przecież jednym z najbardziej niedocenianych aspektów profesjonalnego programowania jest zdolność do rozpoznawania i usuwania błędów kryjących się w każdej większej partii stworzonego kodu. Jeśli tworzysz niebanalne aplikacje, najprawdopodobniej zajmiesz się ich debugowaniem chwilę po zakończeniu ich pisania. To zajęcie w zdecydowanie większym stopniu niż inne aspekty tworzenia oprogramowania jest działalnością intelektualną – ponieważ jego areną jest umysł programisty. Znajdowanie i wyjaśnianie przyczyn problemów powinno być pierwszą czynnością na drodze do ich zwalczania.

Ta książka poświęcona jest właśnie arkanom sztuki debugowania. Jej lektura pozwoli Ci znacznie ograniczyć liczbę popełnianych błędów, a te, które się pojawią, będą łatwiejsze do wykrycia i usunięcia. Podręcznik wyjaśni Ci, jak pisać kod, który łatwo debugować, przeprowadzi Cię przez proces wykrywania błędów, ich reprodukcji, diagnozowania, aż do wprowadzania i wycofywania poprawek w oprogramowaniu. Poznaj empiryczną metodę wykrywania błędów. Dowiedz się, jak ważne jest zapewnienie sobie pewnych sposobów reprodukowania błędnych zachowań. Naucz się unikać pułapek czyhających zarówno na programistów, jak i testerów. Stosuj powszechnie używane narzędzia i metody zapewniające automatyczne wykrywanie potencjalnych przyczyn problemów, zanim jeszcze się one pojawią! Naucz się tworzyć samodebugujące oprogramowanie, które automatycznie informuje o swoim stanie, a także sprawdź, co możesz zrobić, aby szybko wykrywać sytuacje będące potencjalną przyczyną problemów.

- Metoda empiryczna
- Reprodukcja błędów
- Diagnozowanie
- Wyszukiwanie błędów
- Wprowadzanie i wycofywanie poprawek
- Testowanie
- Przyczyny błędów
- Oprogramowanie samodebugujące
- Narzędzia wspomagające

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl

© Helion 1991–2010

Spis treści

Od tłumacza słów kilka 9

Przedmowa 13

Część I. Istota problemu

Rozdział 1. W tym szaleństwie jest metoda 19

1.1. Debugowanie to coś więcej niż eksterminacja błędów 20

1.2. Metoda empiryczna 22

1.3. Struktura procesu debugowania 23

1.4. Przede wszystkim rzeczy najważniejsze 24

1.5. Do dzieła! 28

Rozdział 2. Reprodukcja 29

2.1. Najpierw reprodukcja, potem pytania 29

2.2. Kontrolowane zachowanie aplikacji 32

2.3. Kontrolowane środowisko 32

2.4. Kontrolowane dane wejściowe 34

2.5. Ulepszanie reprodukcji 43

2.6. Gdy błąd nie chce się ujawnić 52

2.7. Podsumowanie 55

Rozdział 3. Diagnoza 57

3.1. Gdy debugowanie staje się nauką 57

3.2. Sztuczki i chwyty 63

3.3. Debuggery 70

3.4. Pułapki ... 71
3.5. Gry umysłowe ... 76
3.6. Zweryfikuj swoją diagnozę ... 80
3.7. Podsumowanie ... 81

Rozdział 4. Poprawki ... 83
4.1. Czysta tablica ... 84
4.2. Testowanie ... 85
4.3. Eliminowanie przyczyn, nie objawów ... 87
4.4. Refaktoryzacja ... 89
4.5. Kontrola wersji ... 91
4.6. Inspekcja kodu ... 92
4.7. Podsumowanie ... 93

Rozdział 5. Refleksja ... 95
5.1. Jak to w ogóle mogło działać? ... 95
5.2. Co poszło nie tak? ... 97
5.3. Nigdy więcej tego samego błędu ... 99
5.4. Zamykanie pętli ... 102
5.5. Podsumowanie ... 102

