helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

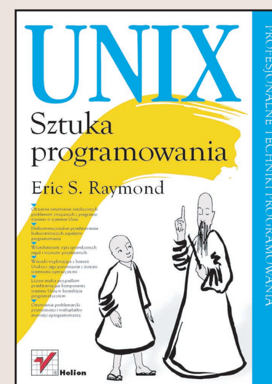
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

UNIX. Sztuka programowania



Autor: Eric S. Raymond
Tłumaczenie: Grzegorz Werner (wstęp, rozdz. 1 – 10), Wojciech Moch (rozdz. 11 – 20, dod. A – D)
ISBN: 83-7361-419-2
Tytuł oryginału: The Art of UNIX Programming
Format: B5, stron: 504

UNIX to nie tylko system operacyjny, to także społeczność i kultura

- Obszerne omówienie niezliczonych problemów związanych z programowaniem w systemie Unix
- Niekonwencjonalne przedstawienie kulturotwórczych aspektów programowania
- Wszechstronny opis sprawdzonych reguł i wzorców projektowych
- Wnioski wypływające z historii Uniksa i jego porównanie z innymi systemami operacyjnymi
- Liczne „studia przypadków” (case studies) przedstawiające komponenty systemu Unix w kontekście programistycznym
- Omówienie problematyki przenośności i wieloplatformowości oprogramowania

Unix to nie tylko system operacyjny; to także kultura i tradycja. Grzechem wielu programistów są programy spełniające wprawdzie swe funkcje, lecz zaprojektowane w pośpiechu, niedbale, a przez to trudne w utrzymaniu i rozwoju, oporne przy przenoszeniu na inną platformę i (z biegiem czasu) nieczytelne nawet dla swych autorów. Na temat dobrego programowania napisano już bardzo wiele; z tej książki dowiesz się nie tylko tego, jakie rozwiązania programistyczne warto poznać i naśladować, lecz także — dlaczego warto to robić.

Ta książka zawiera sporą dawkę wiedzy, lecz jej treść koncentruje się przede wszystkim na doświadczeniu programistycznym. Programowanie przestaje być tylko dyscypliną techniczną, a staje się zagadnieniem o charakterze kulturotwórczym. Doceniając należycie ten fakt, autor nie ogranicza się do technicznego wykładu. Prezentuje poszczególne aspekty profesjonalnego projektowania i implementacji programów w kontekście filozofii i historii Uniksa z jednej strony, a społecznymi uwarunkowaniami kultury uniksowej z drugiej.

Warto przeczytać tę książkę także z tej przyczyny, że mało który podręcznik dla programistów dostarcza tyle historii, folkloru i dygresji — elementów co prawda niekoniecznych z punktu widzenia meritum (choć i nad tą kwestią można by długo dyskutować), lecz znakomicie uprzyjemniających lekturę i być może ułatwiających zrozumienie tego, co w „suchym”, technicznym tekście może nieraz wydawać się zagadkowe.

Jeżeli więc jesteś programistą (niekoniecznie w systemie Unix) albo tylko interesują Cię zagadnienia związane z programowaniem, niniejsza książka z pewnością będzie stanowić interesującą lekturę, a być może również skłoni Cię do spojrzenia w inny sposób na swą codzienną pracę.

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

Spis treści

Przedmowa **13**
Kto powinien przeczytać tę książkę? 14
Jak korzystać z tej książki? 15
Pokrewne źródła 16
Konwencje używane w tej książce 17
Nasze studia przypadków 18

Część I Kontekst **19**

Rozdział 1. Filozofia: filozofia ma znaczenie **21**
1.1. Kultura? Jaka kultura? 21
1.2. Trwałość Uniksa 22
1.3. Argumenty przeciwko nauce kultury uniksowej 23
1.4. Co w Uniksie jest złe? 24
1.5. Co w Uniksie jest dobre? 25
1.5.1. Oprogramowanie o otwartych źródłach 25
1.5.2. Międzyplatformowa przenośność i otwarte standardy 25
1.5.3. Internet i World Wide Web 26
1.5.4. Społeczność Open Source 26
1.5.5. Prawdziwa elastyczność 27
1.5.6. Programowanie Uniksa jest przyjemne 27
1.5.7. Doświadczenie zdobyte w Uniksie można zastosować gdzie indziej 28
1.6. Podstawy filozofii uniksowej 29
1.6.1. Reguła modularności: Pisz proste części połączone przejrzystymi interfejsami 31
1.6.2. Reguła przejrzystości: przejrzystość jest lepsza niż spryt 32
1.6.3. Reguła kompozycji: Projektuj programy tak, aby dało się je łączyć z innymi 32
1.6.4. Reguła oddzielania: Oddzielaj politykę od mechanizmu; oddzielaj interfejsy od głównej części programu 33
1.6.5. Reguła prostoty: Projektuj pod kątem prostoty; uciekaj się do złożoności tylko tam, gdzie to konieczne 34
1.6.6. Reguła powściągliwości: Pisz duży program tylko wtedy, gdy zostanie jasno udowodnione, że nie da się inaczej 35
1.6.7. Reguła przezroczystości: Dbaj o zrozumiałość kodu, aby ułatwić badanie i debugowanie programów 35
1.6.8. Reguła odporności: odporność jest pochodną przezroczystości i prostoty 36
1.6.9. Reguła reprezentacji: Przełóż wiedzę na dane, aby logika programu mogła być prosta i odporna 37
1.6.10. Reguła najmniejszego zaskoczenia: Projektując interfejs, zawsze postępuj w najmniej zaskakujący sposób 37

1.6.11. Reguła milczenia: Kiedy program nie ma nic nieoczekiwanego do powiedzenia, nie powinien mówić nic ... 38
1.6.12. Reguła naprawy: Naprawiaj, co się da, ale jeśli program musi zawieść, niech zawiedzie z hukiem i jak najszybciej ... 38
1.6.13. Reguła ekonomii: Czas programisty jest drogi; oszczędzaj go zamiast czasu komputera ... 39
1.6.14. Reguła generacji: Unikaj programowania ręcznego; jeśli to możliwe, pisz programy piszące programy ... 40
1.6.15. Reguła optymalizacji: Napisz prototyp, zanim zaczniesz dopracowywać program. Sprawdź, czy działa, zanim zaczniesz go optymalizować ... 40
1.6.16. Reguła różnorodności: Nie ufaj żadnym deklaracjom o „jedynym słusznym sposobie” ... 42
1.6.17. Reguła rozszerzalności: Projektuj programy z myślą o przyszłości, bo nadejdzie ona wcześniej, niż się spodziewasz ... 42
1.7. Filozofia uniksowa w jednej lekcji ... 43
1.8. Stosowanie filozofii uniksowej ... 43
1.9. Liczy się też nastawienie ... 44

Rozdział 2. Historia: Opowieść o dwóch kulturach ... 45

2.1. Pochodzenie i historia Uniksa, lata 1969 – 1995 ... 45
2.1.1. Geneza: lata 1969 – 1971 ... 46
2.1.2. Exodus: lata 1971 – 1980 ... 48
2.1.3. TCP/IP i Wojny Uniksowe: lata 1980 – 1990 ... 50
2.1.4. Uderzenia w Imperium: lata 1991 – 1995 ... 56
2.2. Pochodzenie i historia hakerów, lata 1961 – 1995 ... 58
2.2.1. Zabawa w gajach Akademii: lata 1961 – 1980 ... 58
2.2.2. Fuzja internetowa i ruch wolnego oprogramowania: lata 1981 – 1991 ... 60
2.2.3. Linux i reakcja pragmatyków: lata 1991 – 1998 ... 62
2.3. Ruch Open Source — od roku 1998 do chwili obecnej ... 64
2.4. Lekcje płynące z historii Uniksa ... 66

Rozdział 3. Kontrasty: porównanie filozofii uniksowej z innymi ... 67

3.1. Elementy stylu systemu operacyjnego ... 67
3.1.1. Jaka jest idea unifikująca system operacyjny? ... 68
3.1.2. Wielozadaniowość ... 68
3.1.3. Współpracujące procesy ... 69
3.1.4. Granice wewnętrzne ... 70
3.1.5. Atrybuty plików i struktury rekordów ... 71
3.1.6. Binarne formaty plików ... 72
3.1.7. Preferowany styl interfejsu użytkownika ... 72
3.1.8. Zamierzone grono odbiorców ... 73
3.1.9. Bariera oddzielająca użytkownika od programisty ... 73
3.2. Porównanie systemów operacyjnych ... 74
3.2.1. VMS ... 76
3.2.2. MacOS ... 77
3.2.3. OS/2 ... 78
3.2.4. Windows NT ... 80
3.2.5. BeOS ... 83
3.2.6. MVS ... 85
3.2.7. VM/CMS ... 87
3.2.8. Linux ... 89
3.3. Co odchodzi, to wraca ... 90

Część II Projekt 93

Rozdział 4. Modularność: czystość i prostota 95
4.1. Hermetyzacja i optymalny rozmiar modułu 97
4.2. Zwartość i ortogonalność 98
4.2.1. Zwartość 99
4.2.2. Ortogonalność 100
4.2.3. Reguła SPOT 102
4.2.4. Zwartość i jedno silne centrum 103
4.2.5. Zalety niezaangażowania 105
4.3. Oprogramowanie ma wiele warstw 105
4.3.1. Od góry w dół czy od dołu w górę? 106
4.3.2. Warstwy spajające 108
4.3.3. Studium przypadku: język C jako cienka warstwa kleju 108
4.4. Biblioteki 110
4.4.1. Studium przypadku: wtyczki programu GIMP 111
4.5. Unix i języki obiektowe 112
4.6. Kodowanie z myślą o modularności 114

