Inżynieria oprogramowania 1 — Automatyzacja testów

Arkadiusz Chrobot

Katedra Systemów Informatycznych, Politechnika Świętokrzyska

Kielce, 29 stycznia 2025

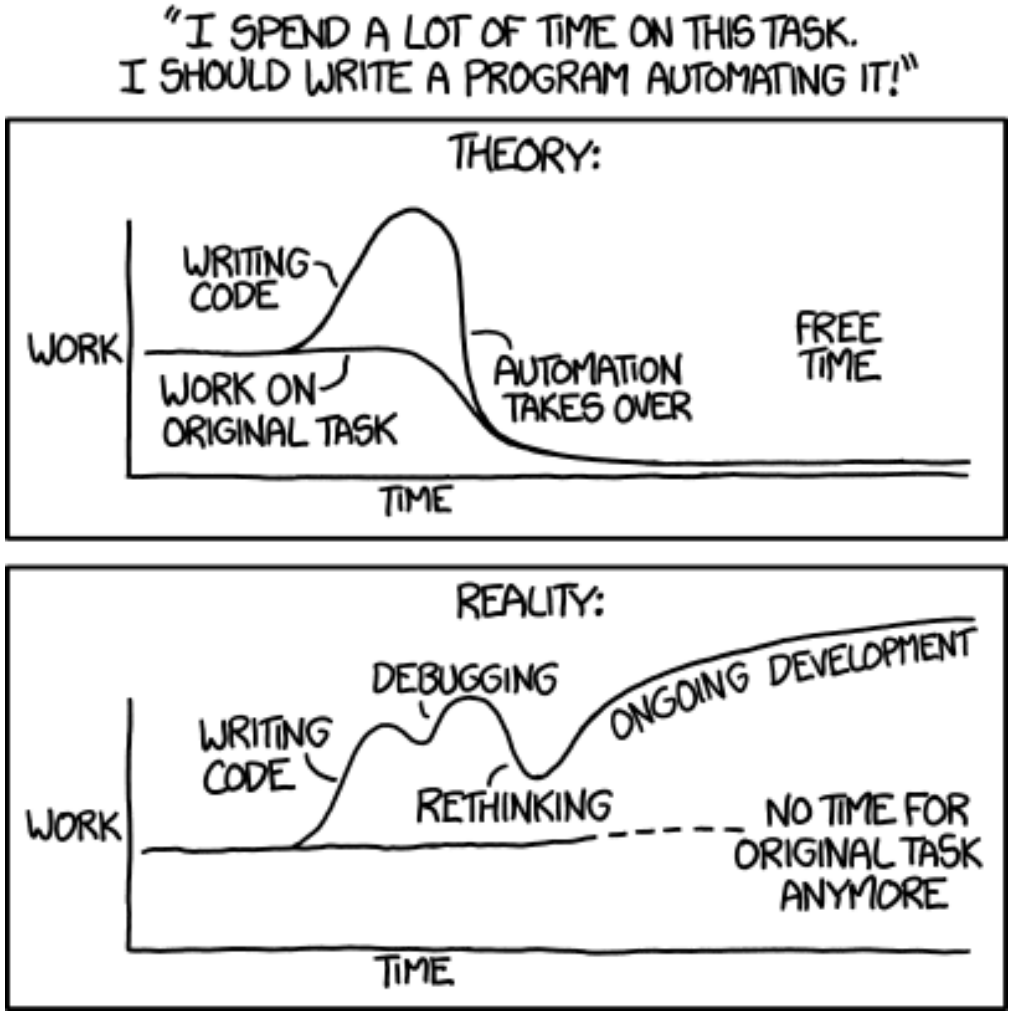
Plan

Motto

„Why program by hand in five days what you can spend five years of your life automating?”

