Testowanie i Jakość Oprogramowania

Akademia Tarnowska

Katedra Informatyki

Tomasz Gądek

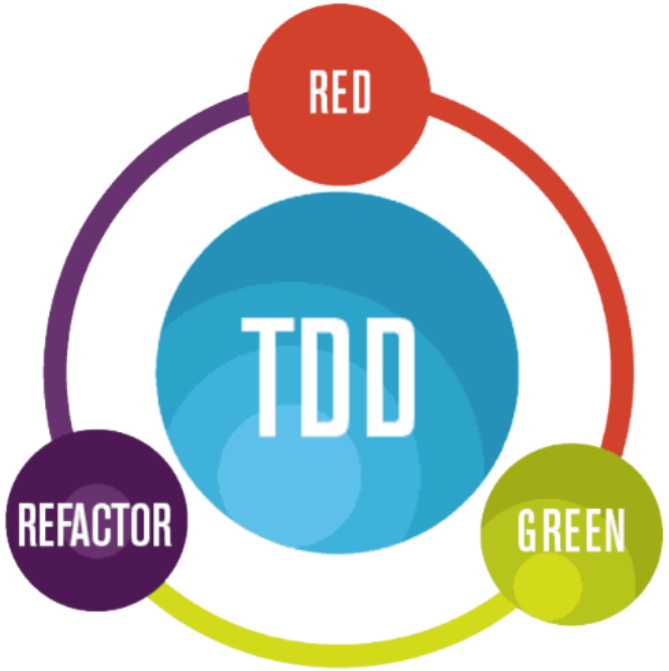
Wykład 13: TDD, E2E

Wprowadzenie do TDD

Test-Driven Development (TDD) to iteracyjny proces tworzenia oprogramowania, w którym testy jednostkowe są pisane **przed** napisaniem kodu implementującego daną funkcjonalność.

Kluczowa idea: Zanim napiszesz choćby jedną linijkę kodu funkcji, najpierw piszesz test, który sprawdza, jak ta funkcja powinna działać.

Cykl TDD: Red-Green-Refactor



Cykl TDD: Red-Green-Refactor

TDD opiera się na krótkich, powtarzalnych cyklach:

1. **Red (Czerwony):** Napisz test jednostkowy dla nowej funkcjonalności. Uruchom testy – nowy test powinien **nie przejść** (stąd **Czerwony**).
2. **Green (Zielony):** Napisz minimalną ilość kodu, która pozwoli **zaliczyć** napisany wcześniej test. Skup się tylko na tym, aby test przeszedł, nie martw się jeszcze o optymalizację czy pełną implementację.
3. **Refactor (Refaktoryzacja):** Popraw napisany kod. Możesz teraz refaktoryzować kod, aby był czystszy, bardziej czytelny i wydajny, **upewniając się, że wszystkie testy nadal przechodzą**.

Dlaczego warto stosować TDD?

- **Lepsze zrozumienie wymagań:** pisanie testów przed kodem zmusza do dokładnego przemyślenia, co dana funkcja ma robić i jakie ma mieć przypadki użycia.
- **Wyższa jakość kodu:** Kod napisany w TDD jest zazwyczaj bardziej modułowy, łatwiejszy do testowania i mniej podatny na błędy.
- **Dokumentacja w formie testów:** Testy jednostkowe stanowią żywą dokumentację, pokazującą, jak poszczególne części kodu powinny działać.
- **Większa pewność przy refaktoryzacji:** Dzięki pokryciu kodu testami, refaktoryzacja staje się bezpieczniejsza – wiesz, że niepoprawne zmiany zostaną szybko wykryte.
- **Częstsze i mniejsze commity:** Cykl Red-Green-Refactor naturalnie prowadzi do mniejszych i częstszych commitów.

Jak stosować TDD? Krok po kroku

1. **Wybierz małą, konkretną funkcjonalność do zaimplementowania.**
2. **Napisz test jednostkowy, który definiuje oczekiwane zachowanie tej funkcjonalności.** Test powinien być jak najbardziej konkretny i dotyczyć jednego aspektu funkcjonalności.
3. **Uruchom wszystkie testy.** Nowo napisany test powinien zakończyć się niepowodzeniem.
4. **Napisz minimalną ilość kodu, który jest niezbędny do zaliczenia tego konkretnego testu.** Nie pisz niczego więcej!