Rozdział 5. Tekstowość: dobre protokoły to dobra praktyka 115
5.1. Dlaczego tekstowość jest ważna? 117
5.1.1. Studium przypadku: format uniksowego pliku haseł 118
5.1.2. Studium przypadku: format pliku .newsrc 120
5.1.3. Studium przypadku: format pliku graficznego PNG 121
5.2. Metaformaty plików danych 122
5.2.1. Styl DSV 122
5.2.2. Format RFC 822 123
5.2.3. Format „słoika ciasteczek” 124
5.2.4. Format „słoika rekordów” 125
5.2.5. XML 126
5.2.6. Format plików INI systemu Windows 128
5.2.7. Uniksowe konwencje dotyczące formatu plików tekstowych 129
5.2.8. Zalety i wady kompresji plików 130
5.3. Projektowanie protokołów aplikacyjnych 131
5.3.1. Studium przypadku: SMTP, protokół transferu poczty 132
5.3.2. Studium przypadku: POP3, protokół skrzynki pocztowej 133
5.3.3. Studium przypadku: IMAP, internetowy protokół dostępu do poczty 134
5.4. Metaformaty protokołów aplikacyjnych 135
5.4.1. Klasyczny internetowy metaprotokół aplikacyjny 136
5.4.2. HTTP jako uniwersalny protokół aplikacyjny 136
5.4.3. BEEP: Blocks Extensible Exchange Protocol 138
5.4.4. XML-RPC, SOAP i Jabber 139

Rozdział 6. Przezroczystość: niech stanie się światłość 141
6.1. Studia przypadków 143
6.1.1. Studium przypadku: audacity 143
6.1.2. Studium przypadku: opcja -v programu fetchmail 144
6.1.3. Studium przypadku: GCC 146
6.1.4. Studium przypadku: kmail 147
6.5.1. Studium przypadku: SNG 148
6.1.6. Studium przypadku: baza danych terminfo 150
6.1.7. Studium przypadku: pliki danych gry Freeciv 153
6.2. Projektowanie pod kątem przezroczystości i odkrywalności 154
6.2.1. Zen przezroczystości 155
6.2.2. Kodowanie pod kątem przezroczystości i odkrywalności 156
6.2.3. Przezroczystość i unikanie nadopiekuńczości 157

6.2.4. Przezroczystość i edytowalne reprezentacje ... 158
6.2.5. Przezroczystość, diagnozowanie błędów i usuwanie skutków błędu ... 159
6.3. Projektowanie pod kątem konserwowalności ... 160

Rozdział 7. Wieloprogramowość: wyodrębnianie procesów w celu oddzielenia funkcji ... 163
7.1. Oddzielanie kontroli złożoności od dostrajania wydajności ... 165
7.2. Taksonomia uniksowych metod IPC ... 166
7.2.1. Przydzielanie zadań wyspecjalizowanym programom ... 166
7.2.2. Potoki, przekierowania i filtry ... 167
7.2.3. Nakładki ... 171
7.2.4. Nakładki zabezpieczające i łączenie Bernsteina ... 172
7.2.5. Procesy podrzędne ... 174
7.2.6. Równorzędna komunikacja międzyprocesowa ... 174
7.3. Problemy i metody, których należy unikać ... 181
7.3.1. Przestarzałe uniksowe metody IPC ... 182
7.3.2. Zdalne wywołania procedur ... 183
7.3.3. Wątki — groźba czy niebezpieczeństwo? ... 185
7.4. Dzielenie procesów na poziomie projektu ... 186

Rozdział 8. Minijęzyki: jak znaleźć notację, która śpiewa ... 189
8.1. Taksonomia języków ... 191
8.2. Stosowanie minijęzyków ... 193
8.2.1. Studium przypadku: sng ... 193
8.2.2. Studium przypadku: wyrażenia regularne ... 193
8.2.3. Studium przypadku: Glade ... 196
8.2.4. Studium przypadku: m4 ... 198
8.2.5. Studium przypadku: XSLT ... 198
8.2.6. Studium przypadku: warsztat dokumentatora ... 199
8.2.7. Studium przypadku: składnia pliku kontrolnego programu fetchmail ... 204
8.2.8. Studium przypadku: awk ... 205
8.2.9. Studium przypadku: PostScript ... 206
8.2.10. Studium przypadku: bc i dc ... 207
8.2.11. Studium przypadku: Emacs Lisp ... 209
8.2.12. Studium przypadku: JavaScript ... 209
8.3. Projektowanie minijęzyków ... 210
8.3.1. Wybór odpowiedniego poziomu złożoności ... 210
8.3.2. Rozszerzanie i osadzanie języków ... 212
8.3.3. Pisanie własnej gramatyki ... 213
8.3.4. Makra — strzeż się! ... 214
8.3.5. Język czy protokół aplikacyjny? ... 215

Rozdział 9. Generacja: podwyższanie poziomu specyfikacji ... 217
9.1. Programowanie sterowane danymi ... 218
9.1.1. Studium przypadku: ascii ... 219
9.1.2. Studium przypadku: statystyczne filtrowanie spamu ... 220
9.1.3. Studium przypadku: modyfikowanie metaklas w programie fetchmailconf ... 221
9.2. Doraźna generacja kodu ... 226
9.2.1. Studium przypadku: generowanie kodu wyświetlającego tabelę znaków w programie ascii ... 226
9.2.2. Studium przypadku: generowanie kodu HTML na podstawie listy tabelarycznej ... 228

Rozdział 10. Konfiguracja: jak zacząć od właściwej nogi ... 231
10.1. Co powinno być konfigurowalne? ... 231
10.2. Gdzie znajdują się dane konfiguracyjne? ... 233

10.3. Pliki kontrolne 234
10.3.1. Studium przypadku: plik .netrc 236
10.3.2. Przenoszenie do innych systemów operacyjnych 237
10.4. Zmienne środowiskowe 237
10.4.1. Systemowe zmienne środowiskowe 238
10.4.2. Zmienne środowiskowe definiowane przez użytkownika 239
10.4.3. Kiedy używać zmiennych środowiskowych? 240
10.4.4. Przenoszenie do innych systemów operacyjnych 241
10.5. Opcje wiersza polecenia 241
10.5.1. Opcje wiersza polecenia od –a do –z 242
10.5.2. Przenoszenie do innych systemów operacyjnych 247
10.6. Którą metodę wybrać? 247
10.6.1. Studium przypadku: fetchmail 248
10.6.2. Studium przypadku: serwer XFree86 249
10.7. O naruszaniu tych reguł 251

Rozdział 11. Interfejsy: Wzorce projektowe interfejsu użytkownika w środowisku uniksowym 253
11.1. Stosowanie Reguły Najmniejszego Zaskoczenia 254
11.2. Historia projektowania interfejsów w systemie Unix 256
11.3. Ocena projektów interfejsów 257
11.4. Różnice między CLI a interfejsami wizualnymi 259
11.4.1. Studium: Dwa sposoby pisania programu kalkulatora 263
11.5. Przezroczystość, wyrazistość i konfigurowalność 264
11.6. Uniksowe wzorce projektowe interfejsów 266
11.6.1. Wzorzec filtra 266
11.6.2. Wzorzec cantrip 268
11.6.3. Wzorzec źródła (source) 269
11.6.4. Wzorzec drenu (sink) 269
11.6.5. Wzorzec kompilatora 269
11.6.6. Wzorzec ed 270
11.6.7. Wzorzec roguelike 271
11.6.8. Wzorzec „rozdzielenia mechanizmu od interfejsu" 273
11.6.9. Wzorzec serwera CLI 278
11.6.10. Wzorce interfejsów oparte na językach 279
11.7. Stosowanie uniksowych wzorców projektowania interfejsów 280
11.7.1. Wzorzec programu poliwalencyjnego (wielowartościowego) 281
11.8. Przeglądarka internetowa i uniwersalny Front End 282
11.9. Milczenie jest złotem 284

Rozdział 12. Optymalizacja 287
12.1. Jeżeli masz zrobić cokolwiek, lepiej nie rób nic 287
12.2. Zmierz przed optymalizacją 288
12.3. Nielokalność bywa szkodliwa 290
12.4. Przepustowość i opóźnienia 291
12.4.1. Grupowanie operacji 292
12.4.2. Nakładające się operacje 293
12.4.3. Buforowanie wyników operacji 293

Rozdział 13. Złożoność: Tak prosto, jak tylko można, ale nie prościej 295
13.1. Mówiąc o złożoności 295
13.1.1. Trzy źródła złożoności 296
13.1.2. Wybór między złożonością interfejsu a złożonością implementacji 298
13.1.3. Złożoność niezbędna, opcjonalna i przypadkowa 299
13.1.4. Mapowanie złożoności 300
13.1.5. Gdy prostota nie wystarcza 301

13.2. Opowieść o pięciu edytorach ... 302
13.2.1. ed ... 303
13.2.2. vi ... 304
13.2.3. Sam ... 305
13.2.5. Wily ... 307
13.3. Właściwy rozmiar edytora ... 308
13.3.1. Identyfikowanie problemów ze złożonością ... 308
13.3.2. Nici z kompromisu ... 312
13.3.3. Czy Emacs jest argumentem przeciwko tradycji Uniksa? ... 313
13.4. Właściwy rozmiar programu ... 315

Część III Implementacja ... 317

Rozdział 14. Języki: w C albo nie w C? ... 319
14.1. Uniksowy róg obfitości języków ... 319
14.2. Dlaczego nie C? ... 320
14.3. Języki interpretowane i strategie mieszane ... 322
14.4. Ocena języków ... 323
14.4.1. C ... 323
14.4.2. C++ ... 325
14.4.3. Powłoka ... 327
14.4.4. Perl ... 330
14.4.5. Tcl ... 332
14.4.6. Python ... 334
14.4.7. Java ... 338
14.4.8. Emacs Lisp ... 341
14.5. Trendy na przyszłość ... 342
14.6. Wybór biblioteki systemu X ... 344

Rozdział 15. Narzędzia: Taktyki rozwoju ... 347
15.1. System operacyjny przyjazny dla programisty ... 347
15.2. Wybór edytora ... 348
15.2.1. Co należy wiedzieć o vi ... 349
15.2.2. Co należy wiedzieć o Emacsie ... 349
15.2.3. Wybór przeciw religii: używaj obu ... 350
15.3. Generatory kodu do zadań specjalnych ... 351
15.3.1. yacc i lex ... 351
15.3.2. Studium: Glade ... 354
15.4. make: automatyzacja przepisów ... 355
15.4.1. Podstawowa teoria make ... 355
15.4.2. Make w językach innych niż C i C++ ... 357
15.4.3. Produkcje użytkowe ... 357
15.4.4. Generowanie plików makefile ... 359
15.5. Systemy kontroli wersji ... 362
15.5.1. Po co kontrolować wersje? ... 362
15.5.2. Ręczna kontrola wersji ... 363
15.5.3. Automatyczna kontrola wersji ... 363
15.5.4. Uniksowe narzędzia kontroli wersji ... 364
15.6. Debugowanie w czasie działania programu ... 367
15.7. Profilowanie ... 368
15.8. Łączenie narzędzi z Emacsem ... 368
15.8.1. Emacs i make ... 369
15.8.2. Emacs i debugowanie w czasie działania programu ... 369
15.8.3. Emacs i kontrola wersji ... 370
15.8.4. Emacs i profilowanie ... 370
15.8.5. Jak IDE, ale lepsze ... 371