— Terence Parr

Motto



Źródło: xkcd.com

Wstęp

Automatyzacja testów nie tylko zmniejsza koszty *kontroli jakości*, ale również umożliwia inżynierom oprogramowania na monitorowanie postępu ich prac oraz (do pewnego stopnia) bezpieczne wprowadzanie modyfikacji do oprogramowania. Testy automatyczne są również formą *dokumentacji oprogramowania*. Automatyzacja czyni także testy powtarzalnymi. Jednakże, aby uzyskać te korzyści należy przeprowadzić automatyzację testów we właściwy sposób.
W ramach tego wykładu omówione zostaną zasady i narzędzia, które pozwalają inżynierom oprogramowania zautomatyzować testy jednostkowe i systemowe.

Testy jednostkowe

Zasady F.I.R.S.T

Automatyczne testy jednostkowe (i integracyjne) powinny być zgodne z zasadami F.I.R.S.T, czyli powinny być:

Szybkie (ang. *Fast*) Testy są tworzone po to, aby dostarczać informacji zwrotnej inżynierom oprogramowania tak szybko, jak to tylko możliwe. Jeśli są one wolne, to programiści będą unikać uruchamiania ich, co opóźni wykrycie potencjalnych defektów.

Niezależne (ang. *Independent*) Żaden test nie powinien zależeć od wyników poprzednich testów. Kolejność ich uruchamiania powinna być dowolna. W przeciwnym przypadku niepowodzenie jednego z nich może spowodować kaskadowe niepowodzenie kolejnych.

Powtarzalne (ang. *Repeatable*) Testy powinny dać się uruchomić w każdym rozsądnie skonfigurowanym środowisku. Inżynierowie oprogramowania powinni mieć możliwość ich wykonania zarówno na swoich laptopach, jaki i w potoku CI/CD.

Testy jednostkowe

Zasady F.I.R.S.T — kontynuacja

Samokontrolujące się (ang. *Self-Validating*) Wynik testu powinien po prostu informować, czy ten test zakończył się pomyślnie, czy nie. Przyczyna niepowodzenia testu jest ważną informacją, ale nie powinna ona przysłaniać jego podstawowego rezultatu. Inżynier oprogramowania nie powinien być zmuszany do czytania długich raportów w celu uzyskania informacji, czy test się powiódł, czy nie.

O czasie (ang. *Timely*) Testy powinny być utworzone przed kodem produkcyjnym, który mają sprawdzać. W przeciwnym przypadku ten kod może się okazać zbyt trudnym lub nawet niemożliwym do przetestowania.

Testy jednostkowe

Metoda Test-Driven Development

Ostatnia z reguł F.I.R.S.T jest również dobrym podsumowaniem metody *Test-Driven Development* (*TDD*) będącej metodą tworzenia oprogramowania, której głównymi założeniami są:

1. tworzenie testów jednostkowych *przed* kodem produkcyjnym;
2. pisanie kodu produkcyjnego tylko po to, żeby jeden z testów zakończył się sukcesem;
3. nie tworzenie większej liczby testów niż jest to potrzebne;
4. nie tworzenie większej ilości kodu produkcyjnego niż jest to potrzebne do przejścia pojedynczego testu.

Metoda TDD ma prawdopodobnie tyle samo zwolenników, co przeciwników. Ci ostatni argumentują, że zbytnia koncentracja wyłącznie na przechodzeniu kolejnych testów może skutkować złą architekturą i użytecznością oprogramowania, jeśli przypadki testowe zostaną nieprawidłowo dobrane. Jeśli programiści nie stosują się do trzeciej z podanych zasad, to mogą utworzyć testy, które będą blokowały wszelką możliwość wprowadzania zmian w kodzie.

TDD jest z całą pewnością użyteczną metodą, ale powinna być stosowana ostrożnie.

Testy jednostkowe

Narzędzia

Powstało wiele bibliotek i platform (ang. *frameworks*) wspierających proces testowania jednostkowego i integracyjnego. W przypadku języka Java najpopularniejszym rozwiązaniem tego typu jest biblioteka *JUnit*, która obecnie jest dostępna w piątej wersji. Interesującą alternatywą dla niej jest TestNG. JUnit jest częścią większej rodziny bibliotek do testów jednostkowych, często nazywanej *x*Unit. Jej pierwszym przedstawicielem była biblioteka SUnit opracowana dla języka programowania Smalltalk.