Część II. Szersza perspektywa

Rozdział 6. Chyba mamy problem... ... 105
6.1. Tropienie błędów ... 106
6.2. Współpraca z użytkownikami ... 109
6.3. Współdziałanie z innymi zespołami ... 116
6.4. Podsumowanie ... 117

Rozdział 7. Pragmatyczna nietolerancja ... 119
7.1. Błędy mają pierwszeństwo ... 119
7.2. Debugowanie a psychologia ... 122
7.3. Zasypywanie przepaści jakościowej ... 125
7.4. Podsumowanie ... 129

Część III. Debug-Fu

Rozdział 8. Przypadki szczególne ... 133
8.1. Łatanie istniejącego oprogramowania ... 133
8.2. Kompatybilność wersji ... 134
8.3. Współbieżność ... 139

8.4. Błędy heisenbergowskie 142
8.5. Problemy z wydajnością 144
8.6. Systemy osadzone 147
8.7. Błędy w obcym oprogramowaniu 150
8.8. Podsumowanie 154

Rozdział 9. Idealne środowisko debugowania 155
9.1. Automatyczne testowanie 155
9.2. Kontrola wersji 158
9.3. Automatyczne budowanie binariów 163
9.4. Podsumowanie 171

Rozdział 10. Naucz swe oprogramowanie samodebugowania 173
10.1. Założenia i asercje 174
10.2. Binaria debugowalne 184
10.3. Wycieki zasobów i błędna obsługa wyjątków 189
10.4. Podsumowanie 195

Rozdział 11. Antywzorce 197
11.1. Inflacja priorytetów 197
11.2. Primadonna 198
11.3. Zespół serwisowy 200
11.4. Gaszenie pożaru 202
11.5. Pisanie od nowa 203
11.6. Bezpański kod 205
11.7. Czarna magia 206
11.8. Podsumowanie 207

Dodatki

Dodatek A Materiały 211
A.1. Systemy kontroli wersji i śledzenia problemów 211
A.2. Narzędzia zarządzania generowaniem binariów i integracją ciągłą 215
A.3. Przydatne biblioteki 216
A.4. Inne narzędzia 218

Skorowidz 223

Rozdział 1.
W tym szaleństwie jest metoda

Tego można się było spodziewać: stworzony program nie chce działać, a jeżeli już działa, to w sposób dziwaczny. I co teraz?!

Niewątpliwie pożyteczny okazuje się wówczas nawyk systematyczności, czyli dążenie do znalezienia prawdziwej przyczyny (przyczyn) tak dziwacznego zachowania (starannie przecież napisanego) programu, wielu programistów jednak zdaje się wykazywać objawy chaotycznego — by nie rzec rozpaczliwego — miotania się po kodzie programu, bez wyraźnego celu i (co zrozumiałe) bez widocznych rezultatów. Cóż jednak odróżnia obie te grupy od siebie?

W tym rozdziale przyjrzymy się dokładnie metodom debugowania kodu, sprawdzonym w warunkach profesjonalnego wytwarzania oprogramowania. Nie da się ich, co prawda, porównać do kamienia filozoficznego, zamieniającego powszednią materię — czyli owoc niedoskonałych działań omylnych programistów — w złoto, czyli kod działający niezawodnie i bezbłędnie; w dalszym ciągu nieocenione okazują się określone kwalifikacje programisty czy testera — wiedza, doświadczenie, intuicja, zdolności detektywistyczne, no i — oczywiście — odrobina szczęścia. Metody te umożliwiają odpowiednie ukierunkowanie zainwestowanego wysiłku, pozwalają uniknąć bezsensownego drążenia rozmaitych hipotez, a w konsekwencji otwierają drogę — możliwie krótką — dotarcia do prawdziwej przyczyny problemu. Poznamy kilka konkretnych koncepcji, składających się na wspomnianą metodologię, m.in.:

- różnicę między „debugowaniem” a „usuwaniem błędów”,
- podejście empiryczne, czyli niech oprogramowanie samo zademonstruje, co się z nim dzieje,

- podstawowe elementy procesu debugowania — reprodukowanie błędów, diagnozowanie ich przyczyny, poprawianie kodu i wyciąganie wniosków na przyszłość,
- konieczność dokładnego rozumienia sensu podejmowanych działań.