Rozdział 16. Ponowne wykorzystanie: Nie wyważajmy otwartych drzwi 373
16.1. Opowieść o Janie Nowicjuszu 374
16.2. Przezroczystość jako klucz do ponownego użycia kodu 377
16.3. Od ponownego wykorzystania do otwartych źródeł 378
16.4. Najlepsze rzeczy w życiu są otwarte 380
16.5. Gdzie szukać? 382
16.6. Kwestie związane z używaniem otwartego oprogramowania 383
16.7. Licencje 384
16.7.1. Co można uznać za otwarte oprogramowanie 385
16.7.2. Standardowe licencje otwartego oprogramowania 386
16.7.3. Kiedy potrzebny jest prawnik? 388

Część IV Społeczność 391

Rozdział 17. Przenośność: Przenośność oprogramowania i utrzymywanie standardów 393
17.1. Ewolucja języka C 394
17.1.1. Wczesna historia języka C 395
17.1.2. Standardy języka C 396
17.2. Standardy Uniksa 398
17.2.1. Standardy i Wojny Uniksów 398
17.2.2. Duch na uczcie zwycięstwa 401
17.2.3. Standardy Uniksa w świecie otwartych źródeł 402
17.3. IETF i Proces Standaryzacji RFC 403
17.4. Specyfikacja to DNA, kod to RNA 406
17.5. Programowanie ukierunkowane na przenośność 408
17.5.1. Przenośność i wybór języka 409
17.5.2. Omijanie zależności od systemu 412
17.5.3. Narzędzia umożliwiające przenośność 413
17.6. Internacjonalizacja 413
17.7. Przenośność, otwarte standardy i otwarte źródła 414

Rozdział 18. Dokumentacja: Objaśnianie kodu w świecie WWW 417
18.1. Koncepcje dokumentacji 418
18.2. Styl Uniksa 420
18.2.1. Skłonność do wielkich dokumentów 420
18.2.2. Styl kulturowy 421
18.3. Zwierzyniec uniksowych formatów dokumentacji 422
18.3.1. troff i narzędzia z Warsztatu Dokumentatora 422
18.3.2. $\TeX$ 424
18.3.3. Texinfo 425
18.3.4. POD 425
18.3.5. HTML 425
18.3.6. DocBook 426
18.4. Istniejący chaos i możliwe rozwiązania 426
18.5. DocBook 427
18.5.1. Definicje typu dokumentu 427
18.5.2. Inne definicje DTD 428
18.5.3. Łańcuch narzędzi DocBook 429
18.5.4. Narzędzia do migracji 431
18.5.5. Narzędzia do edycji 432
18.5.6. Pokrewne standardy i praktyki 432
18.5.7. SGML 433
18.5.8. Bibliografia formatu XML-DocBook 433
18.6. Najlepsze praktyki pisania dokumentacji uniksowej 434

Rozdział 19. Otwarte źródła: Programowanie w nowej społeczności Uniksa 437
19.1. Unix i otwarte źródła 438
19.2. Najlepsze metody pracy z twórcami otwartego oprogramowania 440
19.2.1. Dobre praktyki korygowania programów 440
19.2.2. Dobre praktyki nazywania projektów i archiwów 444
19.2.3. Dobre praktyki rozwoju projektu 447
19.2.4. Dobre praktyki tworzenia dystrybucji 450
19.2.5. Dobre praktyki komunikacji 454
19.3. Logika licencji: jak wybrać 456
19.4. Dlaczego należy stosować standardowe licencje 457
19.5. Zróżnicowanie licencji otwartego źródła 457
19.5.1. Licencja MIT lub X Consortium 457
19.5.2. Klasyczna licencja BSD 458
19.5.3. Licencja Artistic 458
19.5.4. Licencja GPL 458
19.5.5. Licencja Mozilla Public License 459

Rozdział 20. Przyszłość: Zagrożenia i możliwości 461
20.1. Zasadność i przypadki w tradycji Uniksa 461
20.2. Plan 9: Tak wyglądała przyszłość 464
20.3. Problemy w konstrukcji Uniksa 466
20.3.1. Plik w Uniksie jest tylko wielkim workiem bajtów 466
20.3.2. Unix słabo obsługuje graficzne interfejsy użytkownika 468
20.3.3. Plik usunięty na zawsze 469
20.3.4. Unix zakłada istnienie statycznego systemu plików 469
20.3.5. Projekt kontroli zadań jest zupełnym nieporozumieniem 469
20.3.6. API Uniksa nie stosuje wyjątków 470
20.3.7. Za wywołania ioctl(2) i fcntl(2) należy się wstydzić 471
20.3.8. Model bezpieczeństwa w Uniksie może być zbyt prosty 472
20.3.9. W Uniksie jest byt wiele różnych rodzajów nazw 472
20.3.10. Systemy plików można by uznać za szkodliwe 472
20.3.11. W kierunku Globalnej Przestrzeni Adresowej Internetu 473
20.4. Problemy w środowisku Uniksa 473
20.5. Problemy w kulturze Uniksa 475
20.6. Źródła nadziei 478

Dodatek A Słownik skrótów 479

Dodatek B Nawiązania 483

Dodatek C Współpracownicy 493

Dodatek D Korzeń bez korzenia: uniksowe koany Mistrza Foo 495
Wstęp redaktorski 495
Mistrz Foo i dziesięć tysięcy linii 496
Mistrz Foo i Script Kiddie 497
Opowieści Mistrza Foo o dwóch ścieżkach 498
Mistrz Foo i metodolog 499
Opowieści Mistrza Foo o graficznych interfejsach użytkownika 500
Mistrz Foo i fanatyk Uniksa 500
Opowieści Mistrza Foo o naturze Uniksa 501
Mistrz Foo i Użytkownik 502

Skorowidz 503

Rozdział 16.
Ponowne wykorzystanie: nie wyważajmy otwartych drzwi

Gdy wielki człowiek powstrzymuje się od działania,
jego siłę można poczuć z odległości tysięcy mil.

— *Tao Te Ching (popularne, choć nieprawidłowe tłumaczenie)*

Niechęć do niepotrzebnej pracy jest wielką zaletą programistów. Jeżeli chiński mędrzec Lao-cy żyłby dzisiaj i nauczał drogi tao, jego słowa zostałby prawdopodobnie (również błędnie) przetłumaczone jako: „Gdy wielki programista powstrzymuje się od pisania, jego siłę można poczuć z odległości tysięcy mil”. W rzeczywistości ostatnie przekłady sugerują jednak, że chiński zwrot *wu-tei*, który tradycyjnie był tłumaczony jako „bezczynność” lub „powstrzymywanie się od działania”, powinien być oddany jako „minimalne działania”, „najefektywniejsze operacje” albo „działania w zgodzie z prawami natury”. Taka wersja doskonale opisuje dobre praktyki inżynieryjne.

Trzeba pamiętać o Regule Ekonomii. Wyważanie otwartych drzwi w każdym nowym projekcie jest marnotrawstwem. Czas poświęcony na myślenie jest o wiele cenniejszy niż pozostałe składowe procesu rozwoju oprogramowania, w związku z czym powinien być przeznaczany na zmaganie się z nowymi problemami, a nie na rozgrzebywanie starych, dla których rozwiązania już dawno wymyślono. Takie nastawienie sprawdza się doskonale zarówno w zakresie „miękkiego” rozwoju zasobów ludzkich, jak również w zakresie „twardych” zasad opłacalności inwestycji w rozwój oprogramowania.

> Ponowne wymyślanie koła nie jest złe tylko z powodu marnotrawienia cennego czasu, ale dlatego, że na nowo wymyślone koła często okazują się kwadratowe. Rodzą się wówczas nieodparte ciągoty do różnych oszczędności wynikających z tworzenia prymitywnych i nieprzemyślanych projektów, które jednak na dłuższą metę powodują koszty przewyższające uzyskane oszczędności.
>
> — Henry Spencer

Najskuteczniejszym sposobem na uniknięcie ponownego wymyślania koła jest pożyczenie czyjegoś projektu i implementacji, czyli ponowne wykorzystanie istniejącego kodu.

Unix pozwala na ponowne wykorzystanie praktycznie wszystkiego, począwszy od pojedynczych bibliotek, a kończąc na całych programach, które można dowolnie łączyć za pomocą skryptów. Ponowne wykorzystanie kodu jest jednym z najważniejszych wyróżników programistów uniksowych. Doświadczenie wynikające z korzystania z Uniksa pozwala wytworzyć w sobie nawyk prób prototypowania różnych rozwiązań, przez łączenie istniejących komponentów z minimalizacją tworzenia nowych wynalazków. Daje to lepsze rezultaty niż pochopne pisanie autonomicznego kodu, który zostanie użyty tylko w jednym miejscu.

Ponowne wykorzystanie kodu należy do najbardziej podstawowych elementów rozwoju oprogramowania. Niestety wielu programistów przystępujących się do społeczności uniksowej, którzy zdobywali doświadczenie w innych systemach operacyjnych, nigdy nie wyrobili w sobie nawyku ponownego wykorzystania istniejących elementów. Wszędzie kwitnie marnotrawstwo czasu i podwójna praca, mimo że szkodzi to interesom zarówno płacących za kod, jak i twórców kodu. Poznanie powodów utrzymywania się tak dziwacznych tendencji jest pierwszym krokiem do ich zmiany.

16.1. Opowieść o Janie Nowicjuszu

Dlaczego programiści ciągle wymyślają koło na nowo? Powodów jest wiele, zaczynając od najprostszych technicznych potknięć, a kończąc na psychologii programisty i ekonomii systemu produkcji oprogramowania. Efekt takiego powszechnego marnotrawienia czasu programistów daje się odczuć na tych wszystkich płaszczyznach.

Wyobraźmy sobie pierwsze doświadczenia w normalnej pracy pewnego programisty, absolwenta uczelni, Jana Nowicjusza. Załóżmy, że w szkole wpojono mu zalety ponownego wykorzystania kodu, i teraz pełen zapału pragnie wykorzystać te wiadomości.

