Zasady pisania testów automatycznych

whoami

Mateusz Sosnowski

Systems designer @ Logintegra Sp. z o. o.

Wprowadzenie do automatyzacji testów

|Po co automatyzować?

Automatyzujemy powtarzające się zadania.

Smoke testy

Testy wydajnościowe

Testy regresyjne

Procesy konfiguracyjne

|Cele automatyzacji?

Powtarzalność i spójność testów

Wykonywanie testów bez nadzoru

Większe pokrycie testami

Zmniejszenie kosztów testowania

Znajdowanie błędów regresji

Procesy konfiguracyjne

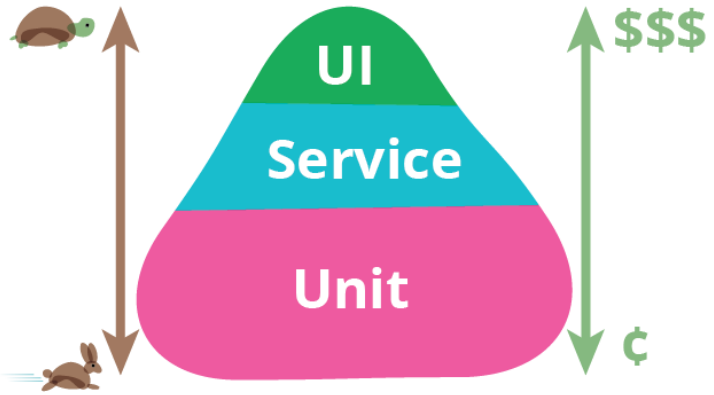
|Wybór testów do automatyzacji

- Najważniejsze testy
- Powtarzające się błędy w aplikacji
- Testy najważniejszych funkcji
- Testy, które najłatwiej zautomatyzować
- Testy, które najszybciej się zwrócą
- Testy wykonywane najczęściej

|Ograniczenia automatyzacji

- Nie zastępuje testów manualnych (testy eksploracyjne, zgadywanie błędów)
- Narzędzia nie mają wyobraźni
- Automatyzacja to tylko mechanism wykonywania testów

Piradmida testów



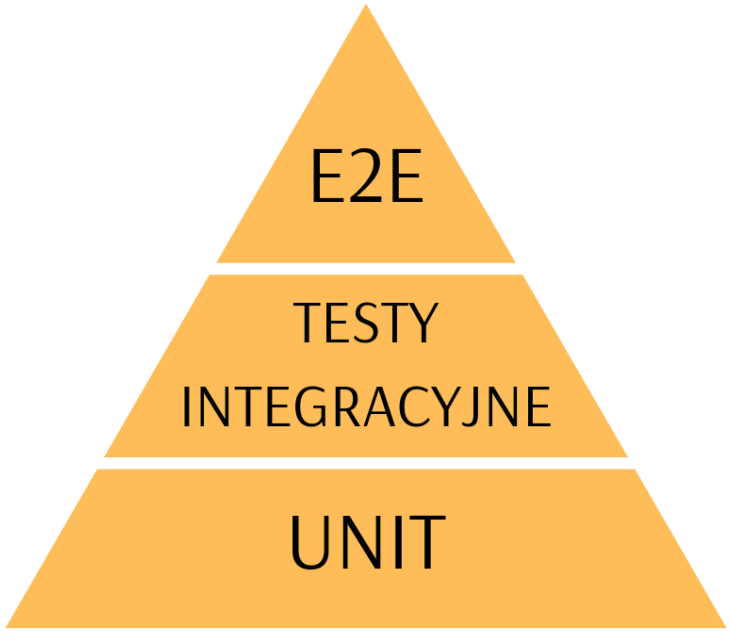
|Piramida testów

Podział testów automatycznych:

- **Testy systemowe** - testy funkcjonalne i niefunkcjonalne (testy bezpieczeństwa, obciążeniowe, wydajnościowe, itd.),

