

**Wydział Fizyki i Informatyki Stosowanej**

Praca inżynierska

**Zastosowanie metodyki Test Driven Development do tworzenia aplikacji internetowych opartych na Java Spring**

Test Driven Development of web applications

based on Java Spring

Autor: Hleb Shypula

Kierunek studiów: Informatyka Stosowana

Opiekun pracy: dr hab. inż. Maciej Wołoszyn

Kraków, 2021/2022

1. **Wprowadzenie**

Moją ambicją jest elegancki, elastyczny i zrozumiały kod, który można łatwo modyfikować, który działa poprawnie i nie sprawia przykrych niespodzianek podczas tworzenia oprogramowania. Aby zrealizować ten zamysł, spróbuję przetestować program, zanim go napiszę. To właśnie ta paradoksalna idea stanowi podstawę metodyki Test-Driven Development (programowanie sterowane testami). Czy to jest nielogiczne? Ja nie spieszyłem się z wyciąganiem pochopnych wniosków.

Głównym celem niniejszej pracy było napisanie aplikacji internetowej opartej na Java Spring przy zastosowaniu metodyki Test-Driven Development. A mianowicie:

* Przedstawienie zasadniczych cech metodyki Test Driven Development;
* Zdobywanie osobistego doświadczenia w programowaniu sterowanym testami;
* Sprawdzenie, czy poziom wejścia do tej metodyki jest akceptowalny dla programisty niezbyt doświadczonego w jakimkolwiek testowaniu;
* Przejście po większości wzorców TDD, ich porównanie między sobą oraz zdefiniowanie dla każdego najbardziej pasujących zadań w tworzeniu oprogramowania.

Temat „Zastosowanie metodyki Test Driven Development do tworzenia aplikacji internetowych opartych na Java Spring” ma na celu wskazać czytelnikowi, że w odróżnieniu od XP (Extreme Programming) metodyka TDD nie jest absolutna. XP mówi: „Mamy rzeczy, które powinniśmy opanować, zanim przejdziemy do kolejnego etapu”. Natomiast TDD – to nie tak konkretna metodyka. TDD zakłada, że podczas programowania w pełni wyobrażamy sobie odległość od momentu podjęcia decyzji o designie do etapu kontroli jakości otrzymanych wyników. Także TDD udostępnia techniki umożliwiające kontrolowanie tej odległości[[1]](#footnote-1).

Główną tezą niniejszego opracowania jest stwierdzenie, że dzięki TDD kod będzie zawierał znacznie mniej uszkodzeń i niedoskonałości. Jeżeli tak, to napisanie kodu, którego wymaga test by przejść, produkuje tezę podrzędną – codziennie będziemy w stanie dostarczać w pełni działający produkt wraz z dodaną nową funkcjonalnością. Dzięki temu współpraca z wirtualnym klientem osiągnie nowy poziom.

1. **Wstęp teoretyczny**

**2.1 Definicja TDD**

Programowanie sterowane testami (ang. *test-driven development, TDD*) – technika tworzenia oprogramowania, polegająca na powtórzeniu bardzo krótkich cykli projektowania: najpierw pisze się test, pokrywający żądane zmiany, następnie pisze się kod, którego wymaga test by przejść, i w końcu przeprowadza się refaktoryzacja nowego kodu, żeby spełniał odpowiednie standardy. Kent Beck, uznawany za wynalazcę tej techniki, argumentował w 2003 r., że programowanie oparte na testach zachęca do prostego designu i wzbudza wiarę w siebie (ang. *inspires confidence*)[[2]](#footnote-2).

W 1999 roku, kiedy TDD pojawiło się po raz pierwszy, technologia była ściśle związana z koncepcją test-first, stosowaną w programowaniu ekstremalnym, ale później pokazała się jako niezależna metodyka. TDD różni się od innych metod tym, że łączy programowanie z pisaniem testów przez tego samego programistę. Ta koncepcja odnawia powszechny szacunek dla testów tworzonych przez programistę[[3]](#footnote-3).