Pierwszym zadaniem Nowicjusza jest praca w zespole tworzącym pewną wielką aplikację. Na potrzeby tego przykładu załóżmy, że jest to środowisko graficzne, które ma pomagać użytkownikom w tworzeniu zapytań i nawigowaniu w dużej bazie danych. Menadżerowie projektu zebrali kolekcję narzędzi i komponentów, które uznali za właściwe do wykonania zadania. W tej kolekcji znajduje się nie tylko wybrany język programowania, ale również wiele bibliotek.

Zebrane biblioteki są kluczowym elementem projektu. Zawierają one wiele ważnych usług — od kontrolek graficznych i połączeń sieciowych, aż do całych podsystemów, takich jak pomoc kontekstowa. Brak tych usług spowodowałby konieczność tworzenia sporej ilości dodatkowego kodu, co w znacznym stopniu wpłynęłoby na budżet projektu i datę wydania produktu.

Właśnie data wydania mocno niepokoi Nowicjusza. Być może nie ma doświadczenia, ale czytał *Dilberta* i słyszał kilka kombatanckich opowieści doświadczonych programistów. Dobrze wie, że zarząd ma tendencje do wyznaczania, delikatnie mówiąc

„agresywnych” terminów. Być może czytał nawet książkę Eda Yourdona *Death March* [Yourdon], który już w roku 1996 zauważył, że budżet większości projektów jest o co najmniej 50% za mały, zarówno jeżeli chodzi o środki, jak i czas, a tendencja do takiego ograniczania budżetów sprawia, że w przyszłości będzie jeszcze gorzej.

Jednak Nowicjusz jest inteligentny i pełen energii. Dochodzi do wniosku, że największą szansę na sukces daje mu szybkie nauczenie się jak najskuteczniejszego wykorzystania przekazanych mu narzędzi i bibliotek. Rozgrzewa palce i rzuca się w wir pracy… w efekcie trafia do piekła.

Wszystko, za co się zabierze, jest trudniejsze i trwa dłużej niż tego oczekiwał. Wygląda na to, że komponenty, których używa, mają wiele przypadków brzegowych, w których zachowują się nieprawidłowo, a czasem wręcz niszczycielsko. Takich przypadków unikano w aplikacjach przykładowych tych komponentów, a kod Nowicjusza tworzy je co krok. Często zastanawia się, co mieli na myśli twórcy biblioteki, ale nie może znaleźć właściwej odpowiedzi, ponieważ biblioteka nie jest właściwie udokumentowana. Dokumentację często bowiem tworzą pracownicy techniczni niebędący programistami i nie myślący jak programiści. Niestety nie można poznać sposobu działania biblioteki przez przeczytanie jej kodu, ponieważ jest ona skompilowanym kawałkiem kodu obiektowego obwarowanego licencjami.

W efekcie Nowicjusz musi tworzyć coraz bardziej złożone obejścia problemów tworzonych przez biblioteki, co powoduje, że korzyści z ich stosowania stają cię coraz mniej widoczne. Ponadto tworzone obejścia powodują, że kod wygląda coraz paskudniej. Być może trafi na kilka miejsc w bibliotece, w których nie da się zmusić jej do wykonania bardzo ważnego zadania (zgodnie ze specyfikacją powinna je wykonywać). Czasami jest całkiem pewny, że istnieje jakiś sposób na zmuszenie tej czarnej skrzynki do działania, ale nie jest w stanie zgadnąć jaki to sposób.

Nowicjusz zauważa, że im bardziej obciąża zadaniami bibliotekę, tym mocniej rośnie czas debugowania kodu — nawet wykładniczo. Cały tworzony przez niego kod prześladuje plaga błędów i wycieków pamięci; ich ślady prowadzą do bibliotek, których kodu nie może zobaczyć ani zmodyfikować. Domyśla się, że wiele tych śladów powraca z biblioteki do jego własnego kodu, ale bez źródeł nie jest w stanie określić, do której jego części.

Nowicjusz jest coraz bardziej sfrustrowany. W szkole słyszał, że w normalnej pracy napisanie stu linii ukończonego kodu w ciągu tygodnia uznawane jest za niezłą wydajność. Wtedy śmiał się z tego, ponieważ w czasie tworzenia projektów szkolnych i swojego prywatnego programowania był wielokrotnie bardziej wydajny. Teraz już nie wydaje mu się to takie zabawne. Nie zmaga się już tylko z własnym brakiem doświadczenia, ale także z masą problemów powstałych w wyniku beztroski i niekompetencji innych. Są to problemy, których nie da się rozwiązać, ale można jedynie tworzyć ich obejścia.

Harmonogram projektu zaczyna się opóźniać. Nowicjusz marzył o pracy architekta, a tymczasem stał się zwykłym murarzem, który dostał niepasujące do siebie cegły, rozpadające się pod naciskiem. Ale menadżerowie nie chcą słyszeć od młodego programisty żadnych wymówek. Zbyt głośne narzekanie na niską jakość komponentów

najprawdopodobniej wpędzi go w konflikty ze starszymi menadżerami, którzy je wybierali. Nawet jeżeli mógłby wygrać tę bitwę, to zmiana komponentów byłaby bardzo trudna do wykonania — wiązałoby się to z pracą zespołu prawników przeglądających szczegóły licencji.

Istnieje tylko niewielka szansa na to, że Nowicjusz doczeka się poprawienia błędów w bibliotekach jeszcze w czasie tworzenia projektu. Przy odrobinie zastanowienia może on zauważyć, że prawidłowo działający kod biblioteki nie zaprząta jego uwagi w takim stopniu, jak znajdujące się w niej błędy i niedociągnięcia. Bardzo chciałby usiąść i porozmawiać z twórcami biblioteki. Podejrzewa, że nie są oni tak skończonymi idiotami, jak by wskazywał kod biblioteki, ale zwykłymi programistami, którzy podobnie jak on zmagają się z przeciwnościami próbując właściwie wykonać swoją pracę. Niestety nie jest w stanie sprawdzić, kim są ci programiści, a nawet jeżeli by się dowiedział, firma dla której pracują, prawdopodobnie nie zgodziłaby się na taką rozmowę.

W akcie desperacji Nowicjusz zaczyna tworzyć własne cegły, symulując niewłaściwie działające elementy bibliotek własnym, lepiej sprawującym się kodem, tworzonym praktycznie od zera. Dzięki temu, że Nowicjusz ma w głowie kompletny model tych implementacji, powstający w ten sposób kod działa znacznie lepiej i jest łatwiejszy w debugowaniu niż połączenie nieprzezroczystych bibliotek i tworzonych do nich obejść.

Nowicjusz czegoś się nauczył: im mniej polega na kodzie tworzonym przez innych ludzi, tym więcej kodu może napisać sam. To bardzo mile łechce jego ego. Podobnie jak wielu innych młodych programistów, w podświadomości sądzi, że jest znacznie mądrzejszy od innych, a jego doświadczenia zdają się to potwierdzać. Zaczyna więc tworzyć swój własny zestaw narzędzi, lepiej dopasowanych do jego wymagań.

Niestety, w ten sposób uzyskuje tylko krótkoterminową i lokalną poprawę, która w efekcie spowoduje długoterminowe problemy. Jest w stanie napisać większą ilość linii kodu, ma on jednak o wiele niższą względną wartość w stosunku do kodu, który mógłby napisać prawidłowo wykorzystując biblioteki. Więcej kodu niekoniecznie oznacza lepszy kod, szczególnie jeżeli jest to kod niskiego poziomu, w znacznym stopniu związany z ponownym wymyślaniem koła.

Nowicjusza czeka jeszcze przynajmniej jedno nieprzyjemne doświadczenie, związane ze zmianą pracy. Najprawdopodobniej okaże się, że nie będzie mógł zabrać ze sobą napisanych przez siebie narzędzi. Jeżeli opuści firmę zabierając ze sobą kod, który napisał w czasie pracy dla tej firmy, jego byli pracodawcy mogą potraktować to jako kradzież własności intelektualnej. Gdy dowiedzą się o tym jego nowi pracodawcy, raczej nie będą uszczęśliwieni, gdy Nowicjusz przyzna się, że używa tego kodu w ich firmie.

Nawet jeżeli Nowicjuszowi uda się przemycić swoje narzędzia do nowej pracy, może się okazać, że będą one tam zupełnie bezużyteczne. Nowi pracodawcy mogą używać zupełnie innych narzędzi, języków i bibliotek chronionych prawami własności. Całkiem prawdopodobne jest, że przy każdym nowym projekcie będzie musiał się nauczyć zupełnie nowego zbioru technik i na nowo wyważać otwarte drzwi.

W ten sposób programiści skutecznie oduczani są od ponownego wykorzystywania kodu (a także innych dobrych praktyk, takich jak modułowość i przezroczystość) przez połączenie problemów technicznych, barier tworzonych przez własność intelektualną,

politykę i potrzeby własnego ego. Spróbujmy takiego Jana Nowicjusza pomnożyć razy sto tysięcy, dodać mu kilka dziesięcioleci i dorzucić mu nieco cynizmu i przyzwyczajenia do takiego systemu. Otrzymamy obraz większej części przemysłu oprogramowania i receptę na wielkie marnotrawienie czasu, kapitału i ludzkich umiejętności. Trzeba je tylko powiększyć o taktyki kontroli rynku stosowane przez dostawców, niekompetentne zarządzanie, niemożliwe do utrzymania terminy i wszystkie inne czynniki sprawiające, że tak trudno jest prawidłowo wykonywać swoją pracę.

Większość z nich odzwierciedla zawodowa kultura wynikająca z doświadczeń Jana Nowicjusza. Firmy tworzące oprogramowanie są nękane kompleksem **Nie Wymyślonego Tutaj**. Ich zachowanie w stosunku do ponownego wykorzystania kodu będzie nader ambiwalentne. W celu dotrzymania terminów będą wymuszały na swoich programistach stosowanie nieodpowiednich, ale szeroko reklamowanych komponentów. Jednocześnie będą odmawiały wykorzystania kodu napisanego i przetestowanego przez ich własnych programistów. Będą masowo produkować oprogramowanie o dość przypadkowym i powtarzającym się kodzie, tworzonym przez programistów zdających sobie sprawę z tego, że wynik ich pracy okaże się wielkim śmietniskiem, ale pogodzonych z myślą, że nie będą w stanie poprawić niczego poza tym, co sami napisali.