1.1. Debugowanie to coś więcej niż eksterminacja błędów

Spytajcie niedoświadczonego programistę, co oznacza termin „debugowanie”, a usłyszycie, że jest to „znajdowanie błędów” czy inną podobną odpowiedź. Oczywiście, eliminowanie błędów *jest* jednym z elementów debugowania, nie jest to jednak element jedyny ani nawet najważniejszy. Efektywne debugowanie ukierunkowane jest pod kątem istotnych celów. Oto one.

1. Znalezienie *przyczyny* nieoczekiwanego zachowania programu.
2. Wyeliminowanie tej przyczyny.
3. Zapobieganie sytuacjom, w których wyeliminowanie jednej przyczyny jednej grupy błędów staje się kolejną przyczyną innych błędów.
4. Utrzymanie (a nawet ulepszenie) ogólnej jakości kodu — jego czytelności, przejrzystości, stopnia pokrycia przypadkami testowymi, wydajności, łatwości konserwacji i rozbudowy itp.
5. Upewnienie się, że określony problem został całkowicie wyeliminowany z kodu programu i nie da znać o sobie nigdy więcej.

Nieprzypadkowo najważniejszy okazuje się pierwszy z wymienionych celów — od prawidłowego rozpoznania rzeczywistej przyczyny problemu uzależnione jest powodzenie pozostałych działań.

Przede wszystkim — zrozumieć

Początkujący, niedoświadczeni programiści (i, niestety, wielu uważających się za doświadczonych) często zaniedbują rozpoznanie problemu „u źródła”, ograniczając się do dokonywania doraźnych poprawek, które *powinny* przywrócić[1] poprawne

[1] Nie można, oczywiście, *przywrócić* czegoś, czego nigdy nie było, szczególnie nie można *przywrócić* poprawnego działania programu, który nigdy poprawnie nie działał. To prawda, nie zapominajmy jednak, że każdy tworzony przez nas program funkcjonuje poprawnie w naszych oczekiwaniach, w naszej wyobraźni (a bywa i tak, że faktycznie okazuje się od razu bezbłędny — najbardziej odpowiednim pytaniem pod adresem programisty może być wówczas: „Co zamierzasz zrobić na bis?”). Umówmy się więc, iż owo „przywrócenie” oznacza przywrócenie adekwatności naszych oczekiwań po początkowo frustrującej konfrontacji z rzeczywistością — *przyp. tłum.*

funkcjonowanie programu. Gdy programista ma trochę szczęścia, owe doraźne zabiegi okazują się od razu chybione, co umożliwia uniknięcie marnotrawstwa czasu i wysiłku, niekiedy jednak, niestety, faktycznie eliminują konkretny błąd lub stwarzają pozory jego wyeliminowania. „Niestety", ponieważ w ostatecznym rozrachunku okazuje się, że owe nieprzemyślane i nie do końca zrozumiane zabiegi jedynie maskują prawdziwe przyczyny błędów, które niespodziewanie mogą się ujawnić, gdy będziemy tego najmniej oczekiwać. Co gorsza, nieprzemyślane łatanie kodu może mieć zgubne konsekwencje w postaci *destrukcji regresywnej*, czyli ponownego wprowadzenia do kodu błędów, których przyczyny zostały wcześniej wyeliminowane.

Zmarnowany czas i wysiłek

Kilka lat temu przyszło mi współpracować z zespołem bardzo doświadczonych i utalentowanych programistów. Swe zawodowe umiejętności zdobywali oni na bazie systemów klasy UNIX, jednak gdy do nich dołączyłem, dokonywali migracji uniksowego oprogramowania na platformę Windows; migracja ta znajdowała się już w dość zaawansowanym stadium.

Jeden z „błędów" wynikających z migracji ujawniał się w warunkach równoległej pracy wielu wątków. Niektóre wątki funkcjonowały bez zarzutu, inne jednak zdawały się pozostawać w stanie zagłodzenia[2]. Ponieważ w środowisku UNIX-a wszystko funkcjonowało bezbłędnie, przyczyna problemu zdawała się ewidentnie tkwić w specyfice mechanizmu zarządzania wątkami przez Windows. Podjęto więc decyzję o stworzeniu własnego podsystemu zarządzania wątkami, zastępującego oferowany standardowo przez Windows — przedsięwzięcie, owszem, bardzo ambitne i pracochłonne, niemniej jednak mieszczące się w granicach możliwości wspomnianego zespołu.