Jak stosować TDD? Krok po kroku

5. **Uruchom ponownie wszystkie testy.** Wszystkie testy, włącznie z nowym, powinny teraz zakończyć się sukcesem.
6. **Refaktoryzuj kod.** Popraw strukturę, czytelność i wydajność kodu, upewniając się, że wszystkie testy nadal przechodzą.
7. **Powtórz kroki 1-6 dla kolejnych małych funkcjonalności.**

Prosta funkcja dodająca (TDD)

> Zaimplementujemy prostą funkcję `add(a, b)`, która dodaje dwie liczby.

Krok 1: Red - Piszemy testy

```
# prod
def add(a, b):
    pass

# tests
assert add(2, 3) == 5
assert add(5, 0) == 5
assert add(-1, -4) == -5
```

Krok 1: Red - Uruchamiamy testy

> Testy powinny zakończyć się niepowodzeniem.

```
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    assert add(2, 3) == 5
AssertionError


** Process exited - Return Code: 1 **
Press Enter to exit terminal
```

Krok 2: Green - Implementujemy kod

> Teraz piszemy minimalną ilość kodu, aby wszystkie testy zaczęły przechodzić.

```
# prod
def add(a, b):
    return a + b

# tests
assert add(2, 3) == 5
assert add(5, 0) == 5
assert add(-1, -4) == -5
```

Krok 2: Green - Uruchamiamy testy ponownie

> Po zmianie kodu i ponownym uruchomieniu testów, powinniśmy zobaczyć, że wszystkie testy zakończyły się sukcesem.

```
** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Krok 3: Refactor - Analiza kodu

W tym prostym przykładzie kod jest już dość czysty i nie wymaga znaczącej refaktoryzacji. W bardziej złożonych scenariuszach moglibyśmy poprawiać nazwy zmiennych, wydzielać fragmenty kodu do osobnych funkcji, itp.

Ważne: Podczas refaktoryzacji uruchamiamy testy po każdej zmianie, aby upewnić się, że niczego nie zepsuliśmy.

TDD (Bardziej złożony przypadek)

Zaimplementuj funkcję `calculate_discounted_price(price, discount_code)`, która oblicza cenę po uwzględnieniu rabatu.

- Kod **PROMO10** daje 10% rabatu.
- Kod **MEGA20** daje 20% rabatu.
- Jeśli kod jest nieznany lub pusty, rabat nie jest naliczany.

Krok 1: Red - Piszemy testy

```python
# prod
def calculate_discounted_price(price, discount_code):
    pass

# tests

# no discount
assert calculate_discounted_price(100, "") == 100
assert calculate_discounted_price(50, None) == 50
assert calculate_discounted_price(200, "UNKNOWN") == 200

# 10% discount
assert calculate_discounted_price(100, "PROMO10") == 90
assert calculate_discounted_price(50, "PROMO10") == 45

# 20% discount
assert calculate_discounted_price(100, "MEGA20") == 80
assert calculate_discounted_price(25, "MEGA20") == 20
```

Krok 1: Red - Uruchamiamy testy

> Testy powinny zakończyć się niepowodzeniem.

```
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    assert calculate_discounted_price(100, "") == 100
AssertionError


** Process exited - Return Code: 1 **
Press Enter to exit terminal
```

Krok 2: Green - Implementujemy kod

> Teraz piszemy minimalną ilość kodu, aby wszystkie testy zaczęły przechodzić.

```
# prod
def calculate_discounted_price(price, discount_code):
    if discount_code == "PROMO10":
        return price * 0.9
    elif discount_code == "MEGA20":
        return price * 0.8
    else:
        return price
```

Krok 2: Green - Uruchamiamy testy ponownie

> Po zmianie kodu i ponownym uruchomieniu testów, powinniśmy zobaczyć, że wszystkie testy zakończyły się sukcesem.

