Metryki

Sztuka Wytwarzania Oprogramowania, w. 8

Konrad Grochowski

Instytut Informatyki, Politechnika Warszawska, 2024 ©

Metryka

- Miara odległości między elementami zbioru
- W inżynierii oprogramowania – trochę mniej formalnie:
  - Liczbowa reprezentacja wybranej cechy oprogramowania
  - Powinna być obiektywnie mierzalna
- Metryki są więc (bardzo) różne i jest ich (bardzo) wiele
- Używane do:
  - Oceny oprogramowania (np. jakości)
  - Estymacji kosztów
  - Walidacji wymagań

Metryki i procesy

- W ramach procesu wytwarzania oprogramowania może być rozsądne *śledzenie (tracking)* wybranych metryk
- Może to pomagać określać postęp prac
- Ale też pilnować określonej oczekiwanej jakości
- Zestaw śledzonych metryk zależy od projektu / branży

Metryki oprogramowania (przykłady)

- Wielkość programu (pliku wykonywalnego/paczki instalacyjnej)
- Czas wykonania programu
- Czas ładowania programu
- Zużycie pamięci/procesora/dysku przez program

- Jak widać – metryki potrafią być „dynamiczne” i „statyczne”
  - I jak zwykle – obie mają wady i zalety

Metryki oprogramowania

- Metryki mogą występować jako „średnie”, czy „maksymalne”, czy „zmierzone w specyficznych warunkach”
- Sposób pomiaru jest *elementem definiującym metrykę*
  - Np. kogoś może ciekawić „średnia liczba linii kodu w funkcji” ale do oceny jakości użyć może „liczby funkcji przekraczających N linijek kodu” (co może być traktowane jako „maksymalna długość funkcji”)
  - W wypadku metryk często stosuje się „skróty myślowe”, ale czasem warto pamiętać o formalizmach i ścisłym ich definiowaniu

Metryki kodu (przykłady)

- Metryki oprogramowania, które można uzyskać analizując kod
- Liczba linii kodu (LOC – *lines of code, kLOC*)
  - Obiektywna, ale nie może służyć porównywaniu różnych programów w kategoriach „trudności” czy „złożoności”
  - W bardzo specyficznych warunkach może pomagać estymować koszty, ale bywa (wciąż) nadużywana w tym zakresie
- Procent linii komentarzy w kodzie
  - *Cóż, wiecie, co ja myślę o komentarzach...*
  - Ale w plikach nagłówkowych bibliotek może być pomocna
  - Niektórzy wciąż widzą w tym jakąś miarę jakości, ale nawet „sztywne” standardy nie zalecają takiego podejścia (ECSS, ISO)

Metryki kodu (przykłady c.d.)

- Złożoność cyklometryczna (*Cyclomatic complexity – CC*)
  - <liczba punktów decyzyjnych w kodzie> + 1
  - Próba mierzenia „złożoności kodu” – nie idealna, ale dobra do sygnalizacji potencjalnie „złych” funkcji
- Liczba linii w funkcji/module
- Liczba argumentów w funkcji / pól w klasie
- Głębokość zagnieżdżeń w kodzie (if{if{if}})

- To są *rozsądne* metryki, dla których warto przyjąć sobie jakieś „wartości ostrzegawcze” i robić *inspekcje* kodu, gdy zostaną przekroczone
- Wiele współczesnych narzędzi do pracy z kodem pozwala skonfigurować ich mierzenie i raportowanie / ostrzeganie

Metryki kodu

- Pokazane metryki wyglądają trochę jak coś, co powinno „wyjść” podczas dobrego *code review*
- Ale:
  - Obiektywna miara to jednak obiektywna miara...
  - ...a ludzie są omylni
  - Podpięcie pomiaru pod CI daje szansę na wczesne ostrzeganie i skraca czas spędzony przez zespół nad *code review*
  - Czasem potrzeba „dowodów” (*evidence*) – zależy od klienta
- Istnieją też metryki „ambitniejsze”, próbujące mierzyć poziom „powiązań” (*coupling*), czy samą „utrzymywalność” w kodzie etc.

