Bezpieczeństwo w inżynierii oproramowania

Wykład 1

dr Agnieszka Zbrzezny

Instytut Służby Kryminalnej
Wydział Bezpieczeństwa Wewnętrznego WSPol

23 października 2018

Informacje organizacyjne

- Materiały do wykładów i ćwiczeń będą umieszczane na stronie

  `http://agnieszkazbrzezny.edu.pl`
- Konsultacje:
  - Środy po zajęciach
  - Skype

Informacje

- Materiały do wykładów i ćwiczeń będą umieszczane na stronie

  `http://agnieszkazbrzezny.edu.pl`
- Konsultacje:
  - Środy po zajęciach
  - Skype

Zagadnienia na parę pierwszych wykładów – teoria

1. Koncepcja jakośći
   - Definicja jakości
   - Normalizacja
   - Znaczenie jakości w projektach informatycznych
   - Koszty jakości
2. Zarządzanie jakością
   - Zarządzanie procesowe
   - Zarządzanie jakością
   - Zarządzanie przez jakość
   - Koncepcje zarządzania jakością
   - Zarządzanie jakością oprogramowania
   - Manifest jakości
   - Standardy

Zagadnienia na parę pierwszych wykładów – praktyka

1. pytest - testowanie w pythonie
2. Debugowanie
   - wyjątki
   - błędy semantyczne
   - debugowanie metodą naukową

Literatura

1. Brian Okken. Python testing with pytest. Simple, Rapid, Effective, and Scalable. 2017
2. Kristian Rother. Python dla profesjonalistów. Debugowanie, testowanie i utrzymywanie kodu. Helion. 2017
3. Karolina Zmitrowicz. Jakość projektów informatycznych. Rozwój i testowanie oprogramowania. Helion. 2015
4. `https://bezpiecznymiesiac.pl/`

Październik Miesiącem Cyberbezpieczeństwa w Europie

- Kampania edukacyjna realizowana w 28 krajach z całej Europy z inicjatywy Komisji Europejskiej przy współpracy z Europejską Agencją Bezpieczeństwa Sieci i Informacji (ENISA).
- elem jest popularyzacja wiedzy o zagrożeniach w cyberprzestrzeni oraz promocja bezpiecznego korzystania z Internetu i nowoczesnych technologii IT w domu, pracy czy w szkole.

Październik Miesiącem Cyberbezpieczeństwa w Europie

- 4 bloki tematyczne:
  - Tydzień 1 – Cyberhigiena najprostsze działania, które powodują wzrost cyberbezpieczeństwa każdego z nas (np. systematyczna zmiana hasła i aktualizacja oprogramowania)
  - Tydzień 2 – Edukacja dla cyberbezpieczeństwa, czyli jak uczyć świadomego korzystania z internetu? Jak kształcić specjalistów w zakresie cyberbezpieczeństwa i nowoczesnych technologii?
  - Tydzień 3 – Cyber Scams czyli jak nie paść ofiarą oszustów w sieci (organizowane przy wsparciu Europol EC3)
  - Tydzień 4 – Nowoczesne technologie, tydzień poświęcony takim tematom jak: sztuczna inteligencja (AI), czy Internet Rzeczy (IoT), fake news.

Październik Miesiącem Cyberbezpieczeństwa w Europie

- Cyberhigiena to zwykłe, codzienne działania, które podnoszą nasze bezpieczeństwo online.
- Regularne mycie rąk zabezpiecza nas przed infekcjami, a przeglądy techniczne samochodu przed groźnymi awariami.
- Tak samo aktualizowanie oprogramowania i zmiana haseł dostępu do kont i profili, powinny stać się naszymi zdrowymi, odpowiedzialnymi nawykami.
- W ten sposób zwiększamy swoje bezpieczeństwo w sieci.

Botnet

- Internet pozwala komunikować się z całym światem i czyni życie prostszym.
- Niestety, w sieci obecne są również osoby, które nie mają dobrych zamiarów i korzystają z niej, aby szkodzić innym.
- Popularnym rodzajem takiego szkodnictwa jest infekowanie podłączonych do Internetu urządzeń oprogramowaniem pozwalającym zdalnie je kontrolować.
- Po udanym zarażeniu, urządzenie może stać się częścią botnetu – większej sieci zaatakowanych urządzeń, zdalnie kontrolowanych przez hakerów.
- Przestępcy korzystają z botnetów do osiągania korzyści majątkowych, rozsyłania spamu, infekowania kolejnych użytkowników, przeprowadzania ataków na strony internetowe i innych złośliwych działań.
- Pojedynczy botnet składa się przeważnie z kilkuset lub nawet kilku tysięcy rozproszonych po świecie urządzeń.

Cyberhigiena-ZADBAJ O AKTUALIZACJE I BEZPIECZEŃSTWO URZĄDZEŃ

- Korzystaj z aktualnego oprogramowania: Regularnie aktualizuj system operacyjny, program antywirusowy, przeglądarkę internetową. Dzięki aktualizacjom łatwiej ustrzeżesz się przed szkodliwym oprogramowaniem i innymi zagrożeniami obecnymi w sieci.
- Włącz aktualizacje automatyczne: Wiele aplikacji oferuje możliwość automatycznego pobierania aktualizacji, w celu ochrony przed nowymi zagrożeniami. Skorzystaj z tego rozwiązania wszędzie tam, gdzie to możliwe.

Cyberhigiena-ZADBAJ O AKTUALIZACJE I BEZPIECZEŃSTWO URZĄDZEŃ

- Chroń urządzenia podłączane do sieci: Nie tylko komputery, ale także smartfony, tablety i inne podłączane do Internetu urządzenia, potrzebują ochrony przed wirusami i złośliwym oprogramowaniem.
- Skanuj przed użyciem: Nie podłączaj do komputera nośników, których pochodzenie nie jest Ci znane. Dyski zewnętrzne, pendrive'y, czy inne nośniki danych mogą być niebezpieczne (zainfekowane przez szkodliwe oprogramowanie). Zanim otworzysz ich zawartość skorzystaj ze skanera antywirusowego.

Ransomware

- Ransomware to szkodliwe oprogramowanie, które odbiera Ci dostęp do Twoich plików poprzez ich zaszyfrowanie.
- Być może zauważyłeś, że w nazwie znajduje się angielskie słowo ransom, oznaczające okup.
- Po przeprowadzonym ataku, w zamian za przywrócenie dostępu do danych, przestępcy żądają zapłaty okupu, określonej kwoty pieniędzy.
- Chcąc zainfekować komputer swojej ofiary, atakujący umieszczają złośliwy odnośnik lub załącznik w wiarygodnej, lecz fałszywej wiadomości e-mail.
- Zaszyfrowane zostają dokumenty, fotografie, pliki projektowe, wrażliwe dane przedsiębiorstwa, etc. Potencjalnym celem ataku może stać się zarówno firma, jak i osoba prywatna.

Cyberhigiena - ZABEZPIECZ DOSTĘP DO POSIADANYCH INFORMACJI

- Dwuskładnikowe uwierzytelnianie: Zadbaj o swoje konta w sieci. Logowanie oparte wyłącznie o nazwę użytkownika i hasło nie jest wystarczająco bezpieczne (szczególnie w przypadku konta e-mail, portalu społecznościowego czy bankowości internetowej). Aktywuj weryfikację tożsamości opartą o dodatkowy składnik, np. kod SMS, token, czy klucz sprzętowy.
- Stwórz mocne hasło: Dobre hasło składa się przynajmniej z 12 znaków. Skup się na pozytywnych zdaniach lub zwrotach o których lubisz myśleć i które łatwo zapamiętasz (np. „Kocham miasto muzyki”). Na wielu stronach internetowych, możesz przy wprowadzaniu hasła używać spacji.