- **Testy integracyjne** - współdziałanie rozłącznych fragmentów systemu,

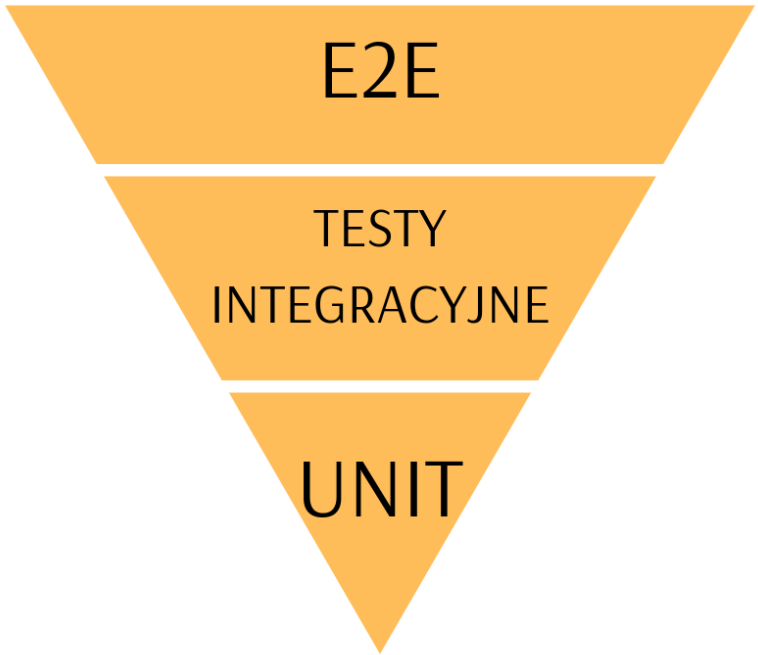
- **Testy jednostkowe** – testy wyizolowanych fragmentów na najniższym poziomie (np. sprawdzenie warunków brzegowych, przypadków pozytywnych i negatywnych, walidacji danych wejściowych).



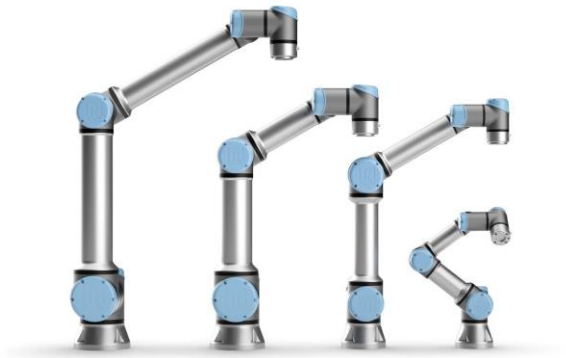
|Odwrócona piramida testów

Powstały opinie twórców portali społecznościowych, według których testy jednostkowe nie są potrzebne.