W takiej kulturze ponowne wykorzystanie kodu zostanie zastąpione dogmatem, że raz opłaconego kodu nie wolno wyrzucić, ale musi on być poprawiany i łatany, mimo że wszyscy dobrze wiedzą, że lepiej byłoby napisać go od nowa. Produkty tworzone w takiej kulturze stają się z czasem coraz bardziej rozdęte i zawierają coraz więcej błędów, mimo że każdy, kto pracuje przy ich tworzeniu, stara się dać z siebie wszystko.

16.2. Przezroczystość jako klucz do ponownego użycia kodu

Historię Jana Nowicjusza przedstawiliśmy kilku doświadczonym programistom. Jeżeli Czytelnik również jest programistą, powinien zareagować mniej więcej tak jak oni: pomrukiem rozpoznania. Jeżeli jednak nie jest programistą a menadżerem zarządzającym programistami — mamy nadzieję, że dzięki tej opowieści doznał oświecenia. Historia miała ilustrować sposób, w jaki różne naciski przeciwko ponownemu wykorzystaniu kodu wzmacniają się wzajemnie, tworząc razem problem o rozmiarach przekraczających wszelkie wyobrażenia.

Większość z nas jest bardzo przyzwyczajona do podstawowych założeń przemysłu oprogramowania, zgodnie z którymi oddzielenie podstawowych przyczyn tego problemu od innych przypadków może wymagać wyjątkowego wysiłku umysłowego. A jednak okazuje się, że nie są one bardzo skomplikowane.

Powodem większości kłopotów Jana Nowicjusza (a także powodowane przez nie większe problemy z jakością) jest przezroczystość, a raczej jej brak. Nie da się naprawić tego, czego wnętrza nie można zobaczyć. Tak naprawdę, w przypadku każdego nietrywialnego API, tego, czego nie można zobaczyć od środka, nie da się nawet *używać*. Dokumentacja jest z zasady niewystarczająca, nie da się w niej umieścić wszystkich niuansów kodu, który opisuje.

W rozdziale 6. rozważaliśmy zbawienny wpływ przezroczystości na jakość oprogramowania. Komponenty składające się jedynie z kodu obiektowego niszczą przezroczystość oprogramowania. Z drugiej strony, jeżeli kod, którego próbujemy użyć ponownie, można bez problemów obejrzeć i zmodyfikować, bardzo zmniejsza się szansa na to, że zacznie on sprawiać niemiłe niespodzianki. Dobrze komentowane źródła kodu są jego najdoskonalszą dokumentacją. Błędy w kodzie źródłowym można poprawić. Źródła można konfigurować i kompilować specjalnie do debugowania, co znacznie ułatwia analizę zachowania kodu w niejasnych przypadkach. A jeżeli zaistnieje taka potrzeba, te zachowania można oczywiście zmieniać.

Istnieje jeszcze jeden ważny powód, dla którego należy żądać kodu źródłowego. Lekcja, którą programiści uniksowi przyswoili sobie na przestrzeni dekad, mówi, że kod źródłowy przetrwa, a kod obiektowy — nie. Zmieniają się platformy sprzętowe, zmieniają się różnego rodzaju biblioteki, systemy operacyjne wytwarzają nowe API i odsuwają stare w cień. Wszystko się zmienia, a nieprzezroczyste binaria nie są w stanie dostosować się do tych zmian. Są bardzo kruche, nie dają się łatwo przenosić do nowszych warunków, ale wymagają stosowania bardzo grubych i podatnych na błędy warstw emulacyjnych. Zamykają użytkowników w kręgu założeń poczynionych przez ich twórców. Kody źródłowe są konieczne nawet wtedy, gdy nie istnieje potrzeba ani chęć zmiany oprogramowania, bo do uruchomienia ich w nowych środowiskach wymagane jest przekompilowanie.

Waga przezroczystości kodu i problem z jego przenośnością są wystarczającymi powodami, dla których należy żądać umożliwienia przeglądania i modyfikowania stosowanego kodu[1]. Przytoczona argumentacja nie obejmuje tego, co nazywany dzisiaj „otwartymi źródłami”, ponieważ ten termin ma znacznie szersze implikacje niż prosty wymóg przezroczystości kodu i możliwości wglądu do niego.

16.3. Od ponownego wykorzystania do otwartych źródeł

We wczesnych latach Uniksa, komponenty systemu operacyjnego, związane z nimi biblioteki i narzędzia, były przekazywane w postaci kodu źródłowego. Taka otwartość była jednym z ważnych elementów kultury Uniksa. W rozdziale 2. opisywaliśmy, że gdy w roku 1984 ta tradycja została przerwana, Unix stracił wiele ze swojego początkowego impetu. Opisywaliśmy także, jak powstanie narzędzi GNU i Linuksa dziesięć lat później spowodowało ponowne odkrycie wartości otwartego źródła.

Dziś znowu otwarty kod jest jednym z najpotężniejszych narzędzi każdego programisty uniksowego. Mimo że dokładna koncepcja open source i najczęściej stosowane licencje tego typu są o dziesięciolecia młodsze od samego Uniksa, bardzo ważne jest zrozumienie obu tych idei, tak aby w kulturze dzisiejszego Uniksa tworzyć doskonałe oprogramowanie.

---

[1] NASA konsekwentnie tworzy oprogramowanie, które ma działać przez dziesięciolecia, dlatego zawsze nalega na udostępnianie kodu źródłowego całości oprogramowania awioniki.

Otwarte źródła wiążą się z ponownym wykorzystaniem kodu podobnie jak miłość romantyczna łączy się z reprodukcją seksualną. Tę pierwszą można opisać za pomocą tej drugiej, ale w ten sposób ryzykuje się pominięcie wielu elementów, które sprawiają, że ta pierwsza jest tak interesująca. Otwarte źródła nie są jedynie taktyką pozwalającą na ponowne wykorzystanie kodu w czasie produkcji oprogramowania. Są raczej fenomenem i swego rodzaju kontraktem zawartym między twórcami i użytkownikami, mającym na celu zabezpieczenie pewnych korzyści wynikających z przezroczystości. Istnieje co najmniej kilka sposobów na zrozumienie tak określonego pojęcia otwartych źródeł.

Wcześniej zajmowaliśmy się opisem historycznym, skupiającym się tylko na kulturowych związkach łączących Uniksa z otwartymi źródłami. Taktyki i tradycje rozwoju otwartych źródeł omówimy w rozdziale 19. W czasie dyskusji na temat teorii i praktyki ponownego wykorzystania kodu warto potraktować otwarte źródła jako bezpośrednią odpowiedź na problemy, tak dramatycznie przedstawione w opowieści o Janie Nowicjuszu.

Programiści pragnęliby, żeby używany przez nich kod był przezroczysty. Co więcej, nie chcą przy zmianie miejsca pracy tracić swoich narzędzi i doświadczenia. Nie chcą być ciągle ofiarami, mają dość frustracji powodowanych przez marne narzędzia, ograniczenia związane z własnością intelektualną i konieczność nieustannego wymyślania koła na nowo.

Są to przyczyny stosowania otwartych źródeł, wynikające z bolesnych doświadczeń Jana Nowicjusza z ponownym wykorzystaniem kodu. Pewną rolę odgrywają tu też wymagania własnego ego. Dodają nieco emocji do dyskusji dotyczącej najlepszych praktyk inżynieryjnych, które bez tego byłyby jałowe i nudne. Twórcy oprogramowania, podobnie jak wielu innych rzemieślników, chcą być artystami w swoim fachu. Wykazują ten sam co artyści zapał i potrzeby, w tym również pragnienie posiadania publiczności. Nie chcą jedynie ponownie wykorzystywać swojego kodu — chcą żeby ich kod był ponownie wykorzystywany przez innych. Jest w tym pewien imperatyw, wykraczający poza poszukiwania doraźnych korzyści ekonomicznych, którego nie jest w stanie zaspokoić produkcja zamkniętego oprogramowania.

Otwarte źródło jest swego rodzaju wyprzedzającym uderzeniem na te wszystkie problemy. Jeżeli wszystkie problemy Jana Nowicjusza z ponownym wykorzystaniem kodu, wynikały z nieprzezroczystości zamkniętego kodu, to znaczy że należy zmienić wszystkie dotychczasowe założenia związane z tworzeniem takiego kodu. Jeżeli problemy wynikają z terytorializmu wielkich korporacji, muszą być one atakowane lub omijane dopóty, dopóki same firmy nie zrozumieją jak samoniszczące są ich terytorialne odruchy. Otwarte źródła powstają wtedy, gdy idea ponownego wykorzystania kodu otrzyma własny sztandar i armię.

Od końca lat 90. nie ma większego sensu rekomendowanie strategii i taktyk ponownego wykorzystania kodu, nie wspominając o otwartych źródłach, związanych z nimi praktykach, licencjach i całej społeczności. Nawet jeżeli gdzie indziej te pojęcia można rozdzielać, w świecie Uniksa nierozerwalnie związały się ze sobą.

W pozostałej części tego rozdziału zajmować się będziemy różnymi kwestiami związanymi z ponownym wykorzystaniem kodu o otwartym źródle: ocenianiu, dokumentowaniu i licencjonowaniu. W rozdziale 19. dokładniej przyjrzymy się modelom rozwoju otwartego źródła, a także konwencjom, których należy przestrzegać w czasie udostępniania kodu innym.

16.4. Najlepsze rzeczy w życiu są otwarte

Z internetu można pobrać dosłownie terabajty źródeł uniksowych systemów, aplikacji, bibliotek, narzędzi do tworzenia GUI i sterowników urządzeń. Większość z nich można w ciągu kilku minut skompilować i uruchomić za pomocą standardowych narzędzi. Wystarczy zawierzyć mantrze: `./configure; make; make install`. Do wykonania instalacji zazwyczaj konieczne jest posiadanie uprawnień administratora.