JUnit, tak jak inne biblioteki do testów jednostkowych, pozwala inżynierom oprogramowania oddzielić kod testów od kodu testowanego. Oznacza to, że testy nie są domyślnie instalowane na środowisku produkcyjnym. Popularne narzędzia do sterowania kompilacją, jak Maven i Gradle obsługują JUnit.

Testy w JUnit są umieszczane w osobnych klasach. Nazwa klasy testów jest najczęściej tworzona z nazwy klasy przez nie weryfikowanej, ale *musi* zaczynać się lub kończyć słowem *Test* lub *Tests*. Każdy test jest implementowany jako oddzielna metoda oznaczona odpowiednią *adnotacją*. Zalecanym jest, aby taka metoda zawierała tylko jedną *asercję*, ale czasem konieczne jest użycie więcej niż jednej.

Testy jednostkowe

Adnotacje

JUnit 5 dostarcza wielu adnotacji umożliwiających tworzenie testów. Niektóre z nich wymieniono w tabeli 1.

Tabela: Adnotacje JUnit 5

Adnotacja	**Opis**
@Test	Oznacza metodę jako test.
@ParameterizedTest	Oznacza metodę jako parametryczny test.
@RepeatedTest	Pozwala powtarzać test.
@TestMethodOrder	Określa kolejność wykonania testów.
@Disabled	Wyłącza test.
@Timeout	Określa limit czasu (ang. *timeout*) dla testu.
@BeforeEach	Metodę wykonywana przez każdym testem.
@AfterEach	Metoda wykonywana po każdym teście.
@BeforeAll	Metoda wykonywana przed wszystkimi testami.
@AfterAll	Metoda wykonywana po wszystkich testach.

Testy jednostkowe

Adnotacje

Metoda testów parametrycznych wymaga dodatkowej adnotacji wskazującej źródło danych wejściowych i oczekiwanych wyników testów. Tą adnotacją jest `@MethodSource`. Jej argumentem jest nazwa statycznej metody, która dostarcza potrzebne dane. Ta metoda musi zwracać strumień obiektów klasy `Arguments`.
Adnotacja `@RepeatedTest` również wymaga argumentu, którym jest liczba powtórzeń testu.

Testy jednostkowe

Asercje

Asercje to warunki, które powinny być zawsze spełnione, jeśli testowany kod działa poprawnie. W terminologii JUnit asercja to metoda statyczna klasy `Assertions`, która sprawdza, czy test zakończył się pomyślnie. Niektóre z tych metod są opisane w tabeli 2. Jest wiele ich przeciążonych wersji dla różnych typów oczekiwanych i rzeczywistych wyników. Niektóre z nich przyjmują dodatkowy argument, którym jest informacja logowana lub wyświetlana, gdy test kończy się niepowodzeniem. JUnit umożliwia użycie asercji zdefiniowanych w innych bibliotekach i platformach, takich jak Hamcrest.

Testy jednostkowe

Asercje

Tabela: Asercje JUnit 5

Asercja	**Opis**
`assertArrayEquals(expected, actual)`	Sprawdza, czy tablice `expected` i `actual` zawierają te same wartości.
`asertEquals(expected, actual)`	Sprawdza czy wartość `expected` jest równa `actual`.
`assertNotEquals(expected,actual)`	Przeciwieństwo poprzedniej asercji.
`assertTrue(condition)`	Zakłada, że warunek `condition` jest prawdziwy.
`assertFalse(condition)`	Zakłada, że warunek `condition` jest fałszywy.
`assertNull(reference)`	Zakłada, że wartość referencji `reference` jest `null`.
`assertNotNull(reference)`	Zakłada, że wartość referencji `reference` nie jest `null`.