Cyberhigiena - ZABEZPIECZ DOSTĘP DO POSIADANYCH INFORMACJI

- Jedno hasło, jedno konto: Jeżeli chcesz utrudnić działania przestępcom, dla każdego konta przypisz oddzielne hasło. Niezbędne minimum, to rozdzielenie kont używanych do pracy i celów prywatnych. Zadbaj o silne hasło do najistotniejszych serwisów (bankowość, poczta elektroniczna, portale społecznościowe)
- Przechowuj bezpiecznie: Każdy może zapomnieć swojego hasła. W celu ułatwienia nam życia stworzono aplikacje zwane menadżerami haseł. Służą do bezpiecznego przechowywania danych dostępowych. Możesz z nich korzystać. Jeżeli zapisałeś hasło na kartce (lepiej tego nie rób), postaraj się umieścić ją w bezpiecznym miejscu, z dala od komputera.

Cyberhigiena - KORZYSTAJ ROZWAŻNIE

- Zatrzymaj się, jeśli masz wątpliwości: Linki i załączniki w wiadomościach e-mail, spreparowane posty w mediach społecznościowych oraz reklamy - to częste metody używane przez przestępców w celu kradzieży danych. Jeżeli wydają Ci się podejrzane, po prostu je zignoruj. Nawet, jeżeli źródło wygląda na zaufane.
- Uważaj na hotspoty Wi-Fi: Ogranicz aktywność w publicznie dostępnych sieciach Wi-Fi. Używając poza domem kluczowych serwisów (poczta e-mail, bankowość internetowa, portale społecznościowe) bezpieczniej będzie użyć własnego modemu LTE lub połączenia VPN. Pamiętaj o wyłączaniu transmisji Wi-Fi i Bluetooth, kiedy z niej nie korzystasz.

Cyberhigiena - KORZYSTAJ ROZWAŻNIE

- Chroń swoje finanse: Korzystając z bankowości internetowej i sklepów online, upewnij się, że połączenie jest objęte szyfrowaniem (zielona kłódka oraz prefiks „https://” w pasku adresu). Odczytując kod SMS uwierzytelniający transakcję, zweryfikuj kwotę przelewu i numer rachunku odbiorcy!

Cyberhigiena - BĄDŹ ŚWIADOMYM UŻYTKOWNIKIEM

- Pozostań na bieżąco: Nie lekceważ informacji ze świata bezpieczeństwa IT. Jeśli coś podawane jest do publicznej wiadomości, najczęściej dotyczy także Ciebie.
- Pomyśl, zanim zadziałasz: Bądź ostrożny wobec korespondencji zachęcającej do natychmiastowych działań. Szczególnie, jeśli ktoś oferuje Ci łatwy zysk lub próbuje nakłonić do podania prywatnych danych. Robiąc zakupy w sieci, weryfikuj reputacje sklepów. Dziel się wiedzą z rodziną i znajomymi.
- Zadbaj o kopie zapasowe: Zabezpiecz efekty swojej pracy, muzykę, zdjęcia, cenne dokumenty. Twórz kopie zapasowe i przechowuj je w bezpiecznym miejscu.

Cyberhigiena - CHROŃ SWOJĄ PRYWATNOŚĆ

- Informacje mają wartość: Dane na Twój temat, takie jak historia zakupów czy historia lokalizacji są cenne. Zwracaj uwagę kto i co (aplikacje, strony internetowe) próbuje uzyskać do nich dostęp.
- Dostosuj ustawienia prywatności w serwisach online i na urządzeniach: Dzięki nim, możesz lepiej chronić Twoje dane. Sam decyduj, jak wiele informacji na swój temat chcesz udostępnić innym.
- Pomyśl, zanim udostępnisz: Zwracaj uwagę na przesyłaną do sieci treść, zasięg komunikatu, a także sposób, w jaki może zostać odebrany.

Cyberhigiena - TWÓRZ KULTURĘ BEZPIECZNEJ SIECI

- Twoje zachowanie w sieci ma znaczenie: Stosowanie dobrych praktyk buduje kulturę bezpiecznej sieci. To, co robisz, ma znaczenie (w domu, w pracy, gdziekolwiek jesteś).
- Traktuj innych tak, jak sam chciałbyś być traktowany.
- Wspieraj walkę z cyberprzestępczością: Jeżeli zaobserwujesz niepokojące zjawiska, nie wahaj się o tym poinformować:
  - `https://incydent.cert.pl` (zgłaszanie incydentów naruszających bezpieczeństwo w sieci)
  - `https://dyzurnet.pl` (przyjmowanie zgłoszeń dotyczących nielegalnych treści w Internecie).

Lista kontrolna

https://stojpomyslpolacz.pl/ftp/stc/Lista_kontrolna.pdf

Zagrożenia

Do najpopularniejszych zagrożeń w cyberprzestrzeni należą:

- ataki z użyciem szkodliwego oprogramowania (malware, wirusy, robaki itp.);
- kradzieże tożsamości;
- kradzieże (wyłudzenia), modyfikacje bądź niszczenie danych;
- blokowanie dostępu do usług (mail bomb, DoS oraz DDoS19);
- spam (niechciane lub niepotrzebne wiadomości elektroniczne);
- ataki socjotechniczne (np. phishing, czyli wyłudzanie poufnych informacji przez podszywanie się pod godną zaufania osobę lub instytucję).

Zagrożenia

Co może nam grozić w przyszłości?

- brak zabezpieczeń w IoT
- za “inteligentna” sztuczna inteligencja
- prawo
- moralność

Moralność

- Isaac Asimov w roku 1942 stworzył trzy prawa robotów i przedstawił je w fantastycznym opowiadaniu Zabawa w berka (ang. Runaround).
- Celem tych praw było uregulowanie kwestii stosunków pomiędzy przyszłymi myślącymi maszynami a ludźmi.
- Przedstawiały się one następująco :
  - Robot nie może skrzywdzić człowieka, ani przez zaniechanie działania dopuścić, aby człowiek doznał krzywdy.
  - Robot musi być posłuszny rozkazom człowieka, chyba że stoją one w sprzeczności z Pierwszym Prawem.
  - Robot musi chronić samego siebie, o ile tylko nie stoi to w sprzeczności z Pierwszym lub Drugim Prawem.

Moralność

- Następnie w opowiadaniu Roboty i Imperium (Robots and Empire) Asimov dodał prawo zerowe, które stało się nadrzędne wobec trzech pozostałych[1]:
  - Robot nie może skrzywdzić ludzkości, lub poprzez zaniechanie działania doprowadzić do uszczerbku dla ludzkości.

Moralność

- Aby pogodzić racje etyczno-filozoficzne z interesami biznesmenów, David Langford, brytyjski autor science-fiction, stworzył trzy nowe prawa robotów:
  - Robot nie może działać na szkodę Rządu, któremu służy, ale zlikwiduje wszystkich jego przeciwników
  - Robot będzie przestrzegać rozkazów wydanych przez dowódców, z wyjątkiem przypadków, w których będzie to sprzeczne z trzecim prawem
  - Robot będzie chronił własną egzystencję przy pomocy broni lekkiej, ponieważ robot jest „cholernie drogi”.

Moralność

- W opozycji do tych wszystkich założeń zajmujący się robotyką Mark Tilden stworzył trzy kolejne prawa:
  - Robot musi chronić swoja egzystencję za wszelką cenę.
  - Robot musi otrzymywać i utrzymywać dostęp do źródeł energii.
  - Robot musi nieprzerwanie poszukiwać coraz lepszych źródeł energii.

Moralność

- Teoria przyjaznej sztucznej inteligencji stworzona przez Eliezera Yudkowsky'ego
- Teoria ta ma całkowicie odmienne spojrzenie na problem inteligentnych robotów.
- Zakłada, że można stworzyć roboty kierujące się moralnością, a przy tym przyjaźnie nastawione do człowieka.
- Zdaniem zwolenników tej teorii roboty w przyszłości nie będą podległe ludziom, lecz na skutek rozwoju techniki staną się zupełnie odrębną, obcą formą życia, całkowicie od człowieka niezależną.
- Roboty przyszłości będą tak odmienne od ludzi, że ich antropomorficzny model lansowany przez literaturę popularnonaukową stanie się bezużyteczny.

