



Analiza oprogramowania

Sztuka Wytwarzania Oprogramowania, w. 12

Konrad Grochowski

Instytut Informatyki, Politechnika Warszawska, 2024 ©





Analiza oprogramowania

- › Analiza, czyli badanie oprogramowania, już pojawiła się na wykładzie:
 - Testowanie (badanie + weryfikacja oczekiwań)
 - Metryki (pomiar)
- › Ale jest to szerszy temat i warto znać parę innych przykładów
- › Podstawowy podział:
 - Analiza statyczna
 - Analiza dynamiczna



Analiza statyczna

- › Mimo, że może być równie różnorodna, co analiza dynamiczna, nazwa często jest używana jako zamiennik „statycznej analizy kodu w poszukiwaniu błędów”
- › Co potrafi statyczna analiza kodu:
 - Wykryć błędy czasu wykonania
 - › także zakładając specyficzny scenariusz wykonania (np. nie osiągniany przez testy)
 - Wskazać *potencjalne* błędy
 - › weryfikacja wejścia ale też „undefined behaviour”
 - Pilnować zgodności ze standardem kodowania (słowo klucz: *linter*)



Analiza statyczna

- › *Bywa szybka*
 - w porównaniu z uruchamianiem wielu testów (w tym systemowych)
 - dla pojedynczego pliku
 - ale jej uruchomienie dla całego projektu może być „wolne” dla człowieka
- › Potrafi wskazywać „nieczytelności”
- › Ale nie jest nieomylna
 - Czyli nie daje *gwarancji*, że danego typu błędu nie ma
- › Miewa też wyniki fałszywie pozytywne (*false positives*)
 - Jednym z celów jest wskazanie fragmentów „do obejrzenia”, dlatego narzędzia często wybierają więcej false positives niż mniej false negative



Analiza statyczna – zalecenia

- › Powinna być elementem pracy nad projektem
- › Im szybciej „odpalana” tym lepiej
 - Na laboratorium *clang-tidy* był „wbudowany” w kompilację
 - Standardowo dodaje się wywołania takich narzędzi do CI
- › Dobrze mieć kilka różnych narzędzi (mają różne cechy i możliwości)
- › Tani sposób usuwania potencjalnych „brzydkich” (czyt. „niebezpiecznych”) kawałków kodu
 - Przy założeniu „zero tolerancji dla zgłaszanych problemów”
 - Nawet dodawanie explicite „ignoruj błąd w tym miejscu” jest lepsze, niż log mający 6000 linii (bo nikt nie zauważy 6001)
 - A warningi to też statyczna analiza ;)



Dynamiczna analiza

- › Analizować można właściwie wszystko, co dzieje się z programem i *jego otoczeniem* podczas wykonania
- › Chyba częściej niż analiza statyczna, analiza dynamiczna bywa używana do badania oprogramowania, którego kod jest nieznany
 - Ale testowanie jest składnikiem analizy dynamicznej, więc to trochę kwestia nazewnictwa (testowanie – znane, analiza – nieznane)
- › Daje *rzeczywiste* wyniki, ale *bywa* czasochłonna i jest zależna od *reprezentatywności* badanego scenariusza
- › Aplikacja może być uruchamiana zarówno w środowisku realnym, jak i „zwirtualizowanym”



Dynamiczna analiza

- › Co można monitorować:
 - Użycie pamięci (+ wycieki)
 - Użycie *cache*
 - Użycie procesora
 - Wykorzystanie sieci
 - Wykorzystanie dysku
 - ...
- › Wykorzystanie tych danych do poprawy wydajności działania aplikacji jest nazywane *profilowaniem* (bo uzyskuje się informację o *profilu* wykonania – zarys „cech”)



Problemy profilowania

- › Profilować należy aplikację skompilowaną z optymalizacją
- › W efekcie relacja instrukcji wykonywanych do kodu może nie zawsze być czytelna
- › Profilowanie może samo zaburzać pomiar
 - Instrumentacja / symulacja
 - Próbkowanie
- › W systemach innych niż systemy czasu rzeczywistego pomiar czasu jest obarczony błędem
- › Wybór scenariusza do profilowania jest kluczowy



Problemy profilowania „współczesne”