```
** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Krok 3: Refactor - Analiza kodu

> Jeśli chcielibyśmy dodać więcej kodów rabatowych?

Nasz kod działa, ale nie jest zbyt elastyczny.

```
# prod
def calculate_discounted_price(price, discount_code):
    if discount_code == "PROMO10":
        return price * 0.9
    elif discount_code == "MEGA20":
        return price * 0.8
    else:
        return price
```

Krok 3: Refactor - Refaktoryzacja kodu

```
def calculate_discounted_price(price, discount_code):
    discounts = {
        "PROMO10": 0.10,
        "MEGA20": 0.20
    }

    if discount_code in discounts:
        return price * (1 - discounts[discount_code])
    else:
        return price
```

Krok 3: Refactor - Ponownie uruchamiamy testy

Po refaktoryzacji ponownie uruchamiamy wszystkie testy, aby upewnić się, że nasza zmiana nie wprowadziła żadnych błędów. Wszystkie testy powinny nadal przechodzić.

TDD (Podsumowanie)

TDD to potężna technika, która może znacząco poprawić jakość i strukturę Twojego kodu. Poprzez iteracyjny cykl **Red-Green-Refactor**, zyskujesz lepsze zrozumienie wymagań, tworzysz bardziej testowalny kod i masz większą pewność co do jego poprawności.

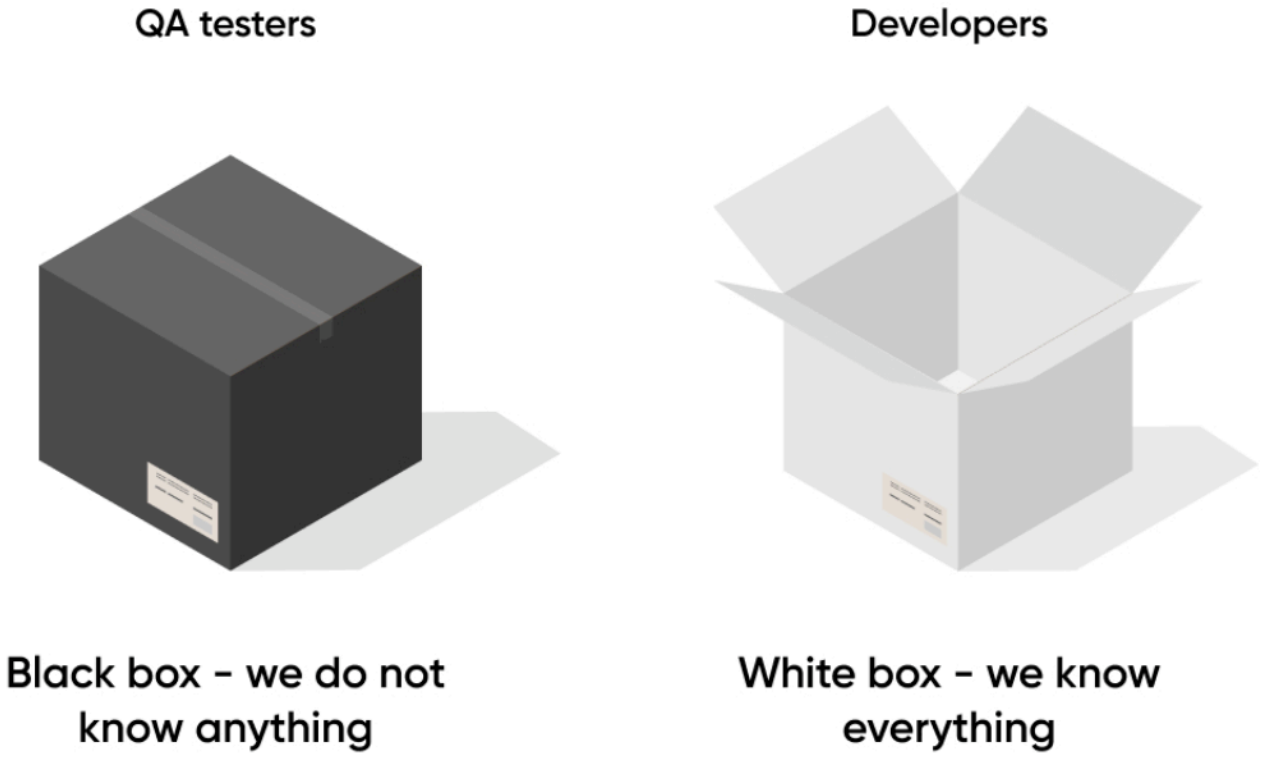
Testy E2E

Testy E2E (end to end)