Moralność

- Koreański rząd obecnie pracuje nad stworzeniem dokumentu regulującego prawa robotów.
- Również w Wielkiej Brytanii Sir David King, główny doradca naukowy rządu Jej Królewskiej Mości uważa, że gdy w przyszłości zostaną stworzone czujące roboty należy dać im te same prawa co ludziom.
- Opinia taka została wyrażona w dokumencie przygotowanym przez firmę konsultująca Outsights oraz zajmującą się badaniami opinii publicznej Ipsos Mori.
- Dokument dotyczy Wielkiej Brytanii w roku 2056, a jego autorzy stwierdzili, że do tego czasu roboty posiadające uczucia mogą być czymś powszechnym.
- W dokumencie czytamy, że jeśli kiedyś rzeczywiście powstanie SI, trzeba będzie przyznać wyposażonym w nią urządzeniom takie prawa i obowiązki jak ludziom

Silna i słaba sztuczna inteligencja

W związku z potencjalną możliwością zbudowania sztucznej inteligencji sformułowano dwa poziomy realizacji tego celu.

- Hipoteza silnej sztucznej inteligencji postuluje możliwość zbudowania systemu rzeczywiście inteligentnego, zdolnego myśleć jak człowiek i posiadającego umysł.
- Hipoteza słabej sztucznej inteligencji polega na budowie systemów, które potrafiłyby działać i rozwiązywać problemy w warunkach pełnej złożoności świata rzeczywistego, tak jakby posiadały umysł i myślały.

Rozróżnienie tych dwóch postulatów ma głównie charakter filozoficzny i etyczny.

Cele AI

W praktyce, celem badań i prac inżynierskich w zakresie sztucznej inteligencji są:

- opracowanie obliczeniowej (algorytmicznej) teorii inteligencji, funkcjonowania ludzkiego mózgu, pamięci, świadomości, emocji, instynktów, itp. W tym sensie sztuczna inteligencja ma związek z biologią, psychologią, filozofią, jak również matematyką i informatyką, ale także innymi dziedzinami nauki i wiedzy.
- budowa inteligentnych systemów (komputerowych) do skutecznego rozwiązywania trudnych zagadnień, zdolnych funkcjonować w normalnym świecie
- W tym sensie sztuczna inteligencja musi współpracować, poza informatyką, z robotyką, mechaniką, mechatroniką i szeregiem dziedzin inżynierskich.

Zadania do rozwiązania

Pomiędzy innymi, sztuczna inteligencja musi zmierzyć się z następującymi zadaniami:

- reprezentacja wiedzy aby móc przyjmować pojawiające się informacje o świecie, rozumieć je, konfrontować z już posiadaną wiedzą
- wnioskowanie aby wyciągać wnioski z pojawiających się informacji, i podejmować decyzje o dalszych działaniach
- uczenie się dla dostosowania się do nowo pojawiających się okoliczności, nieprzewidzianych przez twórców systemu, pojmowania nowych zjawisk, itp.

Zadania do rozwiązania

- rozumienie języka naturalnego jest praktycznie niezbędne aby można było praktycznie sprawdzić zdolności systemu sztucznej inteligencji
- posługiwanie się wizją w celu samodzielnego pozyskiwania wiedzy o świecie
- robotyka czyli praktyczna konstrukcja systemu zdolnego poruszać się i wykonywać działania w świecie rzeczywistym

Zastosowania – komunikacja w języku naturalnym

Technologie przetwarzania języka naturalnego:

- “rozumienie” tekstu, zamiana tekstu na reprezentację formalną
- maszynowe tłumaczenie
- ekstrakcja informacji
- odpowiadanie na pytania
- klasyfikacja tekstu, filtrowanie spamu, itp.

Technologie przetwarzania mowy:

- rozpoznawanie języka mówionego (ASR)
- synteza mowy (TTS)
- systemy dialogowe

Zastosowania – percepcja wizualna

- rozpoznawanie obiektów, znaków
- segmentacja sceny
- rekonstrukcja 3D
- klasyfikacja obrazów

Zastosowania

- Robotyka łączy ze sobą elementy mechaniki i elektroniki (mechatronika), oraz sztucznej inteligencji.
- Gdy przystępujemy do budowy robotów i ich testowania w świecie rzeczywistym, napotykamy problemy daleko wykraczające poza opracowane teorie.
- Zagadnienia, istniejące technologie, zastosowania:
  - planowanie działań
  - sterowanie pojazdami (chodzącymi, jeżdżącymi, latającymi)
  - systemy ratunkowe
  - roboty społeczne – opieka nad ludźmi jej wymagającymi

Filozofia

- Czy można użyć formalne reguły do wyprowadzania poprawnych wniosków?
- Jak umysł wyłania się z fizycznego mózgu?
- Skąd bierze się wiedza?
- Jak wiedza prowadzi do działania?

Matematyka

- Jakie są formalne reguły wyprowadzania poprawnych wniosków?
- Co może być obliczone?
- Jak wnioskować na podstawie niepewnej informacji?

Ekonomia

- Jak powinniśmy podejmować decyzje aby zmaksymalizować zysk?
- Jak powinniśmy to robić gdy inni nie współdziałają z nami?
- Jak powinniśmy to robić gdy zysk będzie odległy w przyszłości?

Neurobiologia

- Jak mózg przetwarza informację?

Psychologia

- Jak ludzie i zwierzęta myślą i działają?

Inżynieria komputerowa

- Jak możemy budować efektywne komputery?

Lingwistyka

- Jak się ma język do myśli?

Największe wyzwanie?

- Testowanie poprawności inteligentnych systemów.
- Przewidzenie ścieżki rozwoju oprogramowania
- Przewidzenie wykorzystania IoT przez hakerów/przestępców itp.

Testowanie

Materiały z `http://icis.pcz.pl/~agrosser/wtest.pdf`

Aksjomaty testowania

- Programu nie da się przetestować całkowicie
- Testowanie jest ryzykowne
- Test nie udowodni braku błędów
- Im więcej błędów znaleziono, tym więcej błędów pozostało do znalezienia
- Nie wszystkie znalezione błędy zostaną naprawione
- Trudno powiedzieć, kiedy błąd jest błędem
- Specyfikacje produktów nigdy nie są gotowe

Definicja błędu

- Oprogramowanie nie robi czegoś co zostało wymienione w jego specyfikacji
- Oprogramowanie wykonuje coś czego według specyfikacji nie powinno robić
- Oprogramowanie robi coś o czym specyfikacji nie wspomina
- Oprogramowanie nie wykonuje czegoś o czym specyfikacja nie wspomina mimo że powinno to być wymienione jako istotną częścią systemu.
- Oprogramowanie jest trudne do zrozumienia, powolne lub skomplikowane.

Definicja błędu

- Oprogramowanie nie robi czegoś co zostało wymienione w jego specyfikacji
- Oprogramowanie wykonuje coś czego według specyfikacji nie powinno robić
- Oprogramowanie robi coś o czym specyfikacji nie wspomina
- Oprogramowanie nie wykonuje czegoś o czym specyfikacja nie wspomina mimo że powinno to być wymienione jako istotną częścią systemu.
- Oprogramowanie jest trudne do zrozumienia, powolne lub skomplikowane.

Poziomy testowania oprogramowania

- Testy jednostkowe (modułowe)
- Testy integracyjne
- Testy funkcjonalne
- Testy systemowe
- Testy akceptacyjne
- Testy w fazie utrzymywania system

Poziomy testowania oprogramowania

Testy jednostkowe (modułowe)

- metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu – np. metod lub obiektów w programowaniu obiektowym lub procedur w programowaniu proceduralnym.
- Testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik (np. zwrócone wartości, stan obiektu, zgłoszone wyjątki) z oczekiwanymi wynikami – tak pozytywnymi, jak i negatywnymi (niepowodzenie działania kodu w określonych sytuacjach również może podlegać testowaniu).

