Cykl wytwarzania (życia) oprogramowania c.d.

Sztuka Wytwarzania Oprogramowania, w. 7

Konrad Grochowski

Instytut Informatyki, Politechnika Warszawska, 2024 ©

Cykl życia c.d.

- Mamy wymagania
- Mamy *hipotezy/modele* jak pracować by przetworzyć je w kolejne wersje oprogramowania
- Warto teraz porozmawiać o elemencie o kluczowym znaczeniu...

Testowanie oprogramowania

- Z punktu widzenia klienta – najważniejsza część procesu wytwarzania oprogramowania
  - Nawet jeśli klient nie zdaje sobie z tego sprawy...
  - Może warto nazywać to *zapewnianiem jakości*?
- Weryfikuje zgodność programu z oczekiwaniami
- *Nic innego* nie daje informacji, *co* program *naprawdę* robi
  - Istnieją metody weryfikacji formalnej, jednak nie zawsze daje się je zastosować, są drogie, i często mogą być użyte do weryfikacji fragmentów „odseparowanych od świata zewnętrznego”.
  - Modelowanie może pomagać, ale zależy od procesu (czy model jest *przetwarzany* czy *przepisywany*) ale i jakości samego modelu

Testowanie oprogramowania – dwa użycia

- Oddzielny element cyklu, dedykowana czynność
  - Czasem zwany „walidacją”, „kwalifikacją”, „certyfikacją” etc.
- Integralny element „implementacji” / codziennej pracy
  - Nie można ocenić wykonanej pracy bez jej przetestowania
- Oba mają sens

Testowanie oprogramowania

- Testowanie może tylko zweryfikować,
  czy nie ma ***znanych błędów***
- Testy robi się *na coś* i tak naprawdę *„zielone testy”* to *„negatywne wyniki”*
- Czyli testowanie nie daje 100% gwarancji poprawności działania
- Ale to nie znaczy, że klient ma testować „na produkcji”...

Testowanie oprogramowania

- Rodzajów testów oprogramowania jest bardzo dużo
- W pewnym sensie każde uruchomienie programu i obserwacja jego zachowania to test
  - Tylko najlepiej, żeby klient obserwował tylko oczekiwane działanie...
- Istnieją dedykowani specjaliści zajmujący aspektami testowania
  - Product Assurance / Quality Assurance
    (choć w tych terminach jest więcej, niż tylko testy)
- Testowanie nie musi dotyczyć wyłącznie kodu
  - Np. zgodność instrukcji użytkownika z rzeczywistym oprogramowaniem

Podział testów

- Ze względu na to czy program jest fizycznie uruchamiany:
  - Statyczne
  - Dynamiczne
- Ze względu na obiekty objęte testem (poziom testu):
  - Jednostkowe
  - Integracyjne
  - Systemowe
  - *Akceptacyjne*
- Testy bezpośrednio weryfikujące wymagania czasem nazywa się *testami walidacyjnymi*

Podział testów

- Ze względu na podejście do obserwacji wyniku:
  - White-box
  - Black-box
  - Grey-box
- Ze względu na sposób przeprowadzenia:
  - Automatyczne
  - Manualne

Podział testów

- Ze względu na badane aspekty:
  - Testy funkcjonalne (functional testing)
  - Testy wydajnościowe (performance testing)
  - Testy przeciążeniowe (stress testing)
  - Testy bezpieczeństwa (security testing)
  - Testy bezpieczeństwa (safety testing)
  - Testy niezawodności (dependability testing)
  - Testy odporności (recovery testing)
  - Testy zgodności (compatibility testing)
  - Testy dokumentacji (documentation testing)
  - ...

Inne przykłady rodzajów testów

- Testy dymne (smoke tests)
- Testy regresji (regression testing)
- Testy destrukcyjne (destructive testing)
- Testy „interakcji z człowiekiem”
  - Testy tłumaczeń i regionalizacji (localization and internationalization)
  - Testy używalności (usability)
  - Testy dostępności (accessibility)

Dobry test

- Jednoznacznie ustalony cel testu
- Czytelny scenariusz testu
- Określony warunek końca testu
- Określony warunek oceny testu (zaakceptowania)
- Jeśli możliwe – odwołanie do wymagań, z których test wynika.
- Idealnie:
  - Automatyczny
  - Deterministyczny i powtarzalny
  - Jednoznacznie i łatwo oceniany
  - Szybki