Makiety (ang. *Mocks*)

Zautomatyzowane testy jednostkowe i integracyjne mogą wymagać dublerów testowych. Istnieje kilka platform, które pomagają tworzyć makiety i inne, podobne obiekty. W języku Java jedną z nich jest EasyMock, ale na tym wykładzie zostanie zaprezentowana platforma Mockito.
Aby ułatwić testowanie inżynierowie oprogramowania często stosują wzorzec projektowy *wstrzykiwanie zależności*. W tym wzorcu obiektu nie tworzy potrzebnych mu obiektów, ale je otrzymuje. To pozwala programistom łatwiej stosować makiety.
Klasa testów, w której należy zastosować makiety z platformy Mockito musi być oznaczona adnotacją `@ExtendWith`. Argumentem tej adnotacji powinna być obiekt `MockitoExtension.class`. Referencja do obiektu zastępowanego przez makietę powinna być oznaczona adnotacją `@Mock`. Metoda używająca tej referencji powinna najpierw sprawdzić, czy nie jest ona równa `null`. Zachowanie makiety można określić przy użyciu instrukcji `when(condition).thenReturn(result)`.

Makiety (ang. *Mocks*)

Platforma Mockito pozwala także tworzyć szpiegi. Szpieg jest obiektem, który opakowuje inny obiekt i weryfikuje jak się on zachowuje w trakcie testów, np. ile razy jego metody są wywoływane. Szpieg może zostać utworzony za pomocą adnotacji `@Spy` lub poprzez przekazanie rzeczywistego obiektu do metody statycznej `spy()`. Aby sprawdzić ile razy dana metoda szpiegowanego obiektu została wywołana można użyć następującego instrukcji:

```
verify(spyReference, atLeastOnce()).methodName(arguments)
```

Zamiast `atLeastOnce()` można użyci kilku innych metod do zliczania wywołań, takich jak `never()`, `atMostOnce()`.

Testy End-To-End

Testy *End-To-End* (*E2E*) są formą testów funkcjonalnych, wykonywanych na poziomie testowania systemowego. Często są wykonywane najpierw manualnie, a następnie mogą być zautomatyzowane. Istnieje kilka narzędzi, które to umożliwiają. Ich wybór zależny jest od interfejsu weryfikowanego oprogramowania. Jeśli jest to interfejs tekstowy, to zwykłe przekierowanie wejścia i wyjścia oraz użycie skryptów powłoki są wystarczające. W przypadku aplikacji z graficznym interfejsem (GUI) można zastosować narzędzia, które nagrywają interakcję z użytkownikiem i ją odtwarzają. Niektóre z nich potrafią zapisać takie nagranie w postaci skryptu, który później może zostać zmodyfikowany. Przykładem takiego narzędzia jest QF-TEST. Niestety te programy mają pewne ograniczenia, które wykluczają ich użycie w szczególnych przypadkach. Wtedy należy opracować manualnie automatyczne testy E2E korzystając ze specjalnych bibliotek. W przypadku aplikacji napisanych w języku Java z użyciem biblioteki AWT lub Swing można zastosować, odpowiednio, Jemmy i Jemmy 2. Dla programów, których GUI jest oparte na JavaFX dostępna jest biblioteka TestFX.

Testy End-To-End

Biblioteka TestFX

Biblioteka TestFX umożliwia inżynierom oprogramowania stworzenie (wirtualnego) *robota*, który klika przyciski, uzupełnia pola tekstowe, wybiera opcje menu, krótko — używa GUI. Mimo, że dostarcza ona własne asercje, to do przeprowadzania testów wymaga użycia przeznaczonej do tego biblioteki, takiej jak JUnit. Klasa implementująca testy powinna dziedziczyć po klasie `ApplicationTest` biblioteki TestFX. Metody testów mogą uzyskać referencje do elementów GUI używając selektorów CSS dla ich atrybutów, takich jak identyfikatory, klasy i inne. Metodą odpowiedzialną za wyszukiwanie tych elementów jest `lookup()`. Zwraca ona obiekt posiadający metodę `querryAll()`, która zwraca kolekcję elementów z wymaganym atrybutem. Pojedynczy obiekt z tej kolekcji można uzyskać za pomocą metody `next()` iteratora zwróconego przez metodę `iterator()` kolekcji. Aby kliknąć ten element należy użyć metody `clickOn()`.
TestFX ma swój własny zbiór obiektów weryfikujących (ang. *matchers*), czyli takich, które sprawdzają, czy określony warunek jest spełniony. Przykładowo metoda `hasText()` z klasy `LabelMatchers` sprawdza czy określony element GUI ma wymagany opis. W tej bibliotece znajduje się również klasa `FXAssert` zawierająca asercje, jak `verifyThat()`.