Ludzie z poza świata Uniksa (zwłaszcza nietechniczni) mają tendencję do wydawania opinii, że otwarte (albo „wolne") oprogramowanie z całą pewnością jest gorsze od komercyjnego, jest marnie wykonane, zawodne i spowoduje więcej kłopotów niż będzie w stanie rozwiązać. Nie uwzględniają oni pewnej bardzo ważnej kwestii: otwarte oprogramowanie jest zwykle tworzone przez ludzi, którzy się nim przejmują, potrzebują go i sami go używają. Takie osoby publikując tworzone przez siebie oprogramowanie, stawiają wyzwanie swojej własnej reputacji. Zwykle nie tracą też czasu na spotkania, wsteczne zmiany projektów i całą dodatkową biurokrację. Z tego powodu są znacznie lepiej zmotywowani do pracy niż podobni do *Dilberta* niewolnicy zarobków, którzy próbują dotrzymać niemożliwych terminów pracując w kabinach wielkich korporacji.

Co więcej, społeczność użytkowników otwartych źródeł nie boi się wytykać znalezionych błędów, dlatego jakość takiego oprogramowania jest bardzo wysoka. Autorzy oddający do użytkowania prace niskiej jakości poddawani są naciskom mającym na celu poprawienie błędów lub wycofanie oprogramowania. Jeżeli zdecydują się na poprawianie, mogą liczyć na znaczącą pomoc doświadczonych kolegów. W efekcie dojrzałe pakiety o otwartych źródłach są zwykle znacznie wyższej jakości niż ich komercyjne odpowiedniki, a nierzadko przewyższają je pod względem funkcjonalności. Być może nie są tak doszlifowane, a ich dokumentacja tworzy wiele założeń dotyczących dotychczasowej wiedzy czytelnika, ale ich najważniejsze funkcje sprawują się zwykle doskonale.

Poza efektem reakcji użytkowników, istnieje jeszcze jeden powód, dla którego należy oczekiwać wyższej jakości otwartego oprogramowania. W świecie otwartych źródeł programiści nigdy nie są zmuszani terminami do przymknięcia oczu oddania programu do rozpowszechniania. Jedną z głównych konsekwencji wynikających z różnic między praktyką otwartych źródeł a innymi jest to, że w świecie open source wersja 1.0 oznacza gotowość programu do normalnego użytkowania. Tak naprawdę, już wersje 0.90 i wyższe, oznaczają że kod programu jest już praktycznie gotowy, ale jego autorzy nie chcą jeszcze ryzykować swojej reputacji.

Czytelnikom spoza świata Uniksa może się to wydawać niemożliwe. Jeżeli tak jest, proponuję zastanowić się nad jednym faktem: w nowoczesnych Uniksach kompilator C jest niemal zawsze produktem o otwartych źródłach. Kolekcja kompilatorów GCC Free Software Foundation jest tak doskonałym i dobrze udokumentowanym narzędziem, że praktycznie nie pozostawił on miejsca na rynku dla żadnego komercyjnego konkurenta. Wśród dostawców Uniksa stało się normą dostarczanie do swoich platform kompilatorów GCC, zamiast tworzenia własnych rozwiązań.

Sposobem oceny pakietów o otwartym źródle jest przeczytanie ich dokumentacji i przejrzenie części kodu źródłowego. Jeżeli sprawia wrażenie napisanego przez kompetentne osoby i właściwie udokumentowanego, można zacząć nabierać do niego zaufania. Jeżeli dodatkowo są dowody na to, że pakiet był używany od jakiegoś czasu i zebrał wiele opinii od użytkowników, można właściwie założyć, że jest on niezawodny (ale i tak lepiej go przetestować).

Dobrym wskaźnikiem dojrzałości projektu i ilości uwag od użytkowników jest liczba osób wymienianych w pliku *README*, a także w plikach historii dystrybucji. Podziękowania dla wielu osób za przesłanie poprawek i łatek są znakiem zarówno sporego zainteresowania użytkowników, jak i sumienności osoby obsługującej projekt, odpowiedzialnej za korespondencję i wprowadzanie poprawek. Jest to też dobry sygnał, że nawet jeżeli pierwsze wersje kodu były podobne do pola zaminowanego błędami, to biegające po nim ostatnio stada nie powodowały bardzo wielu wybuchów.

Dobrym znakiem jest też posiadanie przez projekt własnej strony WWW, dostępnej w sieci listy często zadawanych pytań (*FAQ* — ang. *Frequently Asked Questions*) i listy mailingowej albo grupy dyskusyjnej. To wszystko oznacza, że wokół projektu wyrosła spora społeczność zainteresowana jego rozwojem. Jeżeli strona WWW jest często aktualizowana i zawiera długą listę serwerów lustrzanych (tzw. mirrorów), jest to nieomylnym znakiem tego, że projekt posiada bardzo aktywną grupą użytkowników. Pakiety będące niewypałami nie mają szans na takie zainteresowanie, ponieważ nie są w stanie za nie właściwie odpłacić.

Istnienie wersji na wielu platformach jest również ważnym wskaźnikiem bardzo zróżnicowanej grupy użytkowników. Strony projektów bardzo wyraźnie zaznaczają powstawanie wersji na nowe platformy, ponieważ to oznacza większą wiarygodność projektu.

Poniżej podajmy kilka przykładów stron WWW związanych z wysokiej jakości oprogramowaniem o otwartym źródle:

- GIMP (*http://www.gimp.org/*)
- GNOME (*http://www.gnome.org/*)
- KDE (*http://www.kde.org/*)
- Python (*http://www.python.org/*)
- Jądro Linuksa (*http://www.kernel.org/*)
- PostgreSQL (*http://www.postgresql.org/*)
- XFree86 (*http://xfree86.org/*)
- InfoZip (*http://www.info-zip.org/pub/infozip/*)

Przeglądanie dystrybucji Linuksa jest kolejnym dobrym sposobem na znalezienie oprogramowania niezłej jakości. Twórcy dystrybucji Linuksa i innych Uniksów o otwartych źródłach wykonują wiele kontroli mających na celu sprawdzenie, które projekty są najlepsze w swoim rodzaju. Jeżeli projekt zostanie dodany do dystrybucji, to znaczy że z pewnością jest sporo wart. Osoby posiadające już Uniksa o otwartych źródłach mogą sprawdzić, czy używana przez nich dystrybucja zawiera oceniany właśnie program.

16.5. Gdzie szukać?

Ze względu na ogromną liczbę programów o otwartych źródłach dostępnych w świecie Uniksa bardzo opłacalna jest umiejętność znajdowania właściwego kodu do ponownego wykorzystania — o wiele bardziej niż w innych systemach operacyjnych. Tego typu kody można znaleźć w bardzo różnych postaciach: pojedynczych kawałków kodu z przykładami, bibliotek lub gotowych narzędzi, z których można korzystać za pomocą skryptów. W Uniksie ponowne wykorzystywanie kodu rzadko odbywa się na zasadzie wytnij-i-wklej. Jeżeli ktoś w ten sposób wykorzystuje kod, z całą pewnością istnieje jakiś znacznie lepsza metoda, której nie znalazł. W związku z tym jedną z najużyteczniejszych umiejętności kultywowanych w Uniksie jest poznawanie wszystkich sposobów na łączenie kodu ze sobą, co umożliwia zastosowanie Reguły Kompozycji.

Szukanie kodu do ponownego wykorzystania najlepiej rozpocząć pod własnym nosem. Unix zawsze posiadał bogaty zbiór narzędzi i bibliotek gotowych do ponownego wykorzystania. Nowoczesne Uniksy, takie jak aktualne dystrybucje Linuksa, zawierają tysiące programów, skryptów i bibliotek, które można użyć. Zwykłe przeszukanie podręczników za pomocą polecenia `man -k`, uzupełnionego o kilka słów kluczowych, często daje bardzo ciekawe wyniki.

Poznawanie choć części tego bogactwa zasobów można rozpocząć od stron WWW serwisów SourceForge, ibiblio i Freshmeat.net. W momencie ukazania się tej książki mogą istnieć już inne, równie ważne serwisy, ale wszystkie z wymienionych przez wiele lat posiadały wysoką wartość i wielu użytkowników, dlatego można zakładać, że będą istnieć nadal.

SourceForge (*http://www.sourceforge.net/*) jest przykładem oprogramowania zaprojektowanego specjalnie do wspierania projektów rozwijanych przez wielu programistów, połączonego z serwisami pozwalającymi na pełne zarządzanie danym projektem. Jest to nie tylko archiwum, ale również darmowy serwis pozwalający na przechowywanie i rozwój oprogramowania, który w połowie roku 2003 może pochwalić się tytułem światowego centrum środowiska otwartych źródeł.

Linuksowe archiwa w serwisie ibiblio (*http://www.ibiblio.org/*) do czasu powstania serwisu Sourceforge były uznawane za największe na świecie. Są one pasywnymi archiwami, w których można umieszczać tylko gotowe do publikacji pakiety. Serwis ten posiada lepszy interfejs WWW niż wiele innych pasywnych archiwów (program tworzący interfejs WWW dla ibiblio był opisywany w jednym ze studiów rozdziału 14.). Ibiblio jest też główną stroną Projektu Dokumentacji Linuksa (LDP — Linux Documentation Project), na której znajduje się wiele doskonałych dokumentów wykorzystywanych przez użytkowników i programistów uniksowych.

Freshmeat (*http://www.freshmeat.net/*) jest systemem przeznaczonym do umieszczania ogłoszeń o wydaniach nowego oprogramowania i nowszych wersjach starszych programów. Pozwala użytkownikom dodawać do ogłoszeń recenzje programów.

Te trzy serwisy zawierają kod pisany w wielu językach, jednak ogromną część ich zawartości tworzą języki C i C++. W rozdziale 14. wspomnieliśmy też o istnieniu stron specjalizujących się w obsłudze programów tworzonych w różnych językach interpretowanych.

Archiwum CPAN jest główną składnicą przydatnego kodu w Perlu. Można na nie łatwo trafić ze strony domowej języka (*http://www.perl.com/perl*).

Na głównej stronie języka Python (*http://www.python.org/*) znajduje się archiwum dokumentacji i oprogramowania związanego z tym językiem, nazywane Python Software Activity.

Na stronie apletów Javy (*http://java.sun.com/applets/*) udostępniono wiele apletów, a także odsyłaczy do innych stron zawierających darmowe oprogramowanie napisane w tym języku.

Dla programisty uniksowego jednym z najlepszych sposobów na zainwestowanie czasu jest przeznaczenie go na przeglądanie podanych wyżej stron i poznawanie kodów udostępnionych przez innych. W ten sposób można sobie oszczędzić wiele czasu kodowania!