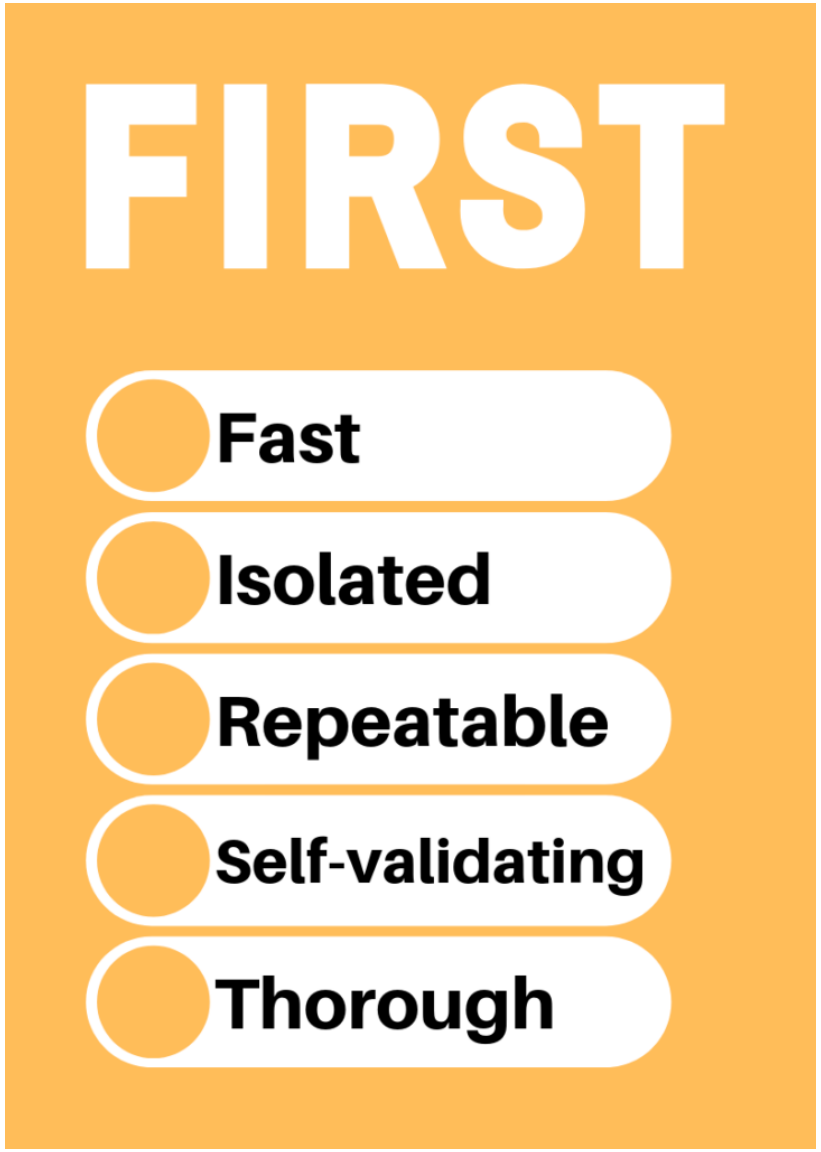
W przypadku aplikacji opartych głównie na interfejsie, w których logika biznesowa oraz złożone algorytmy nie są istotnymi komponentami, można mówić o odwróconej piramidzie, która określa liczbę testów danego rodzaju. W aplikacjach tego typu najważniejsze są testy “końcowego użytkownika”.



Podstawowe zasady testów automatycznych



Zasada first



Charakter testów najlepiej definiuje zasada FIRST.

Zgodnie z tymi wytycznymi testy powinny być:

- szybkie (Fast),
- niezależne (Isolated),
- powtarzalne (Repeatable),
- samosprawdzające (Self-validating).
- kompletne (Thorough).

F (fast) - szybkie

Nikt nie lubi długo czekać na załadowanie się strony, uruchomienie animacji czy zbudowanie programu.

Dla programisty nie ma nic gorszego niż wolno działające narzędzia. Podobnie jest z testami.

Czas wykonywania testów nie powinien zniechęcać do ich uruchamiania.

Podczas pisania testów należy pamiętać o tym, że długi czas ich wykonywania zwiększa czas budowania całej paczki.

Dlatego wprowadzenie wolnych testów, o długim czasie wykonywania skutkuje zmniejszeniem częstotliwości ich uruchamiania. Z czasem konsekwencją tego może być nawet całkowite zaniechanie uruchamiania testów.

I (isolated) - Niezależne

Każdy test powinien sprawdzać tylko jeden, konkretny przypadek.

Oznacza to, że każda metoda lub nawet poszczególne jej części powinny być testowane oddzielną asercją, zawierającą specyficzne dane.

Jeden test nie może być powiązany z innymi ani nie powinien na nie wpływać. Jest to istotne zwłaszcza w kontekście utrzymania i rozwoju kodu – dzięki takiemu rozplanowaniu testów można swobodnie rozbudowywać poszczególne moduły lub dodawać nowe.

Niezależność testów sprawia, że nie mają one wpływu na środowisko testowe, a kolejność ich wykonywania nie powinna mieć znaczenia dla działania całego programu.

|R (repeatable) - powtarzalne

Tylko powtarzalność testów sprawia, że są one miarodajne i świadczą o prawidłowym działaniu programu.