Testy End-To-End

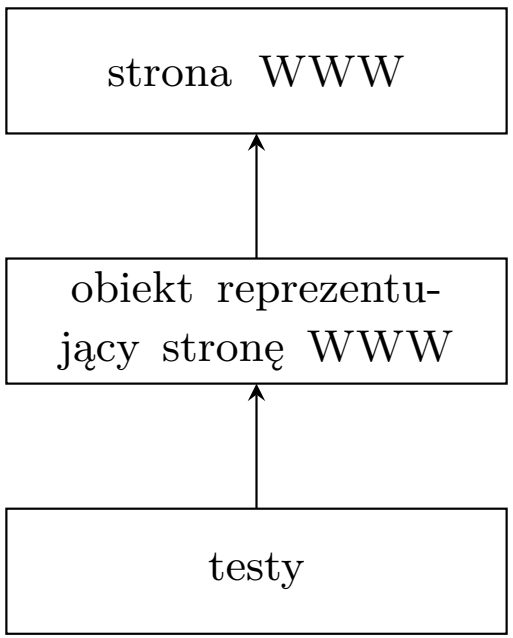
Testy automatyczne aplikacji internetowych

Testy aplikacji internetowych mogą zostać zautomatyzowane przy użyciu narzędzi takich jak QF-TEST, ale istnieją również inne rozwiązania, które pozwalają inżynierom oprogramowania tworzyć takie testy manualnie. Obecnie wszystkie one używają protokołu WebDriver, który pozwala im na interakcje i sterowanie przeglądarką WWW. Wszystkie nowoczesne przeglądarki wspierają to API, a ich producenci dostarczają niezbędnych sterowników. Do tych rozwiązań zaliczają się platformy, takie jak Cypres i biblioteki, np. Selenide. Najstarszym tego typu narzędziem i prawdopodobnie najbardziej popularnym jest biblioteka Selenium.

Testy End-To-End

Testy automatyczne aplikacji internetowych

Najczęściej wykorzystywanym wzorcem projektowym w oprogramowaniu implementującym testy aplikacji internetowych jest wzorzec *PageObject* (dosłownie: *obiekt strony*)[1]. Wymaga on opracowania obiektów, które reprezentują i sterują stronami WWW testowanej aplikacji. Rysunek 1 przedstawia koncepcyjnie zastosowanie tego wzorca.



Rysunek: Wzorzec projektowy PageObject

Testy End-To-End

Testy automatyczne aplikacji internetowych — Selenium

Najważniejszą klasą w bibliotece Selenium jest `WebDriver`, pozwalająca testom na dostęp i sterowanie elementami strony WWW. Elementy te mogą być wyszukane przy użyciu metod `findElement()` i `findElements()`, które pozwalają zastosować klika typów *lokalizatorów*, z których najpopularniejszym jest *XPath*. Do innych zaliczane są selektory CSS, identyfikatory, nazwy, nazwy klas, nazwy znaczników, hiperłącza i częściowe opisy hiperłączy. Metoda `get()` z klasy `WebDriver` nakazuje przeglądarce WWW załadowanie strony o podanym adresie.
Pojedynczy element strony WWW jest reprezentowany przez obiekt klasy `WebElement`. Selenium pozwala niejawnie (ang. *implicit*) i jawnie (ang. *explicit*) oczekiwać aż element strony WWW stanie się dostępny. Czas oczekiwania niejawnego jest definiowany globalnie, natomiast strategie oczekiwania jawnego są dostępne za pomocą obiektów klas `WebDriverWait` i `FluentWait`. Ta druga oferuje więcej możliwości niż pierwsza.

Bibliografia

 Unmesh Gundecha. *Selenium i testowanie aplikacji: Receptury*. Helion, Gliwice, 2017.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!