

Program szkolenia:

Testowanie automatyczne dla programistów (TDD, BDD, wzorce, narzędzia)

Informacje ogólne

Nazwa:	Testowanie automatyczne dla programistów (TDD, BDD, wzorce, narzędzia)
Kod:	Testowanie-auto
Kategoria:	Testowanie
Grupa docelowa:	testerzy, programiści, projektanci, build managerowie
Czas trwania:	3 dni
Forma:	50% wykłady / 50% warsztaty

Podczas szkolenia uczestnicy pozają techniki programowania i testowania ułatwiające codzienną pracę z kodem.

Podczas warsztatów praktycznych uczestnicy posiadają umiejętność pisania i utrzymywania testów oraz pracy techniką Test-Driven Development.

Wsparcie trenera wspomagającego

Podczas tego szkolenia stawiamy szczególny nacisk na ćwiczenia praktyczne rozwiązywane podczas warsztatów.

Dlatego szkolenie jest prowadzone przez **dwóch trenerów**: głównego i wspomagającego, który asystuje podczas warsztatów. Dzięki temu każdy uczestnik ma nieograniczony dostęp do pomocy i wiedzy eksperckiej.

Zalety szkolenia:

- » Narzędzia automatyzacji
- » Najlepsze wzorce i praktyki
- » Aspekty Behavior Driven Development

Program szkolenia:

1. Podstawy testowania

- 1.1. Sposoby testowania systemu
- 1.2. Rodzaje testów i przykłady ich wykorzystania
- 1.3. Automatyzacja procesu testowania
- 1.4. Wybór strategii testowania w projekcie
- 1.5. Strategia budowania piramidy testów
- 1.6. Piramida testów - jak ją interpretować

2. Najlepsze techniki testowania - projektowanie testów, które można utrzymać w przyszłości

- 2.1. Wprowadzenie
 - 2.1.1. Czego nie testować
 - 2.1.2. Struktury przypadków testowych
- 2.2. Organizacja kodu testowego
 - 2.2.1. Klasa testowa per klasa produkcyjna
 - 2.2.2. Klasa testowa per funkcjonalność
 - 2.2.3. Klasa testowa per setup
 - 2.2.4. Testy parametryzowane
- 2.3. Przygotowanie stanu (test fixture setup)
 - 2.3.1. Testy wykorzystujące źródło danych (data-driven testing)
 - 2.3.2. Użycie wzorca budowniczego i dekoratora do przygotowania stanu
 - 2.3.3. Wzorce i szablony
- 2.4. Weryfikacja
 - 2.4.1. Value Object, weryfikacja przez equals
 - 2.4.2. Własne asercje

2.4.3. Weryfikacja przy użyciu specyfikacji (Matcher object)

2.4.4. Upraszczenie asercji z użyciem Assert Object

2.4.5. Poprawna weryfikacja przypadków negatywnych

2.4.6. Wzorce i szablony

2.5. Uprzątniecie po teście (fixture teardown)

2.5.1. Kiedy warto stosować

2.5.2. Manualne

2.5.3. Automatyczne

2.5.4. Wzorce i szablony

2.6. Antywzorce testowania (ponad 20 typowych błędów i pułapek)

2.7. Wykrywanie "Zapachów" złego kodu testowego

2.7.1. Delikatne testy (fragile)

2.7.2. Nieczytelne testy

2.7.3. Wolne testy

2.7.4. Testy niedeterministyczne

3. Testowanie jednostkowe

3.1. Szablony testów w xUnit

3.2. Tworzenie własnych asercji

3.3. Techniki: Mock, Stub, Fake

3.3.1. Dobór technik do potrzeb - czym się kierować

3.3.2. Przykłady implementacji w Mockito

3.4. Mockowanie

3.4.1. Zalety testowania w izolacji

3.4.2. Nagrywanie zachowania

3.4.3. Weryfikacja wywołań

3.4.4. Antywzorce testów wykorzystujące mockowanie

3.5. Co sprawia że czas poświęcony na napisanie testu zwróci się

3.6. Testability - podatność kodu na testy

3.6.1. Jak pisać kod, który daje się testować

3.6.2. Najlepsze praktyki: SOLID, GRASP

3.6.3. Wybrane wzorce projektowe, które zwiększają testability: Factory, Strategy, Value Object

3.6.4. Pułapki i typowe błędy

3.6.5. Code smell - "zapachy" nietestowalnego kodu

4. Test Driven Development

4.1. Cykl czerwony-zielony-refaktoring

4.2. Wady i zalety TDD

4.3. Kiedy warto, dla kogo jest TDD

4.4. Ewolucyjny sposób rozwój kodu

4.5. Podstawowe techniki refactoringu

5. Testowanie integracyjne

5.1. Testowanie architektury opartej na zdarzeniach (event-driven-architecture)

6. Testowanie akceptacyjne

6.1. Technika User Story

6.2. Tworzenie testów akceptacyjnych na podstawie User Story

6.3. Zyski i koszty różnych technik testowania akceptacyjnego

6.4. Efektywne narzędzia wspierające:

6.4.1. Testowanie poprzez warstwę GUI

6.4.2. Testowanie poprzez warstwę serwisów

6.4.3. Przygotowywanie stanu początkowego

6.5. Testy akceptacyjne od warstwy GUI (end-to-end)

6.6. Testy akceptacyjne uproszczone

7. Behaviour Driven Development

7.1. Zalety bliskiej współpracy z klientem

7.2. Tworzenie aplikacji podejściem BDD

7.3. Ewolucyjne podejście do rozwoju systemu - odraczanie decyzji (BDUF), a ewolucja sterowana

7.4. Narzędzia i wzorce

7.4.1. JBehave - najlepsze praktyki

7.4.2. Page Object - modelowanie użytkownika

7.4.3. Technika ujednolicania testów wykonywanych poprzez GUI i Servisy

7.5. Wykorzystanie narzędzi do testowania akceptacyjnego i integracyjnego

8. Techniki organizacji kodu testowego - standardy, wzorce i najlepsze praktyki

8.1. Wsparcie refaktoryzacji ze strony IDE

8.2. Techniki refaktoryzacji kodu testowego

8.3. Efektywne sposoby utrzymania dużej liczby testów

9. Przegląd kodu

9.1. Zbieranie i interpretowanie metryk

9.2. Wykrywanie punktów krytycznych systemu

9.3. Identyfikowanie zapachów kodu

9.4. Wprowadzenie do pair programming

9.4.1. Podstawy psychologiczne

9.4.2. Model Dreyfus

9.4.3. Wykorzystanie potencjału obu półkul mózgowych

10. Techniki specyficzne dla frameworków / architektur (do wyboru)

10.1. Java

10.1.1. Spring framework

10.1.2. JBoss Seam

10.1.3. Hibernate

10.2. .NET

10.2.1. Wykorzystanie ServiceLocator i Dependency Injection

10.3. Architektura n-warstwowa

11. Kompleksowy proces

11.1. Strategiczny design - Domain Driven Design

11.2. Korzystanie z user story, BDD i TDD w codziennej pracy

11.3. Automatyzacja - korzystanie z serwera Continuous Integration

11.4. Narzędzia (do wyboru)

11.4.1. Java: JUnit/TestNG, JBehave (pluginy Eclipse), Selenium

11.4.2. .NET: NUnit, NDbUnit, nBehave, nSubstitute, nCover, Selenium

11.5. Wykorzystanie najlepszych praktyk, wskazanie typowych błędów i pułapek

11.6. Przykładowy projekt