I rzeczywiście, gdy dołączyłem do zespołu, wydawało się, że wspomniany podsystem spełnia pokładane w nim nadzieje — nie zdarzały się już przypadki zagłodzenia wątków. Jednak tak głęboka ingerencja w system operacyjny rzadko obywa się bez skutków ubocznych i nie inaczej było tym razem: ceną zapłaconą za ulepszone zarządzanie wątkami okazało się ogólne, zauważalne spowolnienie pracy całego systemu Windows.

Dla mnie cała ta sytuacja była o tyle intrygująca, że sam w przeszłości musiałem zmagać się z rozmaitymi problemami wynikającymi z wielowątkowej architektury tworzonych przeze mnie aplikacji dla Windows. Pobieżna analiza problemu, którego konsekwencją była głęboka ingerencja w system operacyjny, wykazała, że jego *rzeczywistą przyczyną jest dynamiczne zwiększanie priorytetu wątków* (*dynamic thread priority boost*), które wyłączyć można w sposób

---

[2] Zagłodzeniem (*starvation*) wątku lub procesu nazywamy sytuację, w której mechanizmy szeregowania systemu operacyjnego konsekwentnie ignorują ów wątek lub proces w rywalizacji o dostęp do zasobów, w tym przypadku w rywalizacji o czas procesora — *przyp. tłum.*

elementarny, wywołując funkcję API `SetThreadPriorityBoost()`. Zamiast pracochłonnego — i podatnego na błędy — tworzenia własnego systemu szeregowania wątków, wystarczyłoby dodanie *jednego wiersza* kodu.

Morał? Obserwując niepożądane zachowanie aplikacji, zespół programistów dopatrzył się jego przyczyny w ogólnej architekturze Windows, bez głębszej analizy uwikłanego w to zachowanie podsystemu. Na pewno nie bez znaczenia były uwarunkowania kulturowe, czyli mówiąc po prostu, acz delikatnie, nie najlepsza reputacja systemu Windows w środowisku „hakerów uniksowych". Tak czy inaczej, gdyby programiści z zespołu poświęcili nieco czasu na wnikliwą analizę zjawiska i tkwiących u jego przyczyny mechanizmów oraz dostępnych ustawień, zaoszczędziliby znacznie więcej czasu, jaki stracony został na — niepotrzebną w gruncie rzeczy — konstrukcję własnego podsystemu. Uniknęliby ponadto degradacji wydajności systemu Windows i (bez wątpienia) wprowadzenia doń nowych błędów.

Mówiąc dosadnie: gdy rezygnujemy z poszukiwań prawdziwych przyczyn ujawniających się błędów, sami wystawiamy się poza nawias inżynierii programowania, zapuszczamy się w zdradliwy gąszcz „programowania voodoo"[3] lub „lub programowania przez przypadek"[4].

1.2. Metoda empiryczna

Do zrozumienia przyczyny zaistniałych błędów, manifestujących się niezgodnym z oczekiwaniami działaniem aplikacji, można dochodzić różnymi drogami. Generalnie każda metoda, która przybliża do tego celu, może być uznana za właściwą.

A skoro tak, to w większości przypadków debugowania najbardziej celowe — bo wysoce produktywne —okazuje się podejście *empiryczne*.

Eksperymentuj i obserwuj wyniki.

Podejście to zasadza się na obserwacjach i doświadczeniu, a nie na teorii czy też czysto logicznych spekulacjach. Owszem, *można* studiować kod programu i wyciągać z niego wnioski na temat skutków wykonania poszczególnych jego fragmentów (i czasem zdarza się, że nie ma innego wyboru),

3 Żargonowe określenie naśladujące powiedzenie prezydenta George'a Busha seniora, który mianem „voodoo economics" określał niektóre gospodarcze posunięcia Ronalda Reagana. Oznacza używanie w programowaniu funkcji lub algorytmów, których działania nie rozumie się w pełni, wskutek czego program nie działa albo jeśli przypadkiem działa, programista i tak nie rozumie, dlaczego tak się dzieje, cytat z *pl.wikipedia.org* — *przyp. tłum.*

4 Andrew Hunt, David Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, Reading, MA, 2000.

A propos natury oprogramowania...