Testowanie aplikacji jako nierozdzielnej całości z punktu widzenia użytkownika końcowego.

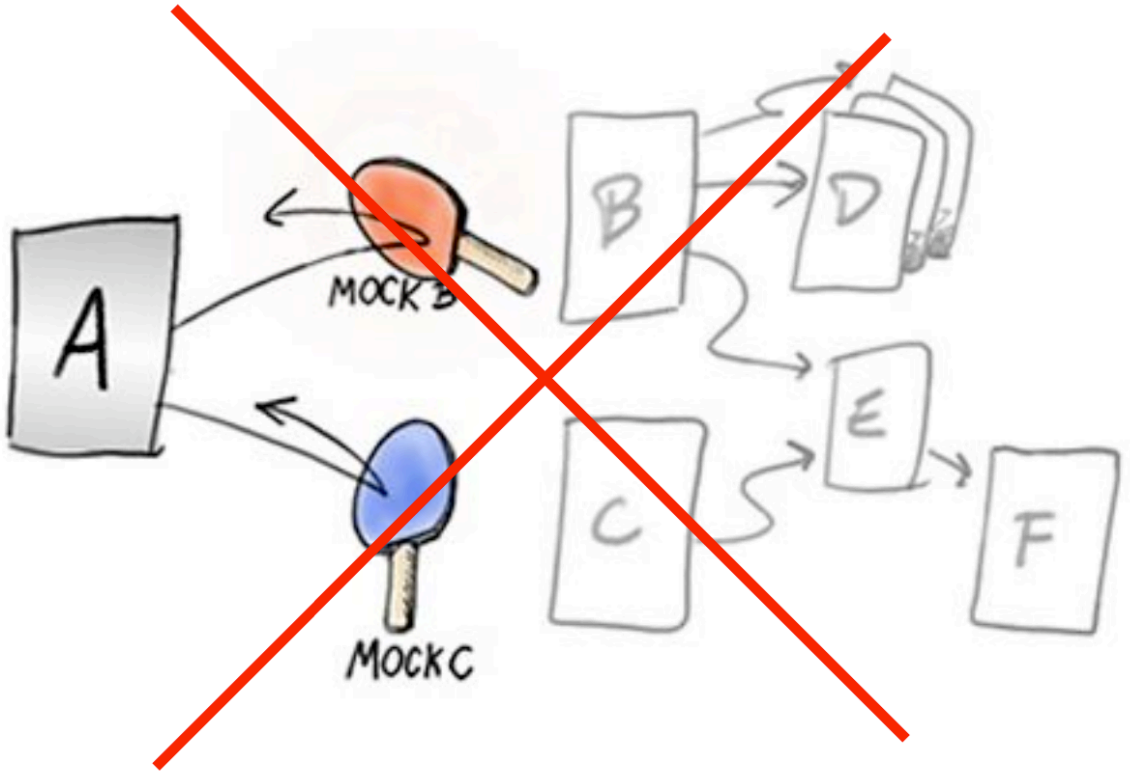
Testy E2E

> Testy czarnej skrzynki (na poziomie interfejsu użytkownika końcowego, bez znajomości wnętrza aplikacji).



Testy E2E

> Bez mockowania fragmentów kodu.



Testy E2E

> Wykonują się wolno.

Testy E2E

> Testowania wszystkich warstw aplikacji na raz powoduje, że zwracana jest mało precyzyjna informacja zwrotną.