Jest to jeden z głównych aspektów, który daje przewagę nad testami manualnymi.

Testy powinny zwracać ten sam wynik niezależnie od środowiska, w którym są uruchamiane, klasy sprzętu maszyny, na której są wykonywane, specyficznych ustawień środowiska czy czasu ich wykonywania.

Problemy z powtarzalnością wykonania testów mogą wskazywać na problemy w samym kodzie produkcyjnym.

Dane testowe powinny być z góry określone, z wyłączeniem przypadków, gdy nie chcemy sprawdzać zachowań programu dla określonych danych, a jedynie dla pewnego zakresu, np. zdefiniowanego poprzez typ (np. Intetger).

|S (self-validating) samosprawdzające

Głównym założeniem testowania programu jest ułatwienie lokalizacji błędów, co przyspiesza proces ich naprawy.

Właśnie dlatego wynik testu powinien być widoczny na pierwszy rzut oka, bez konieczności wnikliwego sprawdzania przypadku i testowanego obszaru.

Aby zachować czytelność i przejrzystość w przeprowadzaniu testów, informacje o sprawdzanym obszarze powinny być zawarte w nazwie metody testujące, a sam test powinien sprawdzać jedno założenie.

W ten sposób programista może szybko zorientować się czego dotyczy dany test i jaki jest jego wynik.

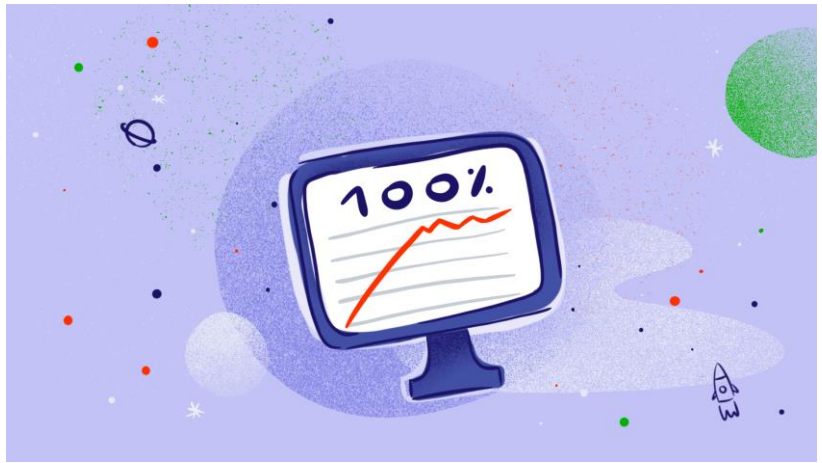
|T (thorough) – kompletne

Testy powinny być niezależne, co nie oznacza jednak, że nie powinny sprawdzać znacznego obszaru kodu.

Duże pokrycie testami przekłada się na zwiększone zaufanie do aplikacji.

Aby podnieść jakość programu i jednocześnie potwierdzić jego niezawodność, dobrane testy powinny być jak najbardziej różnorodne.

Istotne są zarówno testy sprawdzające warunki brzegowe, testy rzucania wyjątków, testy z wykorzystaniem dużych zbiorów danych, jak i testy z użyciem danych wejściowych, które teoretycznie nigdy nie powinny zostać wprowadzone przez użytkownika.



Selenium

|Selenium WebDriver

Selenium to zestaw narzędzi zaprojektowanych do automatyzacji przeglądarek. Jest powszechnie używany do testowania aplikacji internetowych na wielu platformach. Pod parasolem Selenium dostępnych jest kilka narzędzi, takich jak Selenium WebDriver (wcześniej Selenium RC), Selenium IDE i Selenium Grid.

WebDriver to *interfejs* zdalnego sterowania, który umożliwia manipulowanie elementami DOM na stronach internetowych. Dostarcza użytkownikowi gotowe API pozawalające na interakcję z przeglądarkami:

- GeckoDriver (Mozilla Firefox)
- ChromeDriver (Google Chrome)
- SafariDriver (Apple Safari)
- InternetExplorerDriver (MS InternetExplorer)
- MicrosoftWebDriver lub EdgeDriver (MS Edge)
- OperaChromiumDriver (przeglądarka Opera)

|Co można zrobić za jego pomocą?