- › Procesor niekoniecznie jest „wąskim gardłem”
- › Współczesne programy wykorzystują wiele „zewnętrznych” zasobów i dostęp do nich może być tym „wąskim gardłem”, a to czasem ciężko sprofilować
- › Współczesne architektury procesorów premiuja optymalizację na poziomie ułożenia danych w pamięci bardziej niż optymalizację redukcji instrukcji/skoków
 - Oczywiście profilowanie może wskazać „wolną funkcję”, ale przyczyna jej powolności może nie być oczywista
- › Ale nic nie daje takiej satysfakcji, jak po sesji pomiarów i poprawek osiąga się kilkudziesięcioprocentowe przyspieszenie :)



Analiza aplikacji kiedy nie mamy aplikacji

- › Czyli co robić, jak klient napisał „program mi wybuchł”
- › Jeśli nie przygotowaliśmy się na to wcześniej, to mamy problem
- › Niektóre środowiska dają szansę, że być może będzie dostępny tzw. „*stack-trace*” (zrzut ścieżki wykonania kodu z momentu wystąpienia błędu krytycznego).
O ile klient będzie wiedział, jak go zebrać.
- › Można przygotować program tak, by to generował, plus dołączał np. stan rejestrów procesora etc. (*core dump, death report*)
- › Ale wciąż prostym i popularnym rozwiązaniem jest *logowanie*



Logowanie wykonania aplikacji

- › Logowanie polega na umieszczaniu w określonych miejscach kodu instrukcji dodających do rejestru (*log*) informację, że dana linijka się wykonała
 - np. z datą, często z wiadomością czytelną dla człowieka, czy z jakimiś parametrami
- › Dobry system do logowania rozróżnia:
 - Poziom logowania (Fatal/Critical/Error/Warning/Info/Debug/Trace)
 - Źródło logowania (moduł systemu)
 - Punkt docelowy (*sink*, np. plik, czy rejestr systemu operacyjnego)
 - › Bardzo dobry system do logowania pozwala ustalać wiele *sinków* z różnymi filtrami



Logowanie

```
BOOST_LOG_TRIVIAL(trace) << "A trace severity message";  
BOOST_LOG_TRIVIAL(debug) << "A debug severity message";  
BOOST_LOG_TRIVIAL(info) << "An informational severity message";  
BOOST_LOG_TRIVIAL(warning) << "A warning severity message";  
BOOST_LOG_TRIVIAL(error) << "An error severity message";  
BOOST_LOG_TRIVIAL(fatal) << "A fatal severity message";
```

```
if(logger.isDebugEnabled()){  
    logger.debug("This is debug : " + parameter);  
}  
  
if(logger.isInfoEnabled()){  
    logger.info("This is info : " + parameter);  
}  
  
logger.warn("This is warn : " + parameter);  
logger.error("This is error : " + parameter);  
logger.fatal("This is fatal : " + parameter);
```

```
2014-07-02 20:52:39 DEBUG HelloExample:19 - This is debug  
2014-07-02 20:52:39 INFO HelloExample:23 - This is info :  
2014-07-02 20:52:39 WARN HelloExample:26 - This is warn :  
2014-07-02 20:52:39 ERROR HelloExample:27 - This is error  
2014-07-02 20:52:39 FATAL HelloExample:28 - This is fatal
```



Problemy logowania

- › Można logować za dużo i log stanie się nieczytelny
- › Można logować za mało i log nie będzie miał dość informacji
- › Logowanie wpływa na wydajność
 - Bo dotyka I/O
 - Bo można zrobić

```
LOG(debug) << "Additional info: " << complexFunction();
```

- › Logowanie może naruszać bezpieczeństwo
 - LOG(debug) << "Success: " << user->name() << user->password();
 - Albo po prostu zawierać błędy w bibliotece...
- › Trzeba myśleć o tym, czy log jest dla programistów, dla inżynierów wdrożeniowych czy dla „zwykłego” użytkownika



Logowanie - podsumowanie

- › Często traktowane przez niedoświadczonych programistów jako „taki printf do debugowania”
- › A to potężne narzędzie, tylko musi być traktowane tak samo poważnie, jak każdy inny aspekt tworzenia programu:
 - Szczegółowość wiadomości musi być określona
 - Zawartość informacji w wiadomości musi być przemyślana (samo *error...*)
 - Język musi być dopracowany dla odbiorcy
 - Użytkownik musi być w stanie przekazać logi
- › Jak zwykle – nie ma nic za darmo, nie ma miejsc „o które można mniej dbać”. A dobre logi ratują przed bezsennymi nocami :)

Dziękuję za uwagę

Konrad.Grochowski@pw.edu.pl