Testy E2E

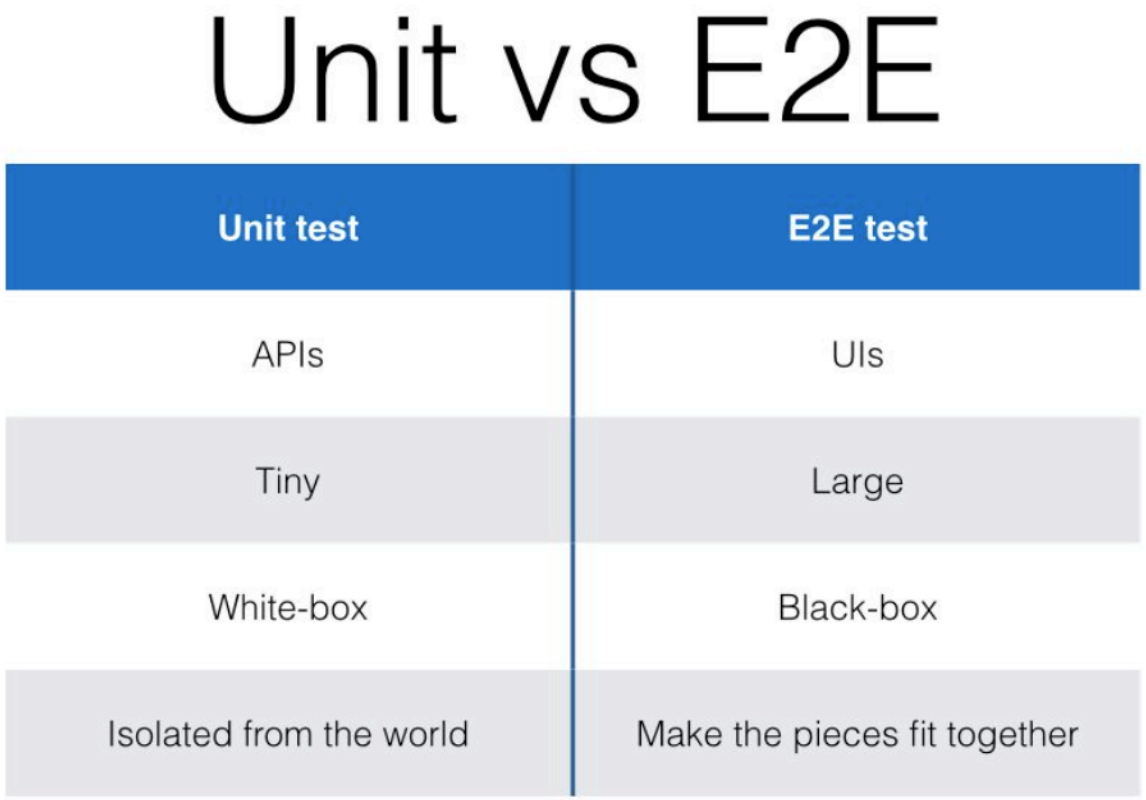
> Są trudne do debugowania (wskazują jedynie, która wysokopoziomowa funkcjonalność nie działa zgodnie z oczekiwaniami, ale nie mówią nam, który fragment kodu jest niepoprawny).

Six Stages of Debugging

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?