„Oprogramowanie” to kategoria zasługująca na szczególną uwagę — co dla nas, mających z nią do czynienia na bieżąco, nie zawsze jest oczywiste.

Niewiele jest w ludzkiej działalności innych obszarów dających tak wielką okazję do nieograniczonej niemal realizacji własnej pomysłowości, wynalazczości; niewiele jest przykładów tworzywa intelektualnego dającego się kształtować w tak plastyczny sposób. Oprogramowanie ma tę cenną zaletę, że jest nośnikiem determinizmu — poza nielicznymi wyjątkami, które opiszemy w dalszej części książki, kolejny stan jego realizacji wyznaczony jest jednoznacznie przez stany poprzednie, a my uzyskujemy powtarzalny dostęp do każdego ze stanów na każde żądanie.

Nie mają tego szczęścia przedstawiciele większości dziedzin tradycyjnej inżynierii. Bo czyż można wyobrazić sobie mechanika Formuły 1, przyglądającego się silnikowi obracającemu się z prędkością 19000 obrotów na minutę i momentalnie — jakby w rezultacie mentalnego zatrzymania tych obrotów — zgłębiającego wszelkie detale rozmaitych konsekwencji tej dynamiki? Albo szczegółowo analizującego przebieg zapłonu w komorze spalania? W konfrontacji z szybkością działania ludzkich zmysłów wydaje się to absolutnie niewykonalne i faktycznie takie jest.

W naszym oprogramowaniu kręcąca się z szybkością milionów (czy nawet miliardów) na sekundę pętla wolna jest od owego piętna ulotności. Gdy na żądanie zostaje zatrzymana — w precyzyjnie określonym miejscu — możemy drobiazgowo, bez presji wynikającej z szalonego tempa, analizować wszelkie (dostępne) aspekty jej realizacji — przypomnijmy: aspekty deterministyczne, a więc powtarzalne. I ten właśnie istotny fakt predestynuje metodę empiryczną jako szczególnie cenną i produktywną w procesie debugowania.

lecz jest to metoda po pierwsze, mało efektywna, po drugie, zawodna. Znacznie efektywniej można dojść sedna problemu, konstruując odpowiedni eksperyment i *obserwując* faktyczne zachowanie się aplikacji. Jest to nie tylko efektywne, lecz daje także okazję do zweryfikowania przyjętych założeń dotyczących oczekiwanego jej zachowania. Tak oto skutecznym narzędziem w walce z tkwiącymi w oprogramowaniu błędami okazuje się… samo oprogramowanie.

W kolejnym punkcie zobaczymy, jak wykorzystać tę empiryczną filozofię do utworzenia strukturalnego modelu walki z błędami w oprogramowaniu.

1.3. Struktura procesu debugowania

Pod względem strukturalnym w procesie debugowania wyróżnić można następujące kluczowe elementy:

- reprodukcję — czyli wypracowanie sposobów na niezawodną i wygodną powtarzalność poszczególnych aspektów błędnego zachowania aplikacji;
- diagnozę — formułowanie i eksperymentalne weryfikowanie hipotez, zmierzające do ostatecznego ustalenia rzeczywistej przyczyny zaistniałych błędów;
- naprawę — projektując i urzeczywistniając zmiany w kodzie źródłowym w celu wyeliminowania konkretnego błędu, należy upewnić się, że nie spowodują one zniwelowania poprawek wprowadzonych wcześniej (a w konsekwencji ponownego zaistnienia błędów już wyeliminowanych — powszechnie określa się to zjawisko mianem *destrukcji regresywnej*); wprowadzane poprawki nie powinny też pogarszać wydajności kodu, jego czytelności i łatwości konserwacji;
- refleksję — skuteczne poprawienie błędu staje się swoistą lekcją, z której warto zawsze wyciągnąć stosowne wnioski. Co zrobiliśmy nie tak w tym fragmencie programu? Czy istnieją inne jeszcze fragmenty noszące znamię analogicznej pomyłki? Co możemy zrobić w celu upewnienia się, że *tego* błędu już nie powtórzymy?