Test to procedura, która pozwala potwierdzić lub zdementować, że kod działa. Testy funkcjonalne pozwalają zobaczyć funkcjonalność aplikacji z punktu widzenia użytkownika. Oznacza to, że testy funkcjonalne mogą być rodzajem specyfikacji aplikacji. Można wydzielić: testy czarnej skrzynki, testy w przeglądarce, testy behawioralne, testy wydajnościowe. Natomiast testy jednostkowe testują aplikację od środka, z punktu widzenia programisty. Testowanie wtedy odbywa się w celu upewnienia się, że obiekty (moduły, podsystemy) działają poprawnie.

**2.2 Wymagania TDD**

Test Driven Development wymaga od dewelopera tworzenia automatycznych testów jednostkowych, które definiują wymagania dla kodu jeszcze przed napisaniem rzeczywistego kodu. Test zawiera procedury sprawdzające warunki, które mogą przejść lub nie przejść. Powodzenie testu weryfikuje zachowanie zgodne z zamierzeniami programisty. Deweloperzy często używają frameworków testowych do tworzenia i automatyzacji wykonywania zestawów testów. W praktyce testy jednostkowe obejmują krytyczne i nietrywialne sekcje kodu. Może to być kod podlegający częstym zmianom lub kod z dużą liczbą zależności.

TDD obejmuje nie tylko walidację, ale także wpływa na design programu. Opierając się na testy, programiści mogą szybciej wyobrazić sobie, jakich funkcjonalności potrzebuje użytkownik. Wtedy szczegóły interfejsu pojawiają się długo przed ostatecznym wdrożeniem rozwiązania.

**2.3 Cykl TDD**

**1. Napisanie testu**

W programowaniu opartym na testach wcielenie każdej nowej funkcjonalności do aplikacji zaczyna się od napisania testu. Aby to zrobić, programista powinien z pewnością rozumieć wymagania do nowej funkcjonalności. W tym celu brane są pod uwagę możliwe przypadki użycia i historyjki użytkownika. Nowe wymagania mogą również wymuszać zmiany w istniejących testach. I właśnie ten pierwszy punkt cyklu odróżnia programowanie sterowane testami od technik, w których testy są tworzone po napisaniu kodu.

**2. Uruchomienie wszystkich testów: nowe testy powinny się zakończyć niepowodzeniem**

Właśnie napisany test nieuchronnie skończy się niepowodzeniem, ponieważ odpowiedni kod nie jest jeszcze napisany. Natomiast jeżeli test przeszedł, oznacza to, że rozważana funkcjonalność już istnieje lub napisany test ma wady. Wiedząc to, na tym etapie sprawdza się, czy napisane właśnie testy nie przechodzą. One nie powinny przejść z oczywistych dla dewelopera powodów. Zwiększy to pewność (chociaż nie gwarantuje całkowicie), że test rzeczywiście sprawdza to, do czego został zaprojektowany.

**3. Napisanie kodu**

W tym momencie programista powinien zaprojektować nowy kod, aby wcześniej napisany test przeszedł pomyślnie. Ten kod nie musi być doskonały. Dozwolone jest, aby zdał „egzamin” w jakiś nieelegancki sposób. Jest to akceptowalne, ponieważ kolejne kroki go poprawią i wypolerują. Ważne jest, aby napisać kod specjalnie do przejścia konkretnego testu. Nie należy dodawać zbędnych, a za tym, niepokrytych testami funkcjonalności.

**4. Uruchomienie wszystkich testów: wszystkie testy powinny się zakończyć powodzeniem**

Jeśli wszystkie testy zakończą się powodzeniem, programista może mieć pewność, że kod spełnia wszystkie testowane wymagania. Następnie można przejść do ostatniego etapu cyklu.

**5. Refaktoryzacja**

Po osiągnięciu wymaganej funkcjonalności kod można „posprzątać”. Refaktoryzacja to proces zmiany wewnętrznej struktury programu, który nie wpływa na jego zewnętrzne zachowanie i ma na celu ułatwienie zrozumienia jego działania, wyeliminowanie duplikacji kodu oraz ułatwienie wprowadzania zmian w najbliższej przyszłości[[4]](#footnote-4).