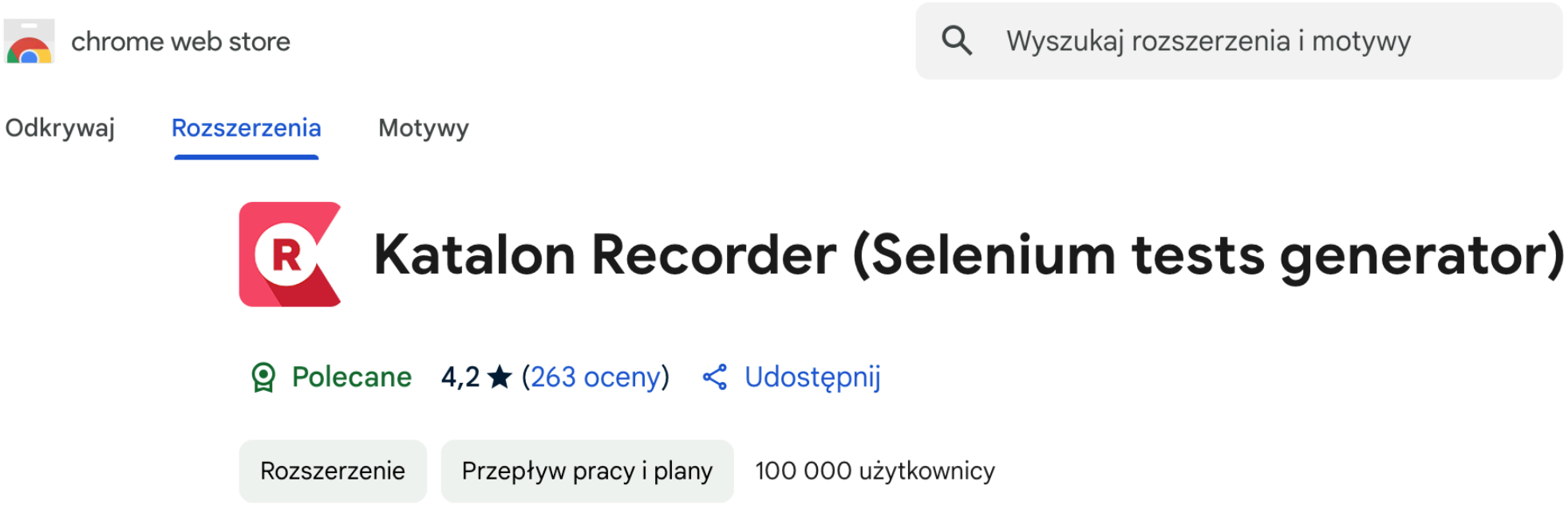
Unit vs E2E

> Testy E2E nie zastępują testów jednostkowych!



Katalon Recorder

> **Katalon Recorder** to narzędzie służące do przeprowadzania automatycznych testów aplikacji webowych.

