TLS-Technologie
email office@tls-technologies.com
www http://www.tls-technologies.com/

Jakość w Open Source

Robert Maron*
robmar@tls-technologies.com

28 listopada 2001

Streszczenie

Referat omawia następujące problemy występujące często w Open Source:

- złą jakość dokumentacji — co uniemożliwia opiekę nad kodem i rozwijanie projektu przez osoby trzecie;
- złą jakość architektury — co skraca życie projektów; na przykład z powodu małej elastyczności;
- brak testów — co zniechęca wielu użytkowników do źle działających wersji beta.

Zajmiemy się też pomysłami na poprawienie tej sytuacji.

1 Wstęp

W dokumencie *Halloween* napisano:

> *The ability of the OSS process to collect and harness the collective IQ of thousands of individuals across the Internet is simply amazing. More importantly, OSS evangelization scales with the size of the Internet much faster than our own evangelization efforts appear to scale.* [a]

[a] Ten i dalsze cytaty pochodzą z dokumentów Halloween zob. Literatura, poz. 1

*Wykładowca w Instytucie Informatyki Uniwersytetu Warszawskiego. Współzałożyciel firmy TLS-Technologie(s). Zajmuje się zastosowaniami kryptografii, sieciami komputerowymi, językami funkcyjnymi. Interesuje się specyfikacją i tworzeniem niezawodnego oprogramowania.

Jednak nawet najlepszymi zasobami można źle zarządzać. Nawet największy zapał można zmarnować. Widziałem już wiele przykładów konfliktu w grupach pracujących nad otwartym oprogramowaniem. Wynikały one często z błędów metodologicznych prowadzących do marnotrawienia czasu i wysiłku deweloperów.

2 Dokumentacja

Stworzenie dobrego i czytelnego projektu nie wymaga wiele pracy. Dla średniej wielkości projektu, którego faza kodowania zajmuje 4–5 miesięcy dokumentację projektową można stworzyć w 1–2 tygodnie.

Fazy analizy biznesowej, analizy wymagań, projektowania są krótkie. Dzięki nim zespół oszczędza wiele wysiłku podczas kodowania. Jeszcze większy zysk z istniejącej dokumentacji projektowej zauważamy podczas dalszych iteracji i tworzenia nowych wersji oprogramowania. Nie bez znaczenia jest też wpływ istnienia takiej dokumentacji na łatwość wprowadzania nowych wykonawców do projektu.

Jakie są więc powody niemal całkowitego nieistnienia takiej dokumentacji w projektach otwartych?

Pierwszy powód jest naturalny. W toku ewolucji roślin lub zwierząt cecha, która jest nieużyteczna, nie musi zaniknąć. Do tego potrzeba, aby cecha była szkodliwa. Liczba chętnych do pomocy przy OpenSource jest bardzo duża i marnotrawienie wysiłku deweloperów nie powoduje szybkiego upadku projektu.

Kiedyś jednak osiągniemy krytyczny rozmiar projektów, albo wyczerpiemy zasoby chętnych do pomocy.

Drugi powód jest trudny do pokonania. Po prostu brakuje posiadających odpowiednią wiedzę chętnych do pracy przy OpenSource. Umiejętności analityka / projektanta są rzadziej spotykane niż umiejętności programisty.

Bardzo ważną rolę we wspieraniu tej fazy projektów otwartych mogą grać (i grają) szkoły wyższe. Gromadzą one wykładowców dysponujących odpowiednimi kwalifikacjami i mogą występować jako inicjatorzy dobrze prowadzonych projektów OpenSource.

2.1 Pomocne narzędzia

Diagramy UML. Do tworzenia diagramów w dokumentacji projektowej można użyć jednego z bardzo wielu istniejących i darmowych narzędzi. Istnieją też tanie programy tego typu, których zakup z pewnością nie jest problemem dla instytucji edukacyjnych wspierających otwarte oprogramowanie.

Dokumentacja techniczna. Istotnym czynnikiem ułatwiającym konserwację cudzego kodu jest istnienie i aktualność dokumentacji samego kodu. Najprościej uzyskać to stosując

narzędzia wspomagające generowanie dokumentacji technicznej z komentarzy umieszczonych w programach.

Dla Javy istnieje dość wygodny standard i narzędzie: JavaDoc (zob. Literatura poz. 5). Dla dużo szerszej gamy języków (Java, C++, Corba/IDL, C) powstał Doxygen (zob. Literatura poz. 6). Na podstawie odpowiednio skomentowanego kodu generuje on dokumentację techniczną (man, tex, rtf, html) z odpowiednimi referencjami w tekście, diagramami klas i diagramami referencji.

3 Architektura

> *OSS development process are far better at solving individual component issues than they are at solving integrative scenarios such as end-to-end ease of use.*

Problem właściwej architektury zaczyna się przy problemie braku dokumentacji. Aby zaistniała jakaś architektura, musi istnieć chęć jej zaprojektowania i przemyślenia.

Podobnie jak przy dokumentowaniu fazy analiz i projektu, do stworzenia dobrej architektury potrzeba odpowiednich umiejętności. Decyzje nie mogą być przypadkowe, lecz powinny wynikać z analizy.

Dzięki sieciom w ciągu ostatnich lat architektura systemów informatycznych uległa znacznej komplikacji. Kluczowe stały się umiejętności tworzenia systemów rozproszonych, tworzenia systemów modularnych, projektowania interfejsów nad protokołami wysokiego poziomu takimi jak Corba czy JavaBeans. XML z młodszego brata SGMLa i statusu ciekawostki awansował do roli jednego z najczęściej wykorzystywanych formatów (danych przesyłanych lub przechowywanych).