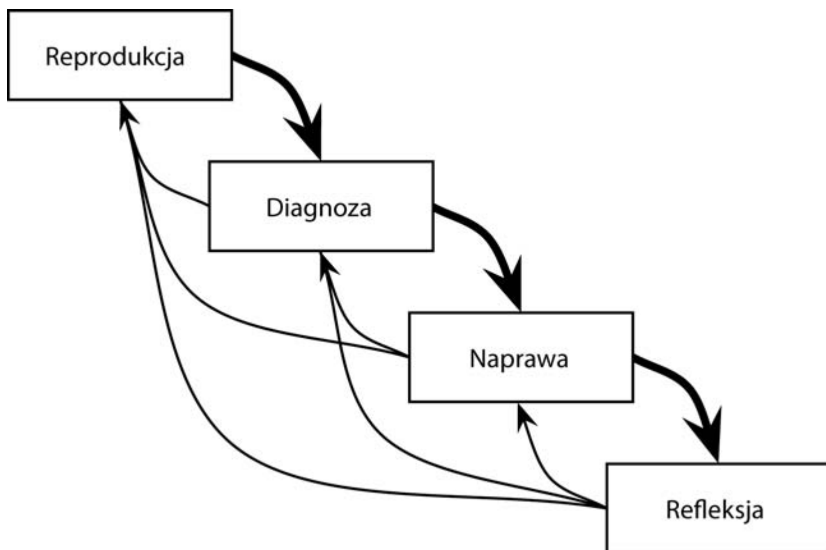
Debugowanie jest procesem iteracyjnym.

Jak pokazano na rysunku 1.1, etapy te zasadniczo układają się w prostą sekwencję; nie powinno to jednak sugerować, iż powiązane są ze sobą na zasadzie wodospadu. Co prawda, trudno przystępować do formułowania hipotez bez uprzedniego zapewnienia środków do reprodukowania błędu na żądanie, nie ma też większego sensu poprawianie kodu bez uprzedniego dogłębnego zrozumienia przyczyny wykrytego błędu, niemniej jednak między wymienionymi czterema etapami istnieją silne sprzężenia zwrotne powodujące, że debugowanie, zamiast sekwencją etapów, staje się procesem iteracyjnym. I tak np. na etapie formułowania hipotez możemy krytycznie ocenić skuteczność metod reprodukowania błędów (i skuteczność tę poprawić), zaś nanoszenie poprawek w kodzie może stać się okazją do zrewidowania przyjętych hipotez.

W następnych rozdziałach zatrzymamy się szczegółowo na każdym z wymienionych etapów; dalsza część tego rozdziału niech natomiast posłuży jako niezbędne do tego przygotowanie.

1.4. Przede wszystkim rzeczy najważniejsze

Mimo silnej pokusy, by jak najszybciej „zacząć debugowanie”, warto jednak zastanowić się, czy jesteśmy do tego należycie przygotowani.



Rysunek 1.1. Struktura procesu debugowania

Czy naprawdę wiesz, czego szukasz?

Zanim przystąpimy do reprodukowania problemu i stawiania hipotez na temat jego przyczyny, powinniśmy dokładnie zdawać sobie sprawę z tego, *cóż* w zachowaniu aplikacji *wydarzyło się* takiego, co skłonni jesteśmy uważać za ów „problem", oraz — co niemniej ważne — *co powinno się wydarzyć*, zgodnie z naszymi oczekiwaniami. Jeśli dysponujemy formalnym raportem na temat błędów, raport ten powinien zawierać niezbędne informacje w tym zakresie (raportami takimi zajmiemy się dokładniej w rozdziale 6.), na przeanalizowanie i zrozumienie których warto poświęcić trochę czasu.

Co powinno się wydarzyć, a co wydarzyło się naprawdę?