Dobrym pomysłem jest przejrzenie metadanych pakietu, ale nie należy na tym poprzestawać. Przejrzenie również kodu pozwala lepiej uchwycić operacje przez niego wykonywane, a w efekcie lepiej go wykorzystywać.

Mówiąc bardziej ogólnie, czytanie kodu jest inwestycją na przyszłość. Można się w ten sposób wiele nauczyć — nowych technik, nowych sposobów dzielenia problemów, różnych styli i rozwiązań. Możliwość wykorzystania kodu i uczenia się z niego jest wielką nagrodą za poświęcony czas. Nawet jeżeli nie wykorzystamy technik stosowanych w analizowanym kodzie, to nowe spojrzenie na problem, uzyskane dzięki zapoznaniu się z pomysłami innych osób, może okazać się pomocne w wymyślaniu własnego rozwiązania.

Czytaj przed pisaniem. Dobrze jest wytworzyć w sobie nawyk czytania kodu. Rzadko spotyka się zupełnie nowe problemy, dlatego prawie zawsze możliwe jest znalezienie kodu, który może być dobrym punktem startowym do tworzenia własnego rozwiązania. Nawet jeżeli rozwiązywany problem jest zupełną nowością, jest całkiem możliwe, że jest on blisko powiązany z innym problemem, który już ktoś kiedyś rozwiązał. W związku z tym rozwiązanie własnego problemu może ściśle wiązać się z istniejącym już kodem.

16.6. Kwestie związane z używaniem otwartego oprogramowania

Można wymienić trzy ważne sprawy związane z używaniem otwartego oprogramowania: jakość, dokumentacja i warunki licencji. Jak już wspomniano, wystarczy nieco poćwiczyć umiejętność prawidłowej oceny różnych alternatyw, a z całą pewnością znajdzie się jedno lub nawet kilka rozwiązań niezłej jakości.

Dokumentacja jest często znacznie większym problemem. Wiele doskonałych pakietów o otwartym źródle jest mniej użytecznych niż teoretycznie powinny być, ponieważ są słabo udokumentowane. Tradycja Uniksa zachęca do tworzenia hierarchicznego stylu dokumentacji (może ona opisywać wszystkie funkcje pakietu), w którym zakłada się,

że czytelnik doskonale zna dziedzinę problemu i czyta dokumentację bardzo dokładnie. Istnieją dobre powody na tworzenie dokumentacji w takim stylu (będziemy o tym mówić w rozdziale 18.), ale tworzy on pewnego rodzaju barierę. Na szczęście umiejętności przyswajania sobie cennych informacji z dokumentacji można się nauczyć.

W sieci WWW warto zadać wyszukiwarce pytanie zawierające nazwę programu albo słowo kluczowe związane z tematem, uzupełnione słowem „HOWTO" lub „FAQ". Takie zapytania często zwracają dokumentację znacznie przydatniejszą dla nowych użytkowników niż polecenie *man*.

Najważniejszą kwestią wiążącą się z wykorzystywaniem otwartego oprogramowania (szczególnie jeżeli tworzony jest produkt komercyjny) jest zrozumienie zobowiązań, jakie licencja pakietu nakłada na użytkownika. Ten problem będziemy omawiać dokładniej w kolejnych dwóch podrozdziałach.

16.7. Licencje

Ze wszystkim co nie należy do dziedziny publicznego oprogramowania (ang. *public domain*) związane jest przynajmniej jedno prawo autorskie. Według prawa Stanów Zjednoczonych autorzy prac posiadają prawa autorskie do nich, nawet jeżeli wyraźnie tego nie zaznaczą[2].

Wskazanie, kto konkretnie posiada prawa autorskie może być bardzo skomplikowane, szczególnie w przypadku, gdy nad produktem pracowało wiele osób. Dlatego tak ważną rolę odgrywają licencje. Mogą one zezwolić takie wykorzystanie kodu, które zgodnie z prawem autorskim byłoby niemożliwe, a prawidłowo skonstruowane mogą chronić użytkowników przed różnymi działaniami właścicieli praw autorskich.

W świecie oprogramowania komercyjnego licencje tworzone są tak, żeby chronić prawa własności. Są sposobem na przyznanie kilku praw użytkownikom i pozostawienie w ręku właściciela praw autorskich jak największej władzy. Właściciel praw autorskich jest w niej najważniejszy, a logika samej licencji tak restrykcyjna, że jej szczegóły techniczne nie są już najistotniejsze.

Jak będzie można niżej zobaczyć, właściciel praw autorskich zwykle używa swoich praw do ochrony licencji, dzięki czemu kod staje się dostępny z ograniczeniami, które właściciel ma zamiar zachować na stałe. W innych przypadkach właściciel rezerwuje sobie jedynie kilka praw, a większość wyborów pozostawia użytkownikom. W szczególności właściciel praw autorskich nie może zmieniać warunków licencji kopii już posiadanych przez użytkowników. Z tych powodów w oprogramowaniu o otwartym źródle właściciel praw autorskich nie jest ważny, ale za to warunki licencji mają bardzo duże znaczenie.

---

[2] Wg polskiego prawa również, kwestie tę reguluje ustawa o prawie autorskim i prawach pokrewnych z 4.02.1994 r. (Dz.U.00.80.904) w artykule 1 — *przyp. red.*

Zazwyczaj właściciel praw autorskich projektu jest osobą aktualnie prowadzącą projekt albo sponsorującą go organizacją. Przekazanie projektu nowemu prowadzącemu jest często sygnalizowane przez przeniesienie na niego praw autorskich. Nie jest to jednak regułą. Wiele projektów o otwartym źródle posiada kilku właścicieli praw autorskich i jak dotąd nie ma oznak, żeby prowadziło to do jakichkolwiek konfliktów z prawem. Niektóre projekty przyznają prawa autorskie fundacji Free Software Foundation, ponieważ z założenia działa ona w interesie obrony otwartego oprogramowania i posiada prawników zajmujących się tymi sprawami.

16.7.1. Co można uznać za otwarte oprogramowanie

Możemy wyróżnić kilka różnych praw, które mogą przekazywać licencje oprogramowania. Są to prawa do kopiowania, dystrybucji, użytkowania, modyfikacji do użytku własnego i w końcu prawo do dystrybuowania zmodyfikowanych kopii. Licencje mogą ograniczać te prawa albo wiązać je z pewnymi warunkami.

Definicja Otwartych Źródeł (ang. *Open Source Definition — OSD*) (*http://www.opensource.org/osd.html*) jest wynikiem wielu przemyśleń na temat tego, co czyni oprogramowanie „otwartym” albo według innej terminologii — „wolnym”. Ta definicja jest w środowisku otwartego źródła powszechnie uznawana za wyraz pewnego kontraktu zawartego między programistami otwartego źródła. Ograniczenia definicji wymuszają istnienie pewnych wymagań:

- Przyznanie nieograniczonego prawa do kopiowania.
- Przyznanie nieograniczonego prawa do redystrybucji w niezmienionej postaci.
- Przyznanie nieograniczonego prawa do wprowadzania modyfikacji na własne potrzeby.

Wskazówki zawarte w definicji zabraniają tworzenia ograniczeń w redystrybucji zmodyfikowanych binariów. To zaspokaja potrzeby dostawców oprogramowania, którzy muszą mieć możliwość dostarczania działającego kodu bez żadnych dodatkowych obciążeń. Autorzy mogą wymagać, żeby zmodyfikowane źródła były dostarczane jako zestaw oryginalnych źródeł i — dostarczonych w osobnych plikach — wprowadzonych w nich poprawek. W ten sposób powstaje możliwość poznania intencji autora i śladu zmian wprowadzonych przez inne osoby.

OSD jest prawną definicją certyfikatu „OSI Certified Open Source”, a jednocześnie najlepszą z powstałych dotychczas definicji otwartego oprogramowania. Z tą definicją zgodne są wszystkie standardowe licencje (MIT, BSD, Artistic, GPL/LGPL i MPL), choć niektóre z nich (na przykład GPL) zawierają pewne dodatkowe restrykcje, z którymi warto się zapoznać przed zastosowaniem licencji.

Należy zauważyć, że licencje zezwalające tylko na niekomercyjne wykorzystanie kodu nie mogą być zaliczane do licencji otwartego źródła, nawet jeżeli powstały w oparciu o licencję GPL lub inną standardową. Takie licencje dyskryminują pewne zawody, osoby i grupy, co jest niezgodne z piątą klauzulą definicji.

Piąta klauzula została dopisana po wielu latach bolesnych doświadczeń. Z licencjami zabraniającymi komercyjnego wykorzystania związany jest następujący problem: nikt nie jest w stanie wyraźnie określić, jakie wykorzystanie można zaliczyć do „komercyjnego". Sprzedaż oprogramowania jako pełnego produktu oczywiście jest takim zastosowaniem, ale co w przypadku, gdy było ono dostarczane za cenę zerową w połączeniu z innymi programami lub danymi, a opłata była pobierana za całą kolekcję? Jaką rolę odegrałby fakt, że program ma duże znaczenie dla funkcjonowania całej kolekcji?

Nikt nie jest w stanie odpowiedzieć na te pytania. Sam fakt, że takie licencje powodują niepewność co do praw dystrybutora jest bardzo ważnym zarzutem przeciwko nim. Jednym z celów definicji OSD jest zapewnienie, że osoby z łańcucha dystrybucji oprogramowania z nią zgodnego nie będą musiały konsultować się z prawnikami, żeby sprawdzić swoje uprawnienia. Dlatego zabronione są złożone restrykcje wymierzone przeciw osobom, grupom i zawodom, żeby osoby korzystające z oprogramowania nie natykały się na nieco różniące się (a czasami nawet wykluczające się) ograniczenia w zależności od tego co robią.

Takie obawy również nie są bezpodstawne. Ważną częścią łańcucha dystrybucji oprogramowania o otwartych źródłach są dystrybutorzy płyt CD-ROM, którzy łączą je w przydatne kolekcje, od prostych zbiorów antologii aż po uruchamialne płyty z systemami operacyjnymi. Dlatego ograniczenia, które bardzo komplikowałyby działania dystrybutorów płyt CD-ROM i innych osób próbujących komercyjnie rozprowadzać otwarte oprogramowanie, musiały zostać zakazane.