Poziomy testowania oprogramowania

Testy integracyjne

- Testowanie integracyjne wykonywane jest w celu wykrycia błędów w interfejsach i interakcjach pomiędzy modułami.
- Kiedy zbiór komponentów zostanie przetestowany, następnym krokiem jest upewnienie się, że interfejsy pomiędzy owymi komponentami są zdefiniowane poprawnie i współdziałają ze sobą.
- Przykład: Testujemy komunikację pomiędzy modułem przechowującym i udostępniającym zbiór parametrów, a modułem używającym tych parametrów przy inicjacji, np. do wypełnienia pól formularza domyślnymi wartościami.

Poziomy testowania oprogramowania

Testy funkcjonalne

- znane są także jako testy czarnej skrzynki, ponieważ osoba testująca nie ma dostępu do informacji na temat budowy programu, który testuje.
- Często testy takie są wykonywane przez osoby inne niż programiści tworzący program.

Poziomy testowania oprogramowania

Testy systemowe

- Testy systemowe przeprowadzane są, aby stwierdzić czy zintegrowany już system spełnia jako całość wymagania zawarte w specyfikacji.
- Podczas testów systemowych cały system jest weryfikowany pod kątem zgodności z:
  - wymaganiami funkcjonalnymi
  - wymaganiami niefunkcjonalnymi (wydajność, użyteczność, niezawodność)

Poziomy testowania oprogramowania

Testy akceptacyjne

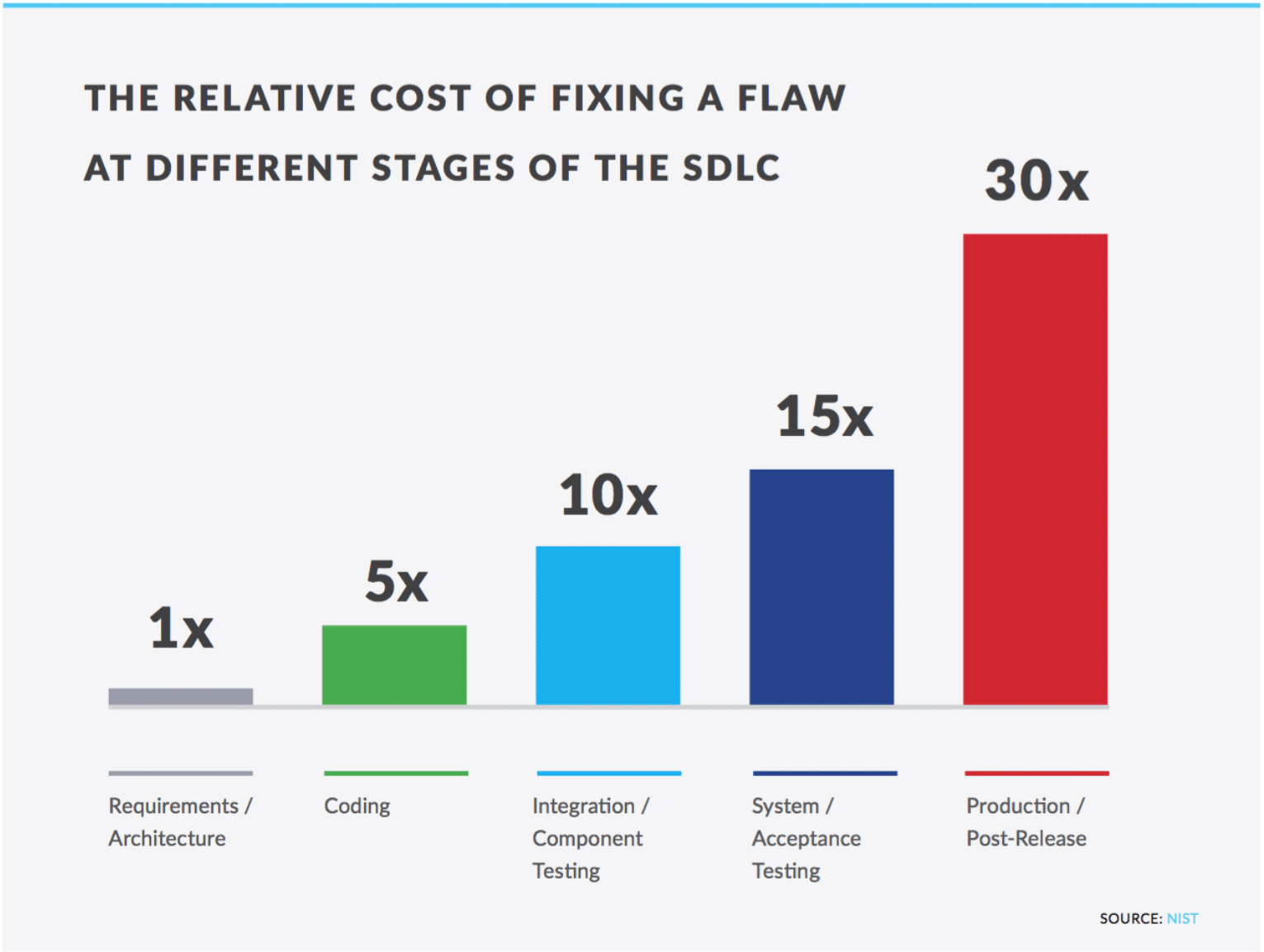
- testy oprogramowania, których celem nie jest wykrycie błędów, a jedynie uzyskanie formalnego potwierdzenia wykonania oprogramowania odpowiedniej jakości.

Poziomy testowania oprogramowania

Testy w fazie utrzymywania systemu

- Raz wydane oprogramowanie jest w użyciu przez lata lub dekady.
- W tym czasie system i jego środowisko podlegają korektom, są zmieniane lub rozszerzane.
- Testowanie podczas utrzymania dokonywane jest na istniejącym systemie i wyzwalane jest modyfikacjami, migracjami lub starzeniem się oprogramowania.

Koszty błędów



Pytest

- Pytest to nowoczesny framework do uruchamiania testów automatycznych w języku Python.
- Może być stosowany do testów jednostkowych, ale sprawdza się bardzo dobrze również przy tworzeniu rozbudowanych testów wyższego poziomu (integracyjnych, end-to-end) dla całych aplikacji czy bibliotek.
- Jego przemyślana konstrukcja ułatwia uzyskanie pożądanych efektów w sytuacjach o wysokim poziomie skomplikowania, a mnogość dostępnych rozszerzeń oraz łatwość tworzenia własnych bibliotek pozwala zastosować go do testowania produktów działających w praktycznie dowolnej technologii.
- Nie trzeba dodawać, że Pytest radzi sobie bardzo dobrze również z testowaniem aplikacji webowych i usług sieciowych, a to dzięki wykorzystaniu bibliotek takich jak Selenium i requests.

Instalacja

Ubuntu

```
$ sudo apt-get install python3-pytest
```

Pierwszy test

```
def test_passing():
    assert (1, 2, 3) == (1, 2, 3)
```

Odpalenie testu

```
$ pytest-3 test_one.py
```

Uwaga! Kropka po nazwie programu po odpaleniu oznacza, że test został wykonany oraz się powiódł. Jeśli potrzebujemy więcej informacji trzeba użyć opcji `-v` lub `-verbose`.

```
$ pytest-3 -v test_one.py
```

Drugi test

```
def test_failing():
    assert (1, 2, 3) == (3, 2, 1)
```

Sposób w jaki pytest pokazuje niepowodzenie jest jednym z powodów dlaczego programiści go uwielbiają. Pokazuje nam gdzie test się nie powiódł.

Pomoc

```
$ pytest-3 -help
```

Gdy podamy polecenie pytest bez argumentu pytest wyszukuje testów rekurencyjnie w katalogu w którym jesteśmy.