Testowanie jednostkowe

- „Najwyższa stopa zwrotu w jakości”
  - Krótka ścieżka pomiędzy inwestycją/czasem a wynikiem i wpływem na kod
- „Najlepszy przyjaciel programisty”
  - Tylko test mówi, co kod naprawdę robi, więc test jednostkowy pozwala wiedzieć co jest zrobione „na koniec dnia”
- Wymusza lepszą architekturę (decoupling)
- Brak formalnej definicji jednostki
- Nie wszystko da się unit-testować
- Nie zawsze odpowiada bezpośrednio wymaganiom
  - Usunie drobne błędy, ale nie poważne naruszenia funkcjonalności

Automatyzacja cyklu

- Mamy zdefiniowany jakiś proces wytwarzania oprogramowania
- Jak tylko pojawia się proces to każdy ~~leniwy~~ porządny inżynier zadaje sobie pytanie:
  jak mogę zmusić maszyny, by mi w tym procesie pomogły?
- Czyli *co można zautomatyzować?*
- Ale czy to faktycznie tylko *lenistwo*?

Automatyzacja procesu a jakość

- *Rozsądny* poziom formalizacji procesu zazwyczaj pomaga osiągnąć zakładane cele (a dokładniej – bez formalizacji nie ma procesu, bez procesu nie ma powtarzalności i przewidywalności)
- Najlepszą formalizacją jest automatyzacja, szczególnie jeśli zawiera obiektywną i jednoznaczną ocenę zgodności z procesem
- We współczesnych procesach wytwarzania oprogramowania stara się automatyzować jak największą część procesu
  (choć *code review* nic nie zastąpi jak długo kod będzie robiony przez ludzi)

Continuous Integration

- Proces ciągłej integracji i weryfikacji zmian wprowadzanych do projektu (często stosowana z *feature branch*)
- Ma zapobiegać „zepsuciu” głównej gałęzi kodu
- Dostarcza informacji zwrotnej autorom zmian – wcześniejsze wykrywanie błędów
- Może przeprowadzać walidacje niedostępne programistom „na co dzień” (kompilacja kilkunastoma kompilatorami etc.)
- Nie jest złotym środkiem na wszystkie problemy –
  jest tak dobry, jak dobre są testy – napisane przez zespół...

Continuous Integration

- Co może robić?
  - kompilować kod
  - analizować kod statycznie
  - uruchamiać testy jednostkowe i zbierać pokrycie
  - uruchamiać testy walidacyjne / akceptacyjne
  - generować dokumentację
  - przygotowywać pakiety instalacyjne
  - czekać na krok wykonany przez człowieka
  - co tylko się chce, jak długo jest to realizowalne przez narzędzie, które da się uruchomić „automatycznie”
- Sekwencję tych operacji nazywa się *potokiem CI (CI pipeline)*

Continuous Delivery / Continuous Deployment

- Takie wykorzystanie *potoku,* które gwarantuje, że zawsze mamy produkt, który możemy dostarczyć klientowi
- W szczególnych przypadkach „dostarczenie” może być nawet częścią samego potoku
- Raczej niespotykane w oprogramowaniu krytycznym, ale sama idea „zawsze dobrego stanu repozytorium” jest bardzo dobra

Continuous Integration - realizacja

- Najczęściej narzędzie „sprzęgnięte” z systemem kontroli wersji
  - Na przykład – każdy *git push* uruchamia proces CI
- Może wymagać dużo zasobów (więcej niż maszyna programisty)
- Stabilne/odtwarzalne środowisko
  - Pomocna bywa wirtualizacja
  - Albo konteneryzacja (*Docker* – uruchamianie aplikacji Linuxowych w odseparowanym środowisku w ramach tego samego systemu)
  - Często „w chmurze” (to też dotyczy sytuacji z dużą ilością zasobów)
- Proces dzieli się na samodzielne „małe” kroki, dla przejrzystości
- Niezależne zadania można zrównoleglić

Continuous Integration - narzędzia

- Współczesne systemy „opakowujące” kontrolę wersji często oferują wbudowany system do CI/CD
  - GitLab CI/CD
  - GitHub Actions
  - Bitbucket Pipelines
- Istnieją dedykowane narzędzia
  - Jenkins
  - Travis CI
  - Circle CI
  - AppVeyor
  - …

Dziękuję za uwagę

Konrad.Grochowski@pw.edu.pl