- Sprawdzać obecność elementów
- Znajdować element
- Klikać
- Wpisywać

- Pobierać tekst
- Pobierać atrybuty element
- Wysyłać formularz
- I wiele innych

|Niezbędne narzędzia - SeleniumWebdriver

- JDK (Java Development Kit)
- IDE (np. IntelliJ)
- Selenium WebDriver
- Chrome Driver (lub inny)

|Konfiguracja projektu Maven z Selenium i udostępnienie na GitHub

1. Stworzenie projektu **Maven** w IntelliJ
2. Stwórz repozytorium w foderze projektu: **git init**
3. Dodaj plik **.gitignore** i wpisz **/.idea**
4. Dodaj pliki do repozytorium: **git add .gitignore, pom.xml**, **git add src/**
5. Stwórz commit: **git commit –m “initial commit”**
6. Stwórz puste repozytorium w GitHub
7. Powiąż repozytorium lokalne ze zdalnym: **git remote add origin https://github.com/nazwa_konta_github/nazwa_repo.git**
8. Wypchnij zmiany do GitHub **git push --set-upstream origin master**
9. Wyszukujemy na stronie https://mvnrepository.com/ najnowszej stabilnej wersji **selenium-java** i wklejamy do pliku pom.xml w węźle <dependencies>
10. Podobnie **selenium-chrome-driver**
11. Tworzymy commit: **git commit –m “selenium maven dependency”**
12. Tworzymy katalogi: /src/test/**resources/drivers.** Dodajemy tu plik chromedriver.exe z https://chromedriver.storage.googleapis.com/index.html?path=87.0.4280.88/

|Przykładowa klasa SeleniumTest – do przetestowania konfiguracji SeleniumWebriver

```java
package pl.gda.wsb;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import java.util.concurrent.TimeUnit;

public class SeleniumTest {

    public static void main(String[] args) {

        // Declaration and instantiation of driver
        System.setProperty("webdriver.chrome.driver", "src/test/resources/drivers/chromedriver.exe");
        WebDriver driver = new ChromeDriver();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        driver.manage().window().maximize();

        // Launch Logintegra website
        driver.navigate().to("https://logintegra.com/");

        // Click to Knowledge button
        driver.findElement(By.xpath("//*[@id=\"menu-item-2168\"]/a")).click();

        // Click to Article
        driver.findElement(By.xpath("/html/body/div[1]/div/div[1]/div[2]/div[1]/h3/a")).click();

        // Get page title and print in console
        System.out.println(driver.getTitle());

        // Quit
        driver.quit();
    }
}
```

(src/test/java/pl/gda/wsb/SeleniumTest.java)

Cucumber

BDD - Cucumber

Behavior Driven Development to zbiór praktyk, dzięki którym opis funkcjonalności będzie w pełni zrozumiały dla biznesu i zapewni poziom szczegółowość potrzeby programistom do implementacji.

W BDD olbrzymią rolę odgrywa współpraca — zarówno między użytkownikami a zespołem projektowym, jak i wewnątrz samego zespołu.

Analitycy biznesowi, programiści i testerzy podczas definiowania i specyfikowania cech funkcjonalnych współpracują z użytkownikami końcowymi, a członkowie zespołu czerpią pomysły ze swoich indywidualnych doświadczeń oraz know-how.

Specyfiką BDD jest język naturalny, którego struktura wywodzi się z określonego z góry modelu. Zawiera wszystkie pojęcia, które będą użyte do definicji zachowania systemu. Umożliwia on płynną komunikację między klientami, a deweloperami.

BDD nie dopuszcza dowolności w formatowaniu opisu cech/funkcjonalności, historii użytkowników oraz scenariuszy. Istnieją określone szablony przygotowane w oparciu o wcześniej stworzony język Gherkin.