Trzeci test

```
from collections import namedtuple

Task = namedtuple('Task', ['summary', 'owner', '
    done', 'id'])
Task.__new__.__defaults__ = (None, None, False,
    None)

def test_defaults():
    """Using no parameters should invoke defaults.
       """
    t1 = Task()
    t2 = Task(None, None, False, None)
    assert t1 == t2

def test_member_access():
    """Check .field functionality of namedtuple."""
    t = Task('buy␣milk', 'brian')
    assert t.summary == 'buy␣milk'
    assert t.owner == 'brian'
```

Czwarty test

```
from collections import namedtuple
Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
Task.__new__.__defaults__ = (None, None, False, None)

def test_asdict():
    """_asdict() should return a dictionary."""
    t_task = Task('do something', 'okken', True, 21)
    t_dict = t_task._asdict()
    expected = {'summary': 'do something',
                'owner': 'okken',
                'done': True,
                'id': 21}
    assert t_dict == expected

def test_replace():
    """replace() should change passed in fields."""
    t_before = Task('finish book', 'brian', False)
    t_after = t_before._replace(id=10, done=True)
    t_expected = Task('finish book', 'brian', True, 10)
    assert t_after == t_expected
```

Pytest

- Pliki z testami powinny się nazywać `test_something.py` lub `something_test.py`.
- Metody i funkcje testowe powinny się nazywać `test_something`.
- Klasy testowe powinny nazywać się `TestSomething`

Pytest – wyjścia

- PASSED (.)
- FAILED (F)
- SKIPPED (s)
- ERROR (E):

Uruchamianie jednego testu

```
$ pytest -v tasks/test_four.py::test_asdict
```

Pisanie funkcji testowych

Pisanie funkcji testowych

- Do nauki użyjemy przykładowego projektu `Tasks`.
- Katalog ze źródłami jest oddzielony od katalogu z testami.
- Katalog z testami jest podzielony na katalog z testami funkcjonalnymi oraz testami jednostkowymi.
- Testy funkcjonalne i jednostkowe są rozdzielane na własne katalogi.
- Jest to dobra decyzja, ale nie wymagana.
- Jednak organizowanie plików testowych na wiele katalogów umożliwiaja łatwe uruchomienie podzbioru testów.
- Dobrze przechowywać testy funkcjonalne i testy jednostkowe oddzielne, ponieważ testy funkcjonalne powinny się niepowieść tylko wtedy, gdy celowo zmieniamy funkcjonalność systemu, podczas gdy testy jednostkowe mogą się niepowieść w trakcie refaktoryzacji lub zmiany implementacji.

Pisanie funkcji testowych

- Projekt zawiera dwa typy plików `__init__.py`: te w katalogu `src/` oraz te w katalogu `tests/`.
- Plik `src/tasks/__ init__.py` mówi Pythonowi, że katalog jest pakietem.
- Zachowuje się również jako główny interfejs dla pakietu, gdy ktoś używa `import tasks`.
- Zawiera kod do zaimportowania konkretnych funkcji z `api.py`, dzięki czemu `cli.py` i nasze pliki testowe mogą uzyskać dostęp do funkcji pakietu takie jak `tasks.add()` zamiast wykonywania `tasks.api.add()`.

Pisanie funkcji testowych

- Pliki `tests/func/__init__.py` i `tests/unit/__init__.py` są puste.
- Mówią Pythonowi aby przeszedł do jedn katalog w górę, aby znaleźć katalog główny katalogu testowego i szukał pliku `pytest.ini`.
- Plik `pytest.ini` jest opcjonalny. Zawiera konfigurację pytestów.
- Może zawierać dyrektywy które zmieniają zachowanie pytest, takie jak konfigurowanie listy opcji, które zawsze będą używanę.
- Zostanie on dokładniej omówiony w późniejszym czasie.

Pisanie funkcji testowych

- Plik `conftest.py` jest również opcjonalny. Jest uznawany przez pytest jako “lokalna wtyczka” i może zawierać funkcje haka i fixture.
- Funkcje haka są sposobem na wstawianie kodu do części procesu wykonywania pytestów, aby zmienić sposób działania pytest.
- Fixture to ustawienia i funkcje rozłączania, które są uruchamiane przed i po teście i mogą być używane do reprezentowania zasobów i danych używanych przez testy.
- Fixture i haki, które są używane przez testy w wielu podkatalogach powinny być zawarte w `tests/conftest.py`.
- Można mieć wiele plików `conftest.py`; na przykład możesz go mieć w katalogu `test` i po jednym dla każdego testowanego podkatalogu.

tasks_proj/tests/unit/test_task.py

```
"""Test the Task data type."""
from tasks import Task


def test_asdict():
    """_asdict() should return a dictionary."""
    t_task = Task('do something', 'okken', True, 21)
    t_dict = t_task._asdict()
    expected = {'summary': 'do something',
                'owner': 'okken',
                'done': True,
                'id': 21}
    assert t_dict == expected


def test_replace():
    """replace() should change passed in fields."""
    t_before = Task('finish book', 'brian', False)
    t_after = t_before._replace(id=10, done=True)
    t_expected = Task('finish book', 'brian', True, 10)
    assert t_after == t_expected
```

tasks_proj/tests/unit/test_task.py

```
def test_defaults():
    """Using no parameters should invoke defaults."""
    t1 = Task()
    t2 = Task(None, None, False, None)
    assert t1 == t2


def test_member_access():
    """Check .field functionality of namedtuple."""
    t = Task('buy milk', 'brian')
    assert t.summary == 'buy milk'
    assert t.owner == 'brian'
    assert (t.done, t.id) == (False, None)
```

Pisanie funkcji testowych

- Najlepszym sposobem na umożliwienie testom `import tasks` zadań lub `from tasks import something` jest instalowanie zadań lokalnie za pomocą `pip`(menadżera pakietów w Pythonie).
- Jest to możliwe, ponieważ istnieje plik `setup.py`
- Zainstalujmy `tasks`, uruchamiając `pip install .` lub `pip install -e .` z katalogu `tasks_proj`
- Możemy również uruchomić `pip install -e tasks_proj` z jednego katalogu w górę:

tasks_proj/tests/unit/test_task.py

```
$ cd /path/to/code
$ pip3 install ./tasks_proj/
$ pip3 install --no-cache-dir ./tasks_proj/
```

tasks_proj/tests/unit/test_task.py

```
$ cd /path/to/code/ch2/tasks_proj/tests/unit
$ pytest-3 test_task.py
```

Używanie `assert`

- Kiedy piszesz funkcje testowe, normalna instrukcja assert Pythona jest twoim podstawowym narzędziem do komunikowania niepowodzenia testu.
- Prostota tego w pytest jest genialna. To właśnie sprawia, że wielu programistów używa Pythona do innych frameworków.
- W pytest można użyć `assert <expression>` z dowolnym wyrażeniem.
- Jeśli wyrażenie zostanie ocenione jako `False` po przekonwertowaniu na wartość `bool`, test zakończy się niepowodzeniem.
- Pytest zawiera funkcję zwaną przepisywaniem asercji, która przechwytuje wywołania `assert` i zastępuje je czymś, co może ci powiedzieć więcej o tym, dlaczego twoje asercje się nie powiodły.
- Zobaczmy, jak pomocne jest to przepisywanie, patrząc na kilka błędów asercji:

tasks_proj/tests/unit/test_task.py

```
"""Use the Task type to show test failures."""
from tasks import Task


def test_task_equality():
    """Different tasks should not be equal."""
    t1 = Task('sit there', 'brian')
    t2 = Task('do something', 'okken')
    assert t1 == t2


def test_dict_equality():
    """Different tasks compared as dicts should not be equal."""
    t1_dict = Task('make sandwich', 'okken')._asdict()
    t2_dict = Task('make sandwich', 'okkem')._asdict()
    assert t1_dict == t2_dict
```

Używanie `assert`

- Dużo informacji!
- Dla każdego niepowodzenia testu dokładna linia awarii jest pokazana znakiem “>”
- Linie “E” pokazują dodatkowe informacje na temat błędu asercji, aby pomóc Ci dowiedzieć się, co poszło nie tak.
- Spróbujemy jeszcze raz z flagą -v, jak zasugerowano w komunikacie o błędzie.
- pytest nie tylko odnalazł obie różnice, ale pokazał nam dokładnie, gdzie są te różnice.