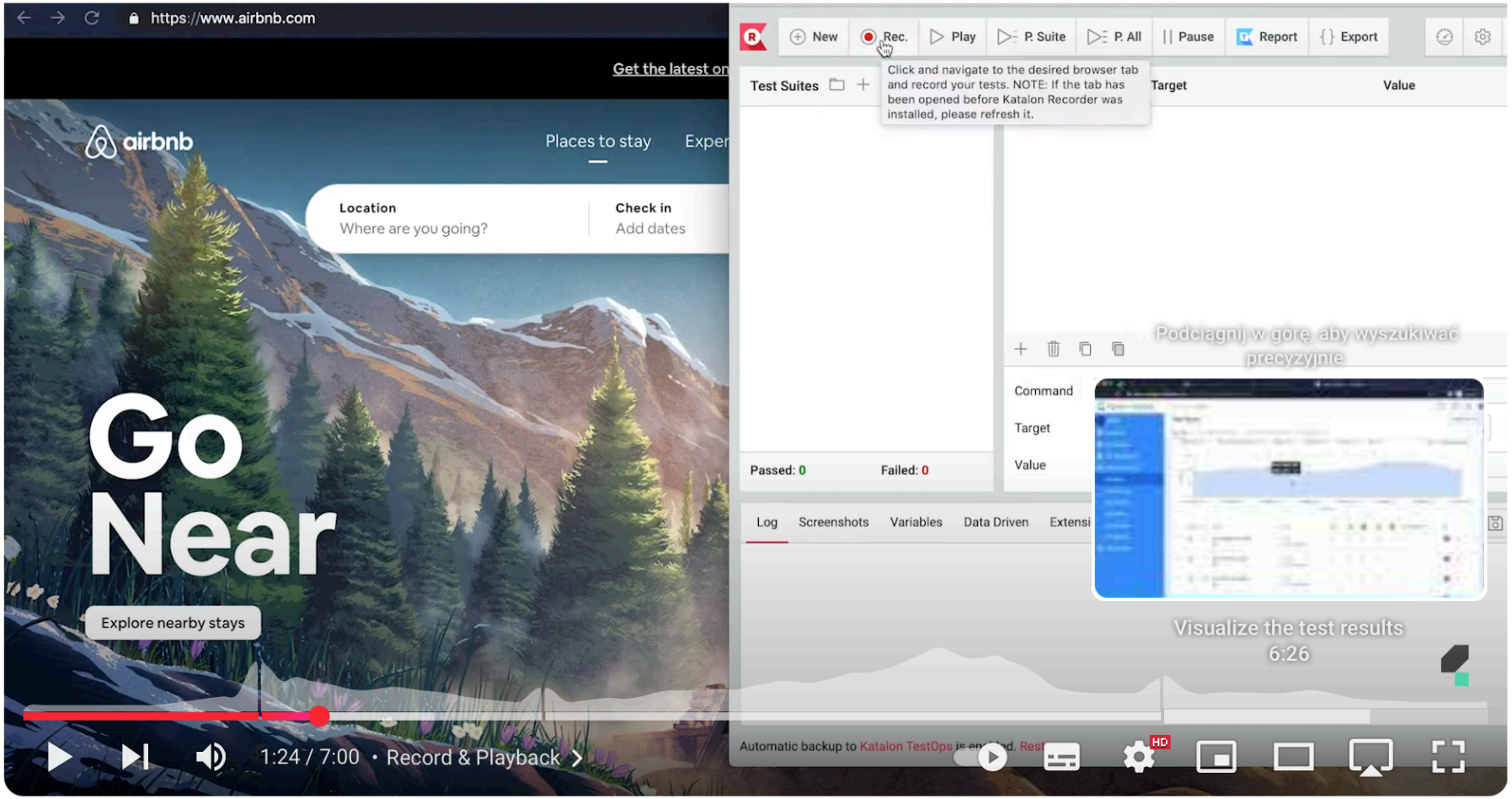


Czym jest Katalon Recorder?

- **Bezpłatne rozszerzenie przeglądarki:** dostępne dla Chrome, Firefox i Edge.
- **Generator testów Selenium:** umożliwia tworzenie skryptów testowych bez pisania kodu.
- **Opiera się na nagrywaniu interakcji:** rejestruje Twoje akcje na stronie internetowej.

Jak działa Katalon Recorder?

https://www.youtube.com/watch?v=sgzFkQ-0Ta8



Do czego służy Katalon Recorder?

- Idealne do szybkiego wygenerowania szkieletu testu.
- Umożliwia tworzenie automatyzacji bez zaawansowanej wiedzy programistycznej.
- Ułatwia szybkie tworzenie prototypów testów i badanie zachowania aplikacji.
- Umożliwia szybką weryfikację funkcjonalności poprzez nagranie i ponowne uruchomienie akcji.

Jak poprawnie zgłaszać znaleziony błąd?

> Przykład: nie działa menu prostej aplikacji webowej.

Jak poprawnie zgłaszać znaleziony błąd?

http://localhost:9000/mems

Najpopularniejsze

Ostatnio dodane

Poczekalnia

← Poprzedni | Jeszcze jeden →

Jak zgłaszać błąd?

Tytuł

> Dobry tytuł jest zwarty i w kilku słowach oddaje istotę problemu.
>
> Należy unikać zbyt ogólnych sformułowań, takich jak **Menu nie działa**, zamiast tego: **Elementy menu stały się nieklikalne**.

Jak zgłaszać błąd?

Opis - miejsce wystąpienia

> W tym miejscu należy podać ciąg dostępu do miejsca, w którym pojawił się błąd, tutaj to adres URL.

http://localhost:9000/mems

Jak zgłaszać błąd?

Opis - środowisko

> Wszelkie informacje dotyczące wersji wykorzystywanego oprogramowania, pakietów, systemu operacyjnego.
>
> W przykładzie niezbędne jest podanie informacji o wersji przeglądarki oraz systemu operacyjnego.

Jak zgłaszać błąd?