Gherkin

Według BDD opis nowej funkcjonalności powinien rozpoczynać się od zdefiniowania historyjki użytkownika, która opisuje:

- cel biznesowy,
- rolę,
- to co dana funkcja ma wykonywać.

Ważne, aby wszyscy członkowie zespołu rozumieli co dany projekt ma realizować. Cele biznesowe muszą być przekazane zespołowi deweloperskiemu, aby znał dokładny kontekst tego co jest do zaimplementowania.

Opis każdej nowej funkcjonalności powinien rozpoczynać się od stwierdzeń:

W celu <osiągnięcia celu biznesowego lub dostarczenia wartości biznesowej>
Jako <interesariusz>
Chcę <czegoś>

Gherkin

Według BDD opis nowej funkcjonalności powinien rozpoczynać się od zdefiniowania historyjki użytkownika, która opisuje cel biznesowy, rolę i to co dana funkcja ma wykonywać.

W celu <osiągnięcia celu biznesowego lub dostarczenia wartości biznesowej>
Jako <interesariusz>
Chcę <czegoś>

W BDD mamy do czynienia z łatwą strukturą opierającą się na trzech krokach:

Krok 1. Given: Warunek początkowy

Krok 2. When: Akcja jaką wykonuje użytkownik

Krok 3. Then: Oczekiwany wynik

W taki sposób można tworzyć testy funkcjonalne poprzez historyjki użytkowników.

Gherkin

Przykładowa historyjka w języku Gherkin wygląda następująco:

Feature : Check login functionality

Scenario:
Given user is on login screen
When user enter valid username and password
And clicks on login button
Then user is navigated to home page

|Konfiguracja Cucumber

1. Do projektu Maven ze skonfigurowanym Selenium Webdriver należy dodać zależności do pliku **pom.xml** (gherkin, hamcrest-core, cucumber-junit, cucumber-java, cucumber-html). Szukamy ich na stronie https://mvnrepository.com/
2. Dodajemy w IntelliJ plugin **Cucumber for Java** (File → Settings → Plugins)
3. Tworzymy folder **Features** w src/test/resources
4. Wpisujemy scenariusz do pliku, np.

```
Feature: Test OrangeHrm Login Page

  Scenario: Check login with wrong credentials
    Given Open login page
    When User enters username and password
    And Clics on login button
    Then The validation message is displayed
```

|Konfiguracja Cucumber

5. Uruchom feature **loginPage.feature.** W konsoli powinny pojawić się niezaimplementowane stepy:

You can implement missing steps with the snippets below:

```
@Given("Open login page")
public void open_login_page() {
   // Write code here that turns the phrase above into concrete actions
   throw new io.cucumber.java.PendingException();
}

@When("User enters username and password")
public void user_enters_username_and_password() {
   // Write code here that turns the phrase above into concrete actions
   throw new io.cucumber.java.PendingException();
}

@When("Clics on login button")
public void clics_on_login_button() {
   // Write code here that turns the phrase above into concrete actions
   throw new io.cucumber.java.PendingException();
}

@Then("The validation message is displayed")
public void the_validation_message_is_displayed() {
   // Write code here that turns the phrase above into concrete actions
   throw new io.cucumber.java.PendingException();
}
```

|Konfiguracja Cucumber

6. Stwórz nową klasę src/test/java/pl/gda/wsb/StepDefinitions/**LoginSteps.java** i wklej propozycje z konsoli.
7. Przy adnotacjach Given, When, Then pojawi się problem – IntelliJ podpowie, że można dodać **importy** (alt + enter).
8. W każdej nowej metodzie treść ciała metody zastępujemy wypisaniem tekstu do konsoli, np. System.out.println(“Open login page”).
9. Uruchamiamy test z pliku *.feature.
10. W konsoli widać, że testy działają.
11. Dodajemy do Cucumber TestRunner, który będzie odpowiedzialny za uruchamianie tetsów. Tworzymy nową klasę src/test/java/pl/gda/wsb/TestRunner/**TestRunner.java.**
12. Wklejamy kod:

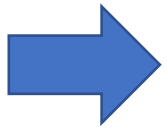
```
package pl.gda.wsb.TestRunner;

    import io.cucumber.junit.Cucumber;
    import io.cucumber.junit.CucumberOptions;
    import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(plugin = {"pretty", "html:target/cucumber"},
    features = "src/test/resources/Features/",
    glue = "pl/gda/wsb/StepDefinitions",
    publish = true)
public class TestRunner {
}
```

|Parametryzacja - Cucumber

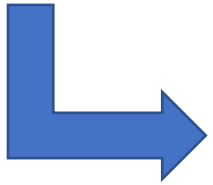
Scenario: Login
1. Open browser
2. Navigate to login page
3. Enter Admin in username box
4. Enter admin1234 in password box
5. …



Scenario Outline: Login
1. Open browser
2. Navigate to login page
3. Enter <username> in username box
4. Enter <password> in password box
5. …

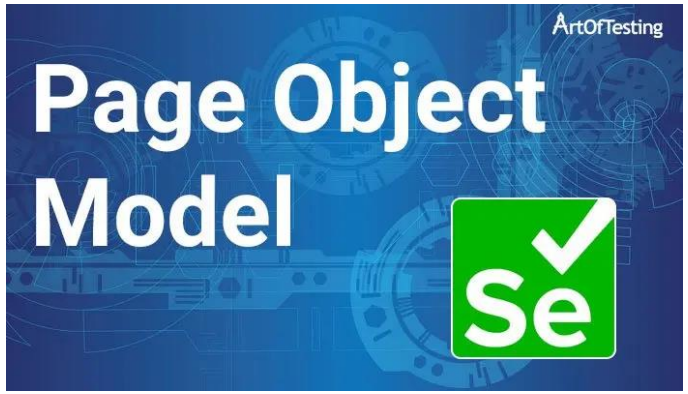
Examples:
username	password
Admin	adin1234
admin	admin

```
@When("User enters incorrect username and password")
  public void user_enters_username_and_password() {
     driver.findElement(By.id("txtUsername")).sendKeys("Admin");
     driver.findElement(By.id("txtPassword")).sendKeys("admin1234");
  }
```

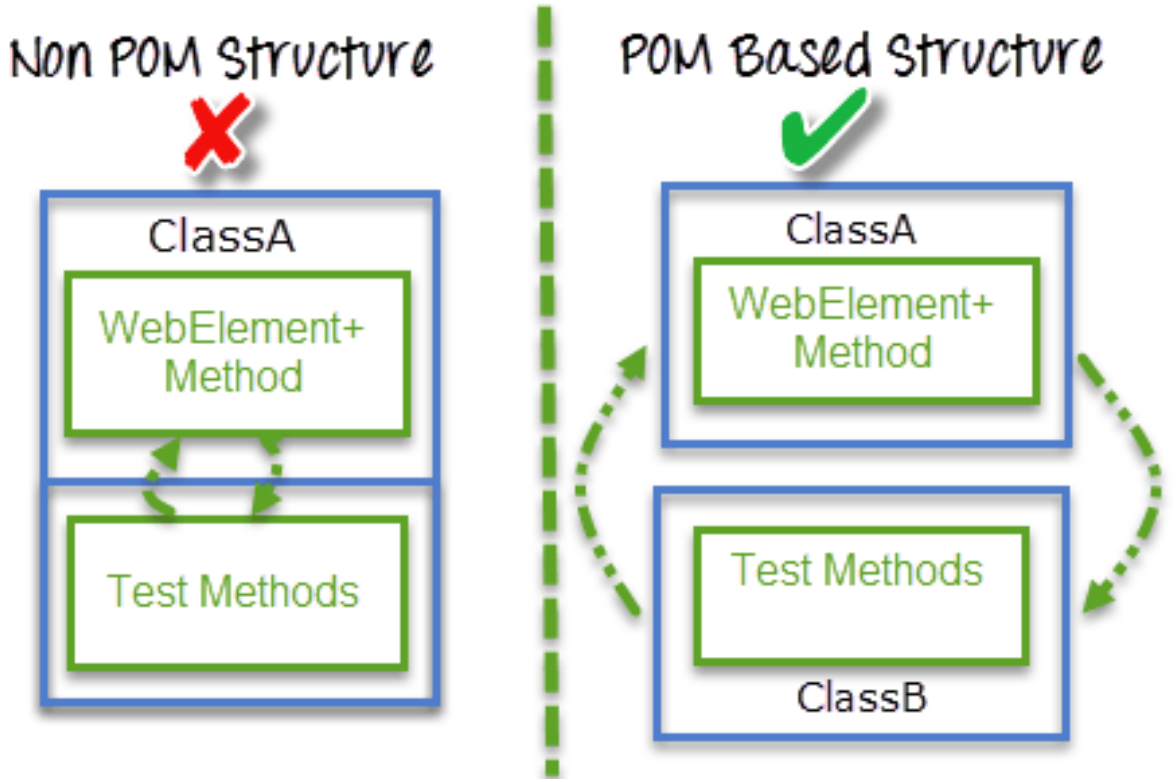


```
@When("^User enters incorrect (.*) and (.*)$")
  public void user_enters_username_and_password(String username, String password) {
     driver.findElement(By.id("txtUsername")).sendKeys(username);
     driver.findElement(By.id("txtPassword")).sendKeys(password);
  }
```