Jeśli nie ma raportu, bo błąd sami wykryliśmy lub dowiedzieliśmy się o nim z nieformalnej rozmowy, tym ważniejsze jest stworzenie adekwatnego obrazu całej sytuacji.

Nie zapominajmy jednak, że formalne raporty o błędach bywają nie mniej mylące niż wszelkie inne dokumenty. Jeżeli mianowicie stwierdzono w raporcie, że „wydarzyło się *to*, a powinno się wydarzyć *tamto*", czy zawsze stwierdzenie takie zgodne jest ze specyfikacją aplikacji? Jeżeli mamy w tym względzie wątpliwości, nie rozpoczynajmy konkretnych działań przed ich wyjaśnieniem, przed upewnieniem się, że wszystko należycie zrozumieliśmy. Nie warto ryzykować „psucia" aplikacji, czyli zmiany pożądanego jej zachowania na błędną, tylko dlatego, że tak sugerować może treść wspomnianego raportu.

Walka z niespójnymi danymi

Przyszło mi kiedyś zmagać się z dość prostym (wydawałoby się) błędem: raport transakcji, generowany przez system ewidencji sprzedaży, nie uwzględniał czasu letniego i okazywał się nieadekwatny do rzeczywistości w dniach, na przełomie których następowała zmiana czasu. Szybko wprowadziłem niezbędną poprawkę i zająłem się następnym problemem.

Niedługo potem poinformowano mnie o kolejnym błędzie: wykazana w raporcie liczba transakcji nie zgadzała się z liczbą transakcji faktycznie zrealizowanych, wskutek czego dział księgowości nie mógł sporządzić bilansu.

Źródło owej rozbieżności od razu stało się dla mnie oczywiste: proces rejestrowania transakcji nie uwzględniał zmiany czasu, w przeciwieństwie do generowanego raportu. Po krótkich konsultacjach dowiedziałem się, że opisany problem dał o sobie znać już rok temu i właśnie dla uniknięcia opisanej rozbieżności postanowiono istnienie czasu letniego po prostu ignorować[5].

Tak oto „błędne" działanie aplikacji, czyli rozbieżność jej zachowania z oczekiwaniami, nie było wcale winą aplikacji, a wynikało po prostu z nieadekwatności wspomnianych oczekiwań. Ponieważ raporty sprzedaży generowane były w różnych kontekstach, niektóre powinny uwzględniać zmianę czasu, inne raczej celowo ją ignorować. Rzetelnym rozwiązaniem problemu mogłoby być wprowadzenie do programu stosownej opcji rozróżniającej te dwa przypadki.

Jeden problem na raz

Ponieważ problemy często pojawiają się w grupach, istnienie pokusa zajmowania się równolegle wieloma z nich. Pokusa taka okazuje się jeszcze silniejsza, gdy wiele błędów wydaje się być związanych z tym samym obszarem kodu.

Takim pokusom należy się zdecydowanie opierać. Debugowanie jest procesem wystarczająco złożonym, by powstrzymać się od dalszej jego komplikacji. Jakkolwiek nie bylibyśmy ostrożni, zawsze istnieje możliwość, iż eksperyment zmierzający do wykrycia przyczyny jednego błędu interferować może z eksperymentami zaprojektowanymi w celu wykrycia innych. To w dużym stopniu utrudniać może rozumienie, co naprawdę dzieje się z aplikacją. Ponadto wprowadzanie do kodu każdej poprawki powinno być logicznie uzasadnione, a to staje się znacznie utrudnione, gdy próbujemy poprawiać kilka błędów na raz (wrócimy do tej kwestii w punkcie 4.5).

[5] Prawdę mówiąc, programista dokonujący poprawek mógł zaoszczędzić nam fatygi i umieścić w kodzie komentarz wyjaśniający przyczynę ignorowania czasu letniego i tym samym upewniający czytelnika kodu, że jest to działanie zamierzone.