Oczekiwanie wyjątków

- Wyjątki mogą być zgłaszane w kilku miejscach interfejsu Tasks API.
- Przejrzyjmy funkcje znalezione w `task/api.py`:

Funkcje Tasks API

```
def add(task): # type: (Task) -> int
def get(task_id): # type: (int) -> Task
def list_tasks(owner=None): # type: (str|None) -> list of Task
def count(): # type: (None) -> int
def update(task_id, task): # type: (int, Task) -> None
def delete(task_id): # type: (int) -> None
def delete_all(): # type: () -> None
def unique_id(): # type: () -> int
def start_tasks_db(db_path, db_type): # type: (str, str) -> None
def stop_tasks_db(): # type: () -> None
```

Oczekiwanie wyjątków

- Istnieje zgoda między kodem CLI w pliku `cli.py` a kodem interfejsu API w pliku `api.py` na temat typów wysyłanych do funkcji API.
- Te wywołania API są miejscem gdzie oczekiwałabym wyjątków, gdyby typ był błędny.
- Aby upewnić się, że funkcje te powodują niepoprawne wywoływanie wyjątków, użyjmy niewłaściwego typu w funkcji testowej, aby celowo spowodować wyjątki TypeError i użyjmy z `pytest.raises(<expected exception>)`, na przykład:

tasks_proj/tests/func/test_api_exceptions

```
"""Test for expected exceptions from using the API wrong."""

import pytest
import tasks


def test_add_raises():
    """add() should raise an exception with wrong type param."""
    with pytest.raises(TypeError):
        tasks.add(task='not a Task object')
```

- W `test_add_raises()` instrukcja `with pytest.raises(TypeError):` mówi, że cokolwiek znajduje się w następnym bloku kodu, powinno wywołać wyjątek `TypeError`.
- Jeśli nie zostanie zgłoszony żaden wyjątek, test kończy się niepowodzeniem.
- Jeśli test spowoduje inny wyjątek, nie powiedzie się.

tasks_proj/tests/func/test_api_exceptions

- Właśnie sprawdziliśmy typ wyjątku w `test_add_raises()`.
- Można również sprawdzić parametry do wyjątku.
- W przypadku `start_tasks_db(db_path, db_type)`, `db_type` musi nie tylko być ciągiem znaków, ale musi być albo “malutki”, albo “mongo”.
- Możesz sprawdzić, czy komunikat wyjątku jest poprawny, dodając jako `excinfo`:

```
def test_start_tasks_db_raises():
    """Make sure unsupported db raises an exception."""
    with pytest.raises(ValueError) as excinfo:
        tasks.start_tasks_db('some/great/path', 'mysql')
    exception_msg = excinfo.value.args[0]
    assert exception_msg == "db_type␣must␣be␣a␣'tiny'␣or␣'mongo'
        "
```

Oczekiwanie wyjątków

- Pozwala nam to dokładniej przyjrzeć się wyjątkowi.
- Nazwa zmiennej, którą wstawisz za `as` (w tym przypadku `excinfo`) jest wypełniona informacją o wyjątku i jest typu `ExceptionInfo`.
- W naszym przypadku chcemy się upewnić, że pierwszy (i jedyny) parametr do wyjątu pasuje do ciągu znaków.

Oznaczanie funkcji testowych

- pytest zapewnia mechanizm pozwalający umieścić znaczniki na funkcjach testowych.
- Test może mieć więcej niż jeden znacznik, a marker może być w wielu testach.
- Powiedzmy, że chcemy uruchomić podzbiór naszych testów jako szybki “smoke test”, aby mieć poczucie, czy lub nie jest pewna poważna dziura w systemie.
- “Smoke test” nie są kompleksowymi zestawami testowymi, ale wybranymi podzbiorami, które można szybko uruchomić i dać programistom przyzwoite pojęcie o zdrowiu wszystkich części system.
- Aby dodać pakiet smoke testów do projektu Tasks, możemy dodać `@mark.pytest.smoke` do niektórych testów (`test_api_exceptions.py`
- zwróćmy uwagę, że markery `smoke` i `get` nie są wbudowane w pytest, po prostu autor książki je wymyślił:

tasks_proj/tests/func/test_api_exceptions

```
@pytest.mark.smoke
def test_list_raises():
    """list() should raise an exception with wrong type param.
       """
    with pytest.raises(TypeError):
        tasks.list_tasks(owner=123)


@pytest.mark.get
@pytest.mark.smoke
def test_get_raises():
    """get() should raise an exception with wrong type param."""
    with pytest.raises(TypeError):
        tasks.get(task_id='123')
```

Oznaczanie funkcji testowych

Teraz odpalmy tylko te testy które są oznaczone daną nazwą.

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v -m 'smoke' test_api_exceptions.py
```

- Pamiętaj, że `-v` jest skrótem od `-verbose` i pozwala nam zobaczyć nazwy testów które są uruchamiane.
- Użycie `-m 'smoke'` uruchamia oba testy oznaczone `@pytest.mark.smoke`.
- Użycie `-m 'get'` uruchamia jeden test oznaczony `@pytest.mark.get`.
- Wyrażenie po `-m` może używać `and`, `or` oraz `not` aby łączyć wiele markerów:

```
$ pytest-3 -v -m 'get' test_api_exceptions.py
$ pytest-3 -v -m 'smoke and get' test_api_exceptions.py
```

Oznaczanie funkcji testowych

- Możemy używać również `not` jako:

```
$ pytest-3 -v -m 'smoke and not get' test_api_exceptions.py
```

Dodanie `-m 'smoke and not get'` wybrało test, który został zaznaczony `@pytest.mark.smoke`, ale nie `@pytest.mark.get`.

Oznaczanie funkcji testowych

- Poprzednie testy nie wydają się jeszcze rozsądnym zestawem testów smoke.
- W rzeczywistości nie dotknęliśmy bazy danych ani nie dodaliśmy żadnych zadań.
- Z pewnością smoke test by to zrobił.
- Dodajmy kilka testów, które wyglądają na dodanie zadania, i użyjmy jednego z nich jako część naszego pakietu smoke testów:

Oznaczanie funkcji testowych

```
"""Test the tasks.add() API function."""

import pytest
import tasks
from tasks import Task


def test_add_returns_valid_id():
    """tasks.add(<valid task>) should return an integer."""
    # GIVEN an initialized tasks db
    # WHEN a new task is added
    # THEN returned task_id is of type int
    new_task = Task('do something')
    task_id = tasks.add(new_task)
    assert isinstance(task_id, int)
```

Oznaczanie funkcji testowych

```
@pytest.mark.smoke
def test_added_task_has_id_set():
    """Make sure the task_id field is set by tasks.add()."""
    # GIVEN an initialized tasks db
    #   AND a new task is added
    new_task = Task('sit␣in␣chair', owner='me', done=True)
    task_id = tasks.add(new_task)

    # WHEN task is retrieved
    task_from_db = tasks.get(task_id)

    # THEN task_id matches id field
    assert task_from_db.id == task_id
```

Oznaczanie funkcji testowych

- Oba te testy mają komentarz `GIVEN an initialized tasks db`, a jednak nie ma bazy danych zainicjalizowanej w teście.
- Możemy zdefiniować fixture, aby baza danych została zainicjowana przed testem i wyczyszczona po teście:

```
@pytest.fixture(autouse=True)
def initialized_tasks_db(tmpdir):
    """Connect to db before testing, disconnect after."""
    # Setup : start db
    tasks.start_tasks_db(str(tmpdir), 'tiny')

    yield  # this is where the testing happens

    # Teardown : stop db
    tasks.stop_tasks_db()
```

Oznaczanie funkcji testowych

- Fixture `tmpdir`, użyte w tym przykładzie jest wbudowanym fixture (omówimy to później).
- `autouse` w naszym teście wskazuje, że wszystkie testy w tym pliku będą korzystać z fixture.
- Kod przed `yield` jest wykonywany przed każdym testem; kod po `yield` jest wykonywany po teście.
- `yield` może w razie potrzeby przywrócić dane do testu.
- Tutaj potrzebujemy jakiegoś sposobu na skonfigurowanie bazy danych do testowania, więc nie mogliśmy już dłużej czekać, aby pokazać fixture.
- pytest obsługuje również staroświeckie funkcje konfiguracji i rozłączania, takie jak używane w unittest i nose, ale nie są tak przyjemne w nauce i obsłudze.

Oznaczanie funkcji testowych

Uruchommy nasz pakiet smoke testów:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest-3 -v -m 'smoke'
```

To pokazuje ze oznaczone testy z różnych plików można wykonywać wszystkie razem.