Page Object Model



POM


Non POM Structure
ClassA
WebElement+
Method
Test Methods
POM Based Structure
ClassA
WebElement+
Method
Test Methods
ClassB

POM

Najważniejszą ideą, stojącą u podstaw wzorca **Page Object Model**, jest opakowanie strony i jej elementów w odpowiednie klasy i ich pola.

Należy pamiętać, że wbrew temu, co sugeruje powszechnie przyjęte nazewnictwo, klasą niekoniecznie musi być cała strona – w praktyce częściej zdarza się, że jest nią określona funkcjonalna część strony, na przykład panel czy lista wyników wyszukiwania.

Elementy strony zaczynają odpowiadać polom klasy, a czynności, które można za ich pomocą wykonać wykonać – metodom tej klasy.

Powstaje w ten sposób swojego rodzaju API (interfejs aplikacji), umożliwiający testom wykonywanie czynności na stronie bez odwoływania się do ścieżek do elementów. Upraszcza to prace nad automatyzacją i poprawia czytelność samych metod testowych.

Przykład klasy LoginPage

Przykład: klasa LoginPage

```java
package pl.gda.wsb.Pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class LoginPage {

    protected WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    private By usernameInput = By.id("txtUsername");
    private By passwordInput = By.id("txtPassword");
    private By loginButton = By.id("btnLogin");
    private By validationMessage = By.id("spanMessage");

    public void enerUsername(String username) {
        driver.findElement(usernameInput).sendKeys(username);
    }

    public void enerPassword(String password) {
        driver.findElement(passwordInput).sendKeys(password);
    }

    public void clickLogin() {
        driver.findElement(loginButton).click();
    }

    public String getValidationMessage() {
        return driver.findElement(validationMessage).getText();
    }
}
```

Page Factory

Page Factory

Page Factory rozszerza Page Object Model wprowadzając nowe element.

Page Factory jest bardziej optymalny.

W Page Factory można wskazać na listę elementów.

| Page Factory - Example

```java
package pl.gda.wsb.PageFactory;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class LoginPage_PF {

    @FindBy(id = "txtUsername")
    WebElement txt_username;

    @FindBy(id = "txtPassword")
    WebElement txt_password;

    @FindBy(id = "btnLogin")
    WebElement btn_Login;

    @FindBy(id = "spanMessage")
    WebElement validationMessage;

    WebDriver driver;

    public LoginPage_PF(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, page: this);
    }

    public void enterUsername(String username) {
        txt_username.sendKeys(username);
    }

    public void enterPassword(String password) {
        txt_password.sendKeys(password);
    }

    public void clickOnLogin() {
        btn_Login.click();
    }

    public String getValidationMessage() {
        return validationMessage.getText();
    }

}
```

Tags, hooks, reports