Z drugiej strony, definicja OSD nie ma wpływu na prawo w wielu krajach. Niektóre z nich posiadają prawo zabraniające eksportowania pewnych zastrzeżonych technologii do określonych „wrogich państw". Definicja OSD nie neguje takich ograniczeń, mówi jedynie o tym, że licencjodawcy nie mogą dodawać własnych.

16.7.2. Standardowe licencje otwartego oprogramowania

Poniżej podane są najczęściej wykorzystywane, standardowe licencje otwartego oprogramowania. Stosowane tutaj skróty są w powszechnym użyciu.

MIT — *http://www.opencource.org/licenses/mit-license.html*

Licencja konsorcjum MIT X (podobna jest do licencji BSD, ale nie zawiera „klauzuli reklamowej").

BSD — *http://www.opencource.org/licenses/bsd-license.html*

Licencja Uniwersytetu Kalifornijskiego w Berkley (stosowana do kodu systemu BSD).

Artistic License — *http://www.opencource.org/licenses/artistic-license.html*

Zawiera te same warunki co licencja Perl Artictic License.

GPL — *http://www.gnu.org/copyleft.html*

GNU General Public License.

LGPL — *http://www.gnu.org/copyleft.html*

Library (albo — „Lesser”) GPL – mniej restrykcyjna od GNU GPL, stosowana często do bibliotek.

MPL — *http://www.opencource.org/licenses/MPL-1.1.html*

Mozilla Public License.

W rozdziale 19. dokładniej opiszemy te licencje z punktu widzenia programisty. Na potrzeby tego rozdziału ważna jest jedynie różnica ich infekcyjności. Licencja jest infekcyjna, jeżeli wymaga, żeby każda praca wywodząca się z oprogramowania podlegającego tej licencji również podlegała jej warunkom.

Przy takich licencjach jedynym przypadkiem, w którym należy się nimi przejmować, jest włączenie otwartego oprogramowania do własnościowego produktu (ale nie zastosowania otwartego oprogramowania do budowania tego produktu). Jeżeli nie ma przeciwwskazań do umieszczenia w licencji produktu odpowiednich wpisów i podaniu w dokumentacji odnośników do wykorzystywanych źródeł, nie powinno być żadnych problemów nawet z bezpośrednim wykorzystaniem kodu. Oczywiście pod warunkiem, że licencja wykorzystywanego oprogramowania nie jest infekcyjna.

Licencja GPL jest najczęściej stosowana, ale jednocześnie jej infekcyjność wzbudza największe kontrowersje. Dzieje się tak z powodu klauzuli 2(b), która wymaga, aby każda praca wywodząca się z programu rozprowadzanego na licencji GPL sama posiadała taką licencję. Pewne kontrowersje powodowała również klauzula 3(b), wymagająca od licencjodawców udostępniania na życzenie źródeł na nośnikach fizycznych. Jednak eksplozja internetu spowodowała takie obniżenie kosztów publikowania archiwów (wymagała tego klauzula 3(a)), że nikt nie przejmuje się już wymaganiami dotyczącymi publikacji źródeł.

Nikt nie jest do końca pewny, co oznacza wyrażenie „zawiera lub jest wywiedziona” znajdujące się klauzuli 2(b), nie wiadomo też jakie sposoby użytkowania są chronione przez „prostą agregację”, o której jest mowa kilka akapitów dalej. Kwestie sporne dotyczą linkowania bibliotek i włączania do programu plików nagłówkowych rozprowadzanych na licencji GPL. Częściowo problem wynika z tego, że w Stanach Zjednoczonych prawo chroniące własność nie określa, czym jest wywodzenie. Tę kwestię pozostawiono sądom do rozpatrywania w konkretnych przypadkach, a oprogramowanie jest obszarem, w którym ten proces dopiero się rozpoczyna.

Z jednej strony, „prosta agregacja” z całą pewności umożliwia dostarczanie oprogramowania o licencji GPL na tym samym nośniku z oprogramowaniem komercyjnym, pod warunkiem że nie łączą się one ze sobą, ani nie wywołują się wzajemnie. Mogą to być narzędzia działające na tych samych formatach plików albo strukturach dyskowych. Z punktu widzenia prawa własności nie będą one wywodziły się jedno z drugiego.

Z drugiej jednak strony, włączenie kodu o licencji GPL do własnego komercyjnego kodu albo linkowanie kodu obiektowego rozprowadzanego na tej licencji do aplikacji komercyjnych, z całą pewnością jest pracą wywiedzioną i wymaga zastosowania licencji GPL do powstającego produktu.

Ogólnie uważa się, że jeden program może wywoływać drugi jako swój podproces, i nie stać się przez to pracą wywiedzioną z wywoływanego programu.

Największe kontrowersje powodują biblioteki linkowanie dynamicznie i biblioteki współdzielone. Fundacja FSF stoi na stanowisku, że jeżeli program wywołuje inny program jako bibliotekę współdzieloną, staje się pracą wywiedzioną z tej biblioteki. Jednak niektórzy programiści uważają, że jest to zbyt daleko idące twierdzenie. Z obu stron wysuwane są techniczne, prawne i polityczne argumenty, ale nie będziemy się tutaj nimi zajmować. Fundacja FSF napisała tę licencję i posiada do niej wszelkie prawa, dlatego należy uznać, że jej stanowisko jest obowiązujące, dopóki sąd nie postanowi inaczej.

Niektórzy uważają, że klauzula 2(b) została tak zaprojektowana, żeby licencja GPL zarażała każde komercyjne oprogramowanie, w którym wykorzystano choć skrawek kodu GPL. W tych środowiskach wspomniana licencja nazywana jest *GPV* (ang. *General Public Virus* — Ogólny Wirus Publiczny). Inni uważają, że „prosta agregacja" dotyczy każdego połączenia kodu GPL i nie-GPL w jednej jednostce kompilacji lub linkowania.

Ta niepewność powodowała w środowiskach otwartego źródła tyle dyskusji, że fundacja FSF napisała kolejną, nieco mniej restrykcyjną licencję „Library GPL" (którą następnie przemianowano na „Lesser GPL"). Pozwala ona na używanie bibliotek dostarczanych razem z kolekcją kompilatorów GCC.

Niestety każdy będzie musiał na własną rękę interpretować zapis klauzuli 2(b). Większość prawników nie zrozumie związanych z nią szczegółów technicznych, a jak dotąd nie ma żadnych przykładów jej interpretacji przez sądy. Jest faktem, że fundacja FSF nigdy (od czasu powstania w roku 1984 do połowy roku 2003) nie pozwała nikogo w związku z naruszeniami licencji GPL, ale wymuszała jej przestrzeganie za pomocą gróźb takich pozwów. Jak dotąd we wszystkich przypadkach okazało się to bardzo skuteczne. I jeszcze jeden fakt. W komercyjnej wersji przeglądarki Netscape Navigator znajdują się kody źródłowe i obiektowe programu rozprowadzanego na licencji GPL.

Licencje MPL i LGPL są znacznie mniej infekcyjne od licencji GPL. Pozwalają na linkowanie z kodem komercyjnym, który nie staje się w ten sposób pracą wywiedzioną, pod warunkiem że całość komunikacji między kodem komercyjnym i kodem GPL odbywać się będzie za pomocą API lub innego dobrze zdefiniowanego interfejsu.

16.7.3. Kiedy potrzebny jest prawnik?

Ten punkt skierowany jest do programistów tworzących oprogramowanie komercyjne i planujących zastosowanie w nim kodu rozprowadzanego pod ochroną którejś z tych standardowych licencji.

Po przebrnięciu przez te wszystkie prawne zawiłości musimy stanowczo zaprzeczyć, jakoby każdy, kto nie jest prawnikiem i ma pewne wątpliwości co do planów włączenia kodu na licencji GPL do swojego komercyjnego oprogramowania, musiał koniecznie korzystać z porady prawnika.

licencji jest tak jasny, jak pozwala na to język prawniczy — specjalnie zostały tak napisane. Ich zrozumienie nie powinno być trudne, jeżeli przeczyta się je wystarczająco uważnie. To raczej prawnikom tekst tych licencji będzie przysparzał więcej problemów. Prawo związane z oprogramowaniem jest dość mętne, a spraw wynikających z licencji otwartych źródeł nie było jeszcze w sądach (stan na połowę roku 2003). Nikt jeszcze nie został pozwany w związku z ich naruszeniem.

To wszystko oznacza, że żaden prawnik nie będzie w lepszej sytuacji niż uważny czytelnik tych licencji, a prawnicy na ogół obawiają się wszystkiego, czego nie są w stanie zrozumieć. W efekcie pytany o poradę prawnik może odpowiedzieć, że nie powinno się nawet zbliżać do oprogramowania o otwartych źródłach, tak naprawdę nie rozumiejąc nawet części aspektów technicznych ani intencji autora.

Ostatecznie, ludzie udostępniający swoje prace pod sztandarem licencji otwartego źródła, nie są wielkimi korporacjami, dla których pracują setki prawników szukających śladów krwi w wodzie. Są to pojedyncze osoby lub grupki ochotników, którzy chcą jedynie przekazać innym efekty swojej pracy. Kilka istniejących wyjątków (to znaczy wielkich korporacji zarówno tworzących oprogramowanie otwarte, jak i mających pieniądze na prawników) jest tak zainteresowanych rozwojem otwartego oprogramowania, że z pewnością nie chcą antagonizować środowiska przez roztrząsanie kwestii prawnych. Z tego powodu prawdopodobieństwo skierowania pozwu do sądu w związku z niewielkim, technicznym naruszeniem licencji jest znacznie mniejsze od prawdopodobieństwa porażenia piorunem w ciągu następnego tygodnia.

Nie należy tego rozumieć tak, że można te licencje traktować jak żart. Oznaczałoby to brak respektu wobec kreatywności i wysiłków, jakie włożono w rozwój tego oprogramowania. Z całą pewnością nie należałoby do przyjemności bycie pierwszą osobą pozwaną do sądu w sprawie o złamanie postanowień licencji otwartego oprogramowania. Jednak w związku z brakiem jakichkolwiek doświadczeń w tym zakresie dobre intencje i próby zaspokojenia wymagań autora kodu to 99% tego, co można zrobić w tym przypadku. Pozostały 1% pewności można uzyskać w wyniku konsultacji prawniczych, ale raczej nie będzie to miało jakiegokolwiek znaczenia.