Opis - ścieżka odtworzenia

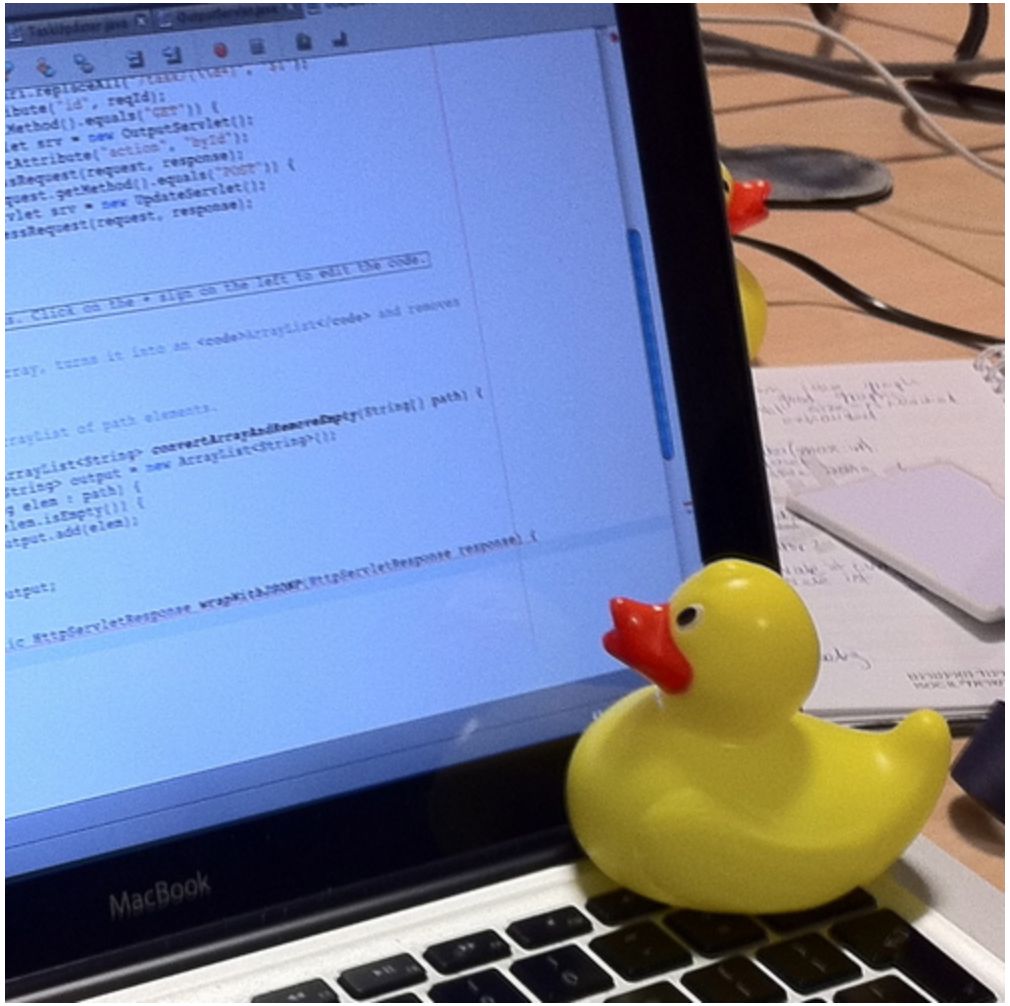
> Ścieżka odtworzenia błędu nieraz jest niezbędna do jego naprawy. Developer może pominąć zgłoszenie, jeśli nie będzie wiedział, jak uzyskać opisywany błąd.

Jak zgłaszać błąd?

Opis - informacje dodatkowe

> Wszelkie inne informacje, dotyczące błędu:
>
> - przypuszczenia co do powodu występowania problemu.
> - informacja czy błąd jest powtarzalny lub jednorazowy.
> - podanie swoich oczekiwań co do spodziewanego efektu naprawy.

Jak efektywnie debugować kod?



Metoda gumowej kaczuszki

> Nieformalny sposób debugowania kodu.
>
> Linia po linii, programista tłumaczy kaczuszce lub innemu obiektowi, co każdy segment kodu powinien robić.

Koniec