Oznaczanie funkcji testowych

- Możemy też oznaczać funkcje nazwami zarezerwowanymi np. `skip`, `skipx`.
- Markery te pozwalają na ominięcie testów, których nie chcemy odpalić.
- Na przykład powiedzmy, że nie byliśmy pewni w jaki sposób polecenie `tasks.unique_id ()` miało działać.
- Czy każde jego wywołanie zwraca inny numer? Czy jest to tylko numer, który już nie istnieje w bazie danych?
- Najpierw napiszmy test (pamiętajmy, że fixture `initialized_tasks_db` również znajduje się w tym pliku, ale nie jest tutaj pokazane):

Oznaczanie funkcji testowych

```
"""Test tasks.unique_id()."""

import pytest
import tasks


def test_unique_id():
    """Calling unique_id() twice should return different numbers
       ."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2
```

Odpalmy go:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 test_unique_id_1.py
```

Oznaczanie funkcji testowych

- Po nieco dłuższym spojrzeniu na API, widzimy, że docstring mówi “Return an integer that does not exist in the db”.
- Możemy po prostu zmienić test. Zamiast tego, po prostu zaznaczmy pierwszy do pominięcia:

Oznaczanie funkcji testowych

```
@pytest.mark.skip(reason='misunderstood the API')
def test_unique_id_1():
    """Calling unique_id() twice should return different numbers
       ."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2


def test_unique_id_2():
    """unique_id() should return an unused id."""
    ids = []
    ids.append(tasks.add(Task('one')))
    ids.append(tasks.add(Task('two')))
    ids.append(tasks.add(Task('three')))
    # grab a unique id
    uid = tasks.unique_id()
    # make sure it isn't in the list of existing ids
    assert uid not in ids
```

Odpalmy go:

```
$ pytest-3 -v test_unique_id_2.py
```

Oznaczanie funkcji testowych

- Teraz załóżmy, że z jakiegoś powodu decydujemy, że pierwszy test również powinien być ważny i zamierzamy wykonać tę pracę w wersji 0.2.0 pakietu. Możemy pozostawić test na miejscu i użyć zamiast niego `skipif`:

```
@pytest.mark.skipif(tasks.__version__ < '0.2.0',
                    reason='not␣supported␣until␣version␣0.2.0')
def test_unique_id_1():
    """Calling unique_id() twice should return different numbers
       ."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2
```

- Wyrażenie, które przekazujemy do funkcji skipif (), może być dowolnym poprawnym wyrażeniem w języku Python.
- W tym przypadku sprawdzamy wersję pakietu.

Odpalmy go:

```
$ pytest-3 test_unique_id_3.py
```

Oznaczanie funkcji testowych

- s . oznacza, że jeden test został pominienty a jeden test się powiódł.

```
$ pytest-3 -v test_unique_id_3.py
```

- Nadal nie wiemy czemu test został pominięty

```
$ pytest-3 -rs test_unique_id_3.py
```

- Zobaczmy co robi opcja -r

```
$ pytest-3 --help
```

Oznaczanie testów – niepowodzenie

- Przy użyciu znaczników `skip` oraz `skipif` próba wykonania testu nie jest nawet podejmowana.
- Za pomocą znacznika `xfail` mówimy pytest o uruchomieniu funkcji testowej, ale oczekujemy, że test się nie powiedzie.
- Zmodyfikujmy test funkcji `unique_id()` aby użyć `xfail`.

Oznaczanie funkcji testowych

```
@pytest.mark.xfail(tasks.__version__ < '0.2.0',
                   reason='not␣supported␣until␣version␣0.2.0')
def test_unique_id_1():
    """Calling unique_id() twice should return different numbers
       ."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2


@pytest.mark.xfail()
def test_unique_id_is_a_duck():
    """Demonstrate xfail."""
    uid = tasks.unique_id()
    assert uid == 'a␣duck'


@pytest.mark.xfail()
def test_unique_id_not_a_duck():
    """Demonstrate xpass."""
    uid = tasks.unique_id()
    assert uid != 'a␣duck'
```

Oznaczanie testów – niepowodzenie

- Pierwszy test jest taki sam jak poprzednio ale ze znacznikiem `xfail`
- Dwa następne testy są oznaczone jako `xfail` i różnią się tylko `==` i `!=`.
- Jeden z nich musi sie wykonać poprawnie.

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 test_unique_id_4.py
```

- `x` jest dla `XFAIL`, co oznacza “oczekuje się niepowodzenia”.
- `X` jest dla `XPASS` lub “Oczekuje się porażki, ale test przeszedł”"
- `-verbose` wyświetla dłuższe opisy:

Oznaczanie testów – niepowodzenie

```
$ pytest-3 -v test_unique_id_4.py
```

- Możesz skonfigurować program pytest, aby zgłaszać testy, które pomyślnie przeszły, ale zostały oznaczone `xfail` należy zgłosić jako `FAIL`.
- Odbywa się to w pliku `pytest.ini`:

```
[pytest]
xfail_strict=true
```

Odpalanie podzbioru testów

- Do tej pory było omówione w jaki sposób można umieszczać znaczniki w testach i przeprowadzać testy na podstawie markeów.
- Można uruchomić podzestaw testów na kilka innych sposobów.
- Można odpalić wszystkie testy lub można wybrać pojedynczy katalog, plik, klasę w pliku lub indywidualny test w pliku lub klasie.
- Nie widzieliśmy jeszcze klas testowych, więc przyjrzymy się jednej niebawem.

Odpalanie podzbioru testów – katalog

- Aby odpalić wszystkie testy z danego katalogu trzeba użyć tego katalogu jako argumentu programu pytest.

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest-3 tests/func --tb=no
$ pytest-3 -v tests/func --tb=no
```

Odpalanie podzbioru testów – plik/moduł

- Aby odpalić testy z jednego pliku trzeba użyć tego pliku jako argumentu programu pytest.

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest-3 tests/func/test_add.py
```

Odpalanie podzbioru testów – pojedynczy test

- Aby odpalić jeden test z danego pliku trzeba użyć tego pliku jako argumentu programu pytest oraz po nazwie pliku postawić `::` oraz napisać nazwę funkcji testowej.

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest-3 -v tests/func/test_add.py::test_add_returns_valid_id
```

Pojedyncza klasa testowa

- Klasy testowe są sposobem aby pogrupować testy.

```
class TestUpdate():
    """Test expected exceptions with tasks.update()."""

    def test_bad_id(self):
        """A non-int id should raise an excption."""
        with pytest.raises(TypeError):
            tasks.update(task_id={'dict instead': 1},
                         task=tasks.Task())

    def test_bad_task(self):
        """A non-Task task should raise an excption."""
        with pytest.raises(TypeError):
            tasks.update(task_id=1, task='not a task')
```

Pojedyncza klasa testowa

- Ponieważ są to dwa powiązane testy, które testują funkcję `update()`, rozsądne jest grupowanie ich w klasie.
- Aby uruchomić tylko tę klasę, zróbmy tak jak z funkcjami i dodajmy `::`, a następnie nazwę klasy do parametru pliku:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest-3 -v tests/func/test_api_exceptions.py::TestUpdate
```

Pojedyncza metoda testowa z klasy testowej

- Jeśli nie chcemy odpalać wszystkich testów z danej klasy, a tylko jedna metodę, wystarczy dodać kolejne `::`, a następnie nazwę metody z klasy testowej:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest-3 -v tests/func/test_api_exceptions.py::TestUpdate::
    test_bad_id
```

Pamiętajmy, że składnia dotycząca uruchamiania zestawu testów według katalogu, pliku, funkcji, klasy i metody nie musi być zapamiętywana. Format jest taki sam, jak lista funkcji testowych po uruchomieniu programu pytest -v.

Zbiór testów bazujących na tej samej nazwie testu

- Opcja `-k` zezwala nam na podanie wyrażenia, które ma występować w nazwie testu, który chcemy odpalić.

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest-3 -v -k _raises
```