Nawet dla projektów, które nie wymagają tak zaawansowanej wiedzy, problem architektury istnieje. Spontaniczne rozrastanie się projektu może go zabić, jeśli za projektem nie stoi przemyślana i uaktualniana w miarę wzrostu architektura warstwowa.

4 Testy

Specyfika pracy nad oprogramowaniem otwartym różni się od metodologii firm deweloperskich. Inne jest też podejście do testów.

Pomyłką jednak jest zakładanie, że testy są zbędne i praca nad błędami jest łatwa i pomijalna.

Takie podejście kończy się zazwyczaj zniechęceniem deweloperów, bardzo długim cyklem prac pomiędzy stabilnymi wydaniami, wreszcie wielokrotnym poprawianiem tego samego błędu czy wielokrotnym zgłaszaniem tego samego błędu.

Praca na błędami, zarządzanie wydaniami, zarządzanie wymaganiami — to bardzo trudne i pracochłonne elementy procesu i warto myśleć o nich wcześnie, zanim ich niska jakość zacznie szkodzić prowadzonemu projektowi.

4.1 Bazy danych z listą błędów

Znamy narzędzia takie jak *Bugzilla* czy *SourceForge*. Pozwalają one na przechowywanie raportów o błędach, śledzenie cyklu życia błędu, śledzenie losu błędu i poprawek.

Można też ich używać jako narzędzi do przechowywania list poprawek, albo wręcz listy wymagań.

Oto lista kilku ważnych elementów niezbędnych do sukcesu:

1. Zgłoszenie błędu musi być łatwe. W tym celu albo wbudowujemy w program narzędzie do tego, które zgłoszenia wysyła e-mailem, albo znajdujemy chętnych testerów, którzy zadadzą sobie trud zgłaszania problemów.
2. Zgłoszenia traktujemy poważnie. Nic tak nie zniechęca testera–ochotnika, jak kłótliwy deweloper upierający się przy swoim.
3. Śledzimy losy błędu. Zaznaczamy w poprawce, którego błędu ona dotyczy. Błąd oznaczamy jako poprawiony dopiero po pomyślnych testach poprawki.
4. **Wykonujemy testy!**

4.2 Narzędzia do testów

Testy warto wykonywać. Mało który projekt ma tylu cierpliwych beta-testerów, że można całkiem pominąć fazę testów przed oddaniem kolejnej wersji.

Zazwyczaj wykonanie choćby minimum testów pozwala zachować opinię poważnej grupy deweloperskiej, która nie zawraca ludziom — użytkownikom wersji niestabilnej — głowy niedziałającymi wydaniami.

Podczas całego procesu tworzenia oprogramowania warto stosować zasadę piramidy — zanim przejdziemy do fazy wymagającej dużych wysiłków należy porządnie i do końca wykonać pracę wymagającą małego wysiłku. Zanim poprosimy kilkadziesiąt / kilkaset osób o pomoc, powinniśmy sami przetestować swój kod.

Znakomicie pomagają zautomatyzowane testy, niemal nieobecne przy tworzeniu oprogramowania OpenSource.

Dzięki automatyzacji i skryptom testowym (pisanym choćby i ręcznie w ulubionym języku skryptowym):

- szybko testujemy każde kolejne wydanie; zamiast męczyć się kolejnymi testami — wykonujemy je coraz szybciej i sprawniej;

- skracamy czas cyklu poprawka — wydanie; dzięki temu możemy dużo sprawniej tworzyć kolejne stabilne wersje oprogramowania;
- możemy przyjmować zgłoszenia błędów od znajomych testerów wraz z gotowymi skryptami (ang. *test–case*) testów, wskazującymi sposób na powtórzenie błędu; nadesłany skrypt jest potem podstawą do stworzenia kompletu skryptów mających potwierdzić skuteczność poprawki; formalizuje to fazę akceptacji błędu;
- dzięki gotowemu kompletowi skryptów wielokrotnie wykonujemy testy regresyjne; w ten sposób oszczędzamy wiele czasu w trakcie poszukiwania ukrytych błędów; obserwujemy to szczególnie pod koniec iteracji, gdy wersja jest „niemal stabilna”, a wciąż wykrywane drobiazgi odwlekają wydanie naprawdę stabilne.

5 Zakończenie

Jednym z zaleceń dokumentu *Halloween* było:

> *OSS projects have been able to gain a foothold in many server applications because of the wide utility of highly commoditized, simple protocols. By extending these protocols and developing new protocols, we can deny OSS projects entry into the market.*

Strategię tę obserwujemy w działaniu, otwarte protokoły wzbogacane są o wiele nowych elementów. Protokoły i formaty będące własnością Microsoftu są wciąż zmieniane.

OpenSource musi za tym nadążyć, co więcej, musi znaleźć siły i nie zniechęcać się trudnościami i ogromem pracy.

6 Literatura

1. Dokumenty Halloween: `http://www.opensource.org/halloween/halloween1.html`
2. Portal SourceForge: `http://www.sourceforge.net`
3. Krótki kurs UMLa: `http://www.togethersoft.com/services/practical\protect\T1\textunderscoreguides/umlonlinecourse/index.html`
4. Magic Draw, tanie narzędzie do rysowania UMLa: `http://www.magicdraw.com`
5. JavaDoc: `http://java.sun.com/j2se/javadoc/index.html`

6. Doxygen: `http://www.doxygen.org/`
7. JavaBeans: `java.sun.com/products/javabeans/`