Czasami może się okazać, że to, co skłonni byliśmy uważać za pojedynczy błąd, spowodowane jest wieloma przyczynami. Uświadamiamy sobie wówczas, że znaleźliśmy się w przysłowiowej „strefie mroku" — z aplikacją dzieją się różne kuriozalne rzeczy, niedające się wyjaśnić w wiarygodny sposób. Obszernie problem ten omówiono w punkcie 3.4.

Zaczynaj od rzeczy prostych

Wiele błędów programistycznych ma swe źródło w zwykłym przeoczeniu tych czy innych rzeczy. Mimo iż często przychodzi nam zmagać się z błędami bardziej subtelnej natury, nie można o takich prozaicznych przyczynach zapominać.

Tak się już stało, że my, programiści, przekonani jesteśmy co do tego, że wszystko powinniśmy robić we własnym zakresie. Przekonanie to często ujawnia się w postaci syndromu „co nie moje, to niedobre" (*Not Invented Here*), polegającego na ponownym wynajdywaniu koła, czyli (ułomnego nieraz) implementowania rzeczy, dla których gotowe, często perfekcyjne implementacje, dostępne są na wyciągnięcie ręki. W odniesieniu do debugowania syndrom ten przyjmuje postać dążenia do osobistego wyjaśnienia każdego aspektu nieoczekiwanego zachowania aplikacji.

A wystarczy po prostu spytać kolegów, czy w przeszłości zetknęli się z takim, czy innym problemem. To niewiele kosztuje, a może przyczynić się do uniknięcia znacznego marnotrawstwa czasu i wysiłku. Szczególnie w sytuacji, gdy nie jesteśmy ekspertami w dziedzinie, którą przyszło się nam zajmować.

Niespójność treści Subversion

(*Sean Ellis*)

Nie dalej jak kilka dni temu nasz nowy kolega zetknął się z problemem dziwnego działania polecenia `svn export`. Dziwnego, bo mimo identycznych numerów wersji, postać kodu aplikacji na serwerze Subversion różniła się od kopii roboczej w jego komputerze. Nic, tylko rwać włosy z głowy.

Po bezskutecznych próbach samodzielnego uporania się z problemem kolega poddał się i zapytał mnie, czy widzę jakieś wyjaśnienie rozbieżności, którą on jest mi w stanie natychmiast zaprezentować.

Odpowiedziałem twierdząco, wyjaśniając, iż przyczyną problemu jest błąd w jednej z bibliotek *runtime* serwera Apache, manifestujący się pod postacią nieprawidłowej interpretacji ścieżek zawierających sekwencje ../../... Po ostatecznym upewnieniu się, iż to właśnie jest istotą problemu, wystarczyło nam kilka minut, by zainstalować nowszą, poprawną wersję biblioteki.

Ostatecznie okazało się, iż błąd ten odkryty został już wiele miesięcy wcześniej, jednak odkrywca widocznie zapomniał podzielić się swą wiedzą z kolegami.

Jak widać, komunikacja jest zawsze istotna — nie tylko w kwestiach subtelnych, trudnych do pisania zwykłymi słowami, lecz także wówczas, gdy wystarczy po prostu zapytać: „Czy widzieliście już wcześniej coś podobnego?".

W następnych rozdziale zajmiemy się pierwszym z etapów iteracyjnego debugowania — reprodukowaniem błędnego zachowania aplikacji.

1.5. Do dzieła!

Dla przypomnienia...

- Upewnij się, że:
  - dążysz do poznania *prawdziwej przyczyny*, powodującej nieoczekiwane zachowanie aplikacji,
  - rzeczywiście eliminujesz określony błąd,
  - nie stwarzasz zagrożenia destrukcji „dobrego" kodu,
  - nie pogarszasz jakości kodu,
  - błąd, który właśnie poprawiasz, nie pozostanie w innym miejscu kodu,
  - nie popełnisz w przyszłości tego samego błędu.
- Wykorzystuj możliwości oprogramowania w zakresie *demonstracji* jego zachowania.
- Nie zajmuj się wieloma problemami jednocześnie.
- Upewnij się, że doskonale wiesz:
  - jak *powinna* zachowywać się aplikacja?
  - jak zachowuje się faktycznie?
- Poszukując przyczyn błędów, zaczynaj od rzeczy oczywistych.