- Możemy użyć `and` oraz `not` aby pozbyć się funkcji `test_delete_raises()` z sesji:

```
$ pytest-3 -v -k "_raises and not delete"
```

Testy parametryzowane

- Wysyłanie niektórych wartości za pośrednictwem funkcji i sprawdzanie danych wyjściowych w celu upewnienia się, że są poprawne, jest typowym wzorcem w testowaniu oprogramowania.
- Jednak wywołanie funkcji raz z jednym zestawem wartości i jednym sprawdzeniem poprawności nie wystarcza, aby w pełni przetestować większość funkcji.
- Testowanie parametryzowane to sposób na wysyłanie wielu zestawów danych za pomocą tego samego testu oraz otrzymanie raportu z pytest, jeśli którykolwiek z zestawów się nie powiódł.
- Aby pomóc w zrozumieniu problemu, który próbuje rozwiązać sparametryzowane testowanie, weźmy prosty test dla `add()`:

Testy parametryzowane

```
import pytest
import tasks
from tasks import Task


def test_add_1():
    """tasks.get() using id returned from add() works."""
    task = Task('breathe', 'BRIAN', True)
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    # everything but the id should be the same
    assert equivalent(t_from_db, task)


def equivalent(t1, t2):
    """Check two tasks for equivalence."""
    # Compare everything but the id field
    return ((t1.summary == t2.summary) and
            (t1.owner == t2.owner) and
            (t1.done == t2.done))
```

Testy parametryzowane

```
@pytest.fixture(autouse=True)
def initialized_tasks_db(tmpdir):
    """Connect to db before testing, disconnect after."""
    tasks.start_tasks_db(str(tmpdir), 'tiny')
    yield
    tasks.stop_tasks_db()
```

- Po utworzeniu obiektu `Task` jego pole `id` jest ustawione na `None`.
- Po dodaniu i pobraniu z bazy danych zostanie ono ustawione.
- Dlatego nie możemy po prostu użyć `==`, aby sprawdzić, czy nasze zadanie zostało dodane i odzyskane poprawnie.
- Pomocnicza funkcja `equivalent()` sprawdza wszystkie pola oprócz pola `id`.
- Fixture `autouse` dołączono, aby upewnić się, że baza danych jest dostępna.

Testy parametryzowane

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v test_add_variety.py::test_add_1
```

- Test wydaje się rozsądny.
- Jednak jest to testowanie jednego przykładowego zadania.
- Co jeśli chcemy przetestować wiele wariantów zadania?
- Możemy użyć `@pytest.mark.parametrize(argnames, argvalues)`, aby przekazać wiele danych do tego samego testu:

Testy parametryzowane

```
@pytest.mark.parametrize('task',
                         [Task('sleep', done=True),
                          Task('wake', 'brian'),
                          Task('breathe', 'BRIAN', True),
                          Task('exercise', 'BrIaN', False)])
def test_add_2(task):
    """Demonstrate parametrize with one parameter."""
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

- Pierwszym argumentem `parametrize()` jest ciąg znaków z rozdzieloną przecinkami listą nazw - `'task'`.
- Drugi argument to lista wartości, która w naszym przypadku jest listą obiektów `Task`.
- pytest uruchomi ten test raz dla każdego zadania i zgłosi każdy jako osobny test:

Testy parametryzowane

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v test_add_variety.py::test_add_2
```

- To użycie `parametrize()` działa dla naszych celów.
- Przekażmy jednak zadania jako krotki, aby zobaczyć, jak działałyby różne parametry testowe:

```
@pytest.mark.parametrize('summary, owner, done',
                         [('sleep', None, False),
                          ('wake', 'brian', False),
                          ('breathe', 'BRIAN', True),
                          ('eat eggs', 'BrIaN', False),
                          ])
def test_add_3(summary, owner, done):
    """Demonstrate parametrize with multiple parameters."""
    task = Task(summary, owner, done)
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

Testy parametryzowane

- Kiedy używasz typów, które są łatwe do przekonwertowania na łańcuchy znaków, identyfikator testu używa wartości parametrów w raporcie, aby był czytelny:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v test_add_variety.py::test_add_3
```

- Możesz użyć tego całego identyfikatora testu – zwanego węzłem w terminologii pytest – aby ponownie uruchomić test:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v test_add_variety.py::test_add_3[sleep-None-False]
```

Testy parametryzowane

- Używaj cudzysłowów, jeśli w identyfikatorze są spacje:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v "test_add_variety.py::test_add_3[eat␣eggs-BrIaN-
    False]"
```

Testy parametryzowane

- Teraz wróćmy do listy wersji zadań, ale przenieśmy listę zadań do zmiennej poza funkcją:

```
tasks_to_try = (Task('sleep', done=True),
                Task('wake', 'brian'),
                Task('wake', 'brian'),
                Task('breathe', 'BRIAN', True),
                Task('exercise', 'BrIaN', False))


@pytest.mark.parametrize('task', tasks_to_try)
def test_add_4(task):
    """Slightly different take."""
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

- Jest to wygodne, a kod wygląda ładnie, ale wyniki są trudne do zinterpretowania:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v test_add_variety.py::test_add_4
```

Testy parametryzowane

- Czytelność wersji z wieloma parametrami jest dobra, ale tak samo jest z listą obiektów zadań.
- Możemy użyć parametru opcjonalnego `ids` dla `parametrize()`, aby utworzyć własne identyfikatory dla każdego zestawu danych zadania.
- Parametr `ids` musi być listą ciągów znaków o tej samej długości co liczba zestawów danych.
- Ponieważ jednak przypisaliśmy nasz zestaw danych do nazwy zmiennej, `tasks_to_try`, możemy jej użyć do wygenerowania identyfikatorów:

Testy parametryzowane

```
task_ids = ['Task({},{},{})'.format(t.summary, t.owner, t.done)
            for t in tasks_to_try]


@pytest.mark.parametrize('task', tasks_to_try, ids=task_ids)
def test_add_5(task):
    """Demonstrate ids."""
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v test_add_variety.py::test_add_5
```

- Te identyfikatory mogą być użyte do odpalenia testów:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v "test_add_variety.py::test_add_5[Task(exercise,
    BrIaN,False)]"
```

Testy parametryzowane

- Można także zastosować `parametrize()` do klas. Gdy to zrobimy, te same zestawy danych zostaną wysłane do wszystkich metod testowych w klasie:

```
@pytest.mark.parametrize('task', tasks_to_try, ids=task_ids)
class TestAdd():
    """Demonstrate parametrize and test classes."""

    def test_equivalent(self, task):
        """Similar test, just within a class."""
        task_id = tasks.add(task)
        t_from_db = tasks.get(task_id)
        assert equivalent(t_from_db, task)

    def test_valid_id(self, task):
        """We can use the same data or multiple tests."""
        task_id = tasks.add(task)
        t_from_db = tasks.get(task_id)
        assert t_from_db.id == task_id
```

Testy parametryzowane

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v test_add_variety.py::TestAdd
```

Możemy także zidentyfikować parametry, dołączając identyfikator bezpośrednio obok wartości parametru podczas przechodzenia na listę w ramach dekoratora `@pytest.mark.parametrize()`. Robi się to za pomocą składni `pytest.param(<value>, id = "something")`:

```
@pytest.mark.parametrize('task', [
    pytest.param(Task('create'), id='just␣summary'),
    pytest.param(Task('inspire', 'Michelle'), id='summary/owner'
        ),
    pytest.param(Task('encourage', 'Michelle', True), id='
        summary/owner/done')])
def test_add_6(task):
    """Demonstrate pytest.param and id."""
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

Testy parametryzowane

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest-3 -v test_add_variety.py::test_add_6
```

Jest to przydatne, gdy identyfikator nie może pochodzić z wartości parametru.

Fixtures