Metryki procesów

- Pomiarom można poddawać nie tylko samo oprogramowanie, ale i proces jego wytwarzania
  - i nie jest to zadanie tylko dla menadżerów
- Mierzyć można:
  - Czas poświęcony przez zespół (np. w stosunku do zaplanowanego czasu / budżetu)
  - Częstotliwość zgłaszania problemów
  - Czas rozwiązania problemów zgłaszanych przez klientów
  - „Masa” wykonywana w Sprincie to też metryka
- Warto dyskutować/interesować się stosowanymi w firmie metrykami procesowymi, bo mogą okazać się pomocne w walce o jakość kodu

Metryki testów

- Większość tego co nadaje się do programu, nadaje się i do mierzenia jego testów
- Ale jest pewna grupa metryk związanych z testowaniem
- **Pokrycie** *(coverage)*
- Ta metryka faktycznie pozwala *pomóc* ocenić jakość testów (w obrębie danego projektu, czyli na ile dany projekt jest dobrze przetestowany)

Pokrycie

- Pokrycie linii – linie kodu wywołane przez test
- Pokrycie gałęzi (*branch*) – ścieżki kodu wykonane przez test
- Pokrycie decyzji – każda wykonana „decyzja” w kodzie
  - Dwa powyższe bywają „mieszane”, dużo zależy od dostępnych narzędzi

- Ale też:
  - Pokrycie wymagań przez testy
    (czy każde testowalne wymaganie ma swój test?)
  - I to też można mierzyć i automatyzować

Pokrycie

- To, że badamy pokrycie, nie oznacza od razu, że robimy *white-box!*
  - *Gdzieś widziałem taki błędny napis, a potem trafia na kolokwium...*
- Black/white box dotyczy konstrukcji testu i sposobu *weryfikacji* wyniku – to jak „patrzymy” na testowane oprogramowanie
- Pokrycie tak, dotyka kodu, ale używa go do uzyskania „miary”, a nie do konstrukcji testu
- Co więcej – **robienie testu „dla pokrycia” jest błędnym podejściem**
- *Powtórka:* Dobry test funkcjonalny *często* jest typu black-box, a testy jednostkowe *często* są testami funkcjonalnymi... (*grey-box*)

Pokrycie a jakość testu

- Inne pokrycie niż 100% linii i gałęzi mówi, że mamy sekwencje kodu, których nie uruchomiliśmy
  (*czyli nie wiemy, co się w nich dzieje*)
- Ale uzyskanie 100% jest naprawdę drogie...
  - Przyjmuje się, że wartości > 80-90% są już „dobre” (dla całego projektu)
  - Ale oprogramowanie krytyczne musi mieć 100%
  - TDD pomaga, ale trzeba pamiętać, że ono nie testuje
- A co jak jesteśmy przekonani, że dobrze napisaliśmy testy, a i tak nie ma 100%?
  - Analiza „martwego kodu” – może coś do wyrzucenia? A może jednak testy nie tak dobre, jak myśleliśmy?

Pokrycie a jakość testu

- Czy testy dające 100% pokrycia gwarantują poprawność działania programu?
  - Nie.
  - Testy nigdy nic nie gwarantują.
  - Ale i tak pozostają niezbędnym narzędziem...
- Jakość testu opiera się na jego konstrukcji i zgodności z wymaganiami – to może być niemierzalne...
  - Czy asercje są dobre?
  - Czy scenariusz jest reprezentatywny?
  - Czy dobrze pomyślana dziedzina problemu (warunki brzegowe)?

Jak napisać dobry test (funkcjonalny)?

- Mam funkcję `int fun(int x)`
- Żeby ją w pełni przetestować, potrzebuję 4 mld testów...
- Chyba, że wiem coś o `x` (z kodu – *white box,* z wymagań – *black*).
- Na przykład, że tylko dla wartości [1, 100) funkcja ma sens.
- No to mogę napisać 101 testy dla wartości [0, 100].
- Jakbym wiedział, że dla całego przedziału funkcja wykonuje takie same obliczenia, to mógłbym dalej redukować do 4 – 5 testów:
  - 0, 1, *2*, 99, 100
  - Taki podział to *klasy jednoznaczności/równorzędności* (equivalence class)

Powtórka: jak zrobić dobry test (jednostkowy)?

- Automatyczny
- Powtarzalny i deterministyczny
- Szybki
- Jednoznaczny
- Skoncentrowany na pojedynczym aspekcie
- Czytelny (testy to najlepsza dokumentacja kodu)
- Niezależny

Podsumowanie

- Nie ma nic lepszego niż *code review*, bo wciąż głównym odbiorcą kodu jest drugi człowiek
- Metryki mogą pomagać wychwytywać problemy,
  ale nie są *bezwzględnie* obiektywną miarą
  (są obiektywne w zakresie danego projektu)
- Warto mierzyć swoje oprogramowanie (i procesy), tak by mieć argumenty na podparcie swoich „odczuć”
- Dobry kod i dobre testy to wciąż domena *rzetelności, doświadczenia i profesjonalizmu* – automaty mogą pomóc, ale nie mogą zastąpić dobrego procesu inżynieryjnego

Dziękuję za uwagę

Konrad.Grochowski@pw.edu.pl