Opisany cykl powtarza się, wdrażając coraz więcej nowych funkcjonalności. Jeśli nowy kod nie spełnia odpowiednich testów lub stare testy przestają przechodzić, programista powinien zwrócić się do debugowania lub po prostu cofnąć się do poprzedniej wersji aplikacji i spróbować podejść do problemu z innej strony. Kroki powinny być małe, pozwoli to na łatwiejsze zidentyfikowanie błędów w przypadku ich wystąpienia, a w przypadku nierozwiązalnych trudności cofnięcie do poprzedniej wersji nie będzie aż tak bolesne.

**2.4 Dlaczego TDD**

Przed rozpoczęciem tworzenia aplikacji, zastanowiłem się, dlaczego chcę użyć TDD, jakie problemy, które spotkałem w swoich poprzednich projektach ta metodyka by pokonała. Podczas przygotowywania się do napisania pracy i czytania tematycznej literatury wymieniłem dla siebie kilka bolączek, które cały czas nie dawały mi spokoju oraz które TDD ma na celu wyleczyć:

* Większość moich testów były manualne;
* Jeżeli miałem testy automatyczne, to nie wykrywały one rzeczywiste problemy;
* Testy automatyczne były pisane i przeprowadzane za późno, żeby wnieść użyteczny wkład w rozwój aplikacji;
* Zawsze miałem coś ważniejszego do zrobienia, niż napisanie testów;
* Niemożliwość refaktoryzacji kodu ze względu na strach zepsucia się jakichś funkcjonalności;
* Сiągły lęk przed wdrożeniem aplikacji, ponieważ na pewno coś opuściłem podczas przeprowadzenia swoich biednych testów[[5]](#footnote-5).

Oczywiście, rozumiałem, że TDD nie pokona magicznie wszystkie te problemy, ale miałem nadzieję, że wyciągnę dla siebie niezmierzony postęp na tych poziomach tworzenia oprogramowania.

Także na początku wydzieliłem kilka zasadniczych zalet TDD, i nie mogłem się doczekać, żeby sprawdzić je na własnym przykładzie. Wyszczególnię zwłaszcza kilka z nich:

* Zmniejsza zależność od debugowania – ponieważ najpierw nacisk jest na pisanie testów, a następnie generowanie kodu, aby przejść te testy, wiele programistów uważa, że znacznie zmniejsza się potrzeba debugowania. Ponieważ podczas pisania i kodowania testu wymagane jest głębsze zrozumienie wymagań logicznych i funkcjonalnych, często można szybko zidentyfikować i rozwiązać przyczynę niepowodzenia testu;
* Programowanie sterowane testami oferuje więcej niż tylko walidację kodu, ma również wpływ na projektowanie aplikacji. Skupiając się początkowo na testach, łatwiej wyobrazić sobie, jakiej funkcjonalności potrzebuje użytkownik. W ten sposób programista zastanawia się nad szczegółami interfejsu przed napisaniem kodu. Testy zmuszają nas do pisania kodu bardziej testowalnego. Na przykład, porzucenie zmiennych globalnych, singletonów, sprawia, że klasy są mniej spójne i łatwiejsze w użyciu. Silnie powiązany kod będzie znacznie trudniejszy do przetestowania. Testy jednostkowe przyczyniają się do tworzenia przejrzystych i małych interfejsów. Każda klasa będzie miała określoną (zwykle małą) rolę;
* Testy pozwalają na refaktoryzację kodu bez ryzyka zepsucia go. Wprowadzając zmiany w dobrze przetestowanym kodzie, ryzyko nowych błędów jest znacznie mniejsze. Jeśli nowa funkcjonalność prowadzi do błędów, testy natychmiast to wykryją. Dobrze pokryty testami kod jest łatwy do refaktoryzacji. Programiści będą mieli dużo większą swobodę, jeśli chodzi o wprowadzanie zmian w architekturze, które mają na celu ulepszenie projektu;
* Testy mogą służyć jako dokumentacja. Dobry kod powie, jak działa lepiej niż jakakolwiek dokumentacja. Dokumentacja i komentarze do kodu mogą być nieaktualne – może to być mylące dla programistów czytających kod. A ponieważ dokumentacja, w przeciwieństwie do testów, nie może powiedzieć, że jest przestarzała, nierzadko zdarzają się sytuacje, w których dokumentacja nie odpowiada rzeczywistości;
* Praktyka pokazuje, że ogólny czas pracy nad projektem jest skrócony w porównaniu z tradycyjnymi metodami tworzenia oprogramowania. Podczas gdy całkowita liczba linijek kodu wzrasta (dzięki testom), częstsze testowanie eliminuje błędy w procesie i znacznie wcześniej identyfikuje istniejące, zapobiegając problemom później[[6]](#footnote-6).

**2.5 Fazy testowania**

**Testowanie integracyjne –** tojedna z faz testowania oprogramowania, w której poszczególne warstwy oprogramowania są łączone i testowane w grupie[[7]](#footnote-7).

* W testach integracyjnych z źródłem danych skupię się na interakcji tego źródła z klasą-repozytorium.

Klasa-repozytorium – to komponent, który hermetyzuje logikę wymaganą do uzyskania dostępu do źródeł danych. Implementuje on ogólną funkcjonalność dostępu do danych i zapewnia infrastrukturę, potrzebną do uzyskania dostępu do bazy danych z warstwy modelu aplikacji[[8]](#footnote-8).

Ten typ testowania integracyjnego przeprowadza się w celu upewnienia się, że mapowanie obiektowo-relacyjne działa poprawnie, semantyka zapytań (jeżeli są samodzielnie zdefiniowane) nie jest naruszona oraz konfiguracja połączeń działa słusznie.

1. **Opis aplikacji**

**3.1 Charakterystyka problemu**

Zadaniem aplikacji jest kontrola oraz obsługa procedury i warunków prowadzenia certyfikacji zawodowej pracowników medycznych, farmaceutycznych i innych pracowników opieki zdrowotnej zgodnie z rozporządzeniem ministerstwa zdrowia Białorusi z dnia 28 maja 2021 r[[9]](#footnote-9).

Niżej przedstawiłem tłumaczenie najbardziej znaczących zagadnień z tego rozporządzenia. Wiedząc fakt, że testy mogą służyć jako dokumentacja, w rozdziale poświęconym implementacji postaram się pokazać jak poniższe urywki najpierw realizują się w testach, a później implementują się w kodzie. Tym zdaniem chciałem powiedzieć, że czytacze tej pracy nie powinny się martwić, jeżeli jakieś rzeczy nie będą zrozumiałe na bieżąco.

Certyfikacja jest przeprowadzana w celu przypisania lub potwierdzenia kategorii kwalifikacyjnej pracownika. Przypisanie lub potwierdzenie kategorii kwalifikacyjnej odbywa się sekwencyjnie – druga, pierwsza, wyższa.

Rozporządzenie określa, że pracownicy sfery medycznej, mające w dniu 23 lipca 2021 r. obecną kategorię kwalifikacyjną, powinni przejść certyfikację zawodową w celu potwierdzenia własnej kategorii w ciągu 4 lat i 9 miesięcy od ww daty.

Certyfikacji podlegają:

* kadra kierownicza i specjaliści medyczni oraz farmaceutyczni, pracownicy opieki zdrowotnej;
* przedstawiciele służby zdrowia;
* kadra dydaktyczna państwowych placówek edukacyjnych, realizujących szkolenia i przekwalifikowanie specjalistów o profilu kształcenia „Medycyna” (oraz specjalność „Farmacja”);
* inni pracownicy medyczni o wykształceniu wyższym lub średnim, pracujący w jednostkach służby zdrowia, placówkach edukacyjnych, szkoleniowych, specjaliści o profilu kształcenia „Opieka zdrowotna”.

Pracownicy mogą ubiegać się o wyższą kategorię kwalifikacyjną nie wcześniej niż trzy lata od daty wydania poprzedniej kategorii kwalifikacyjnej. Certyfikacja na potwierdzenie kategorii kwalifikacyjnej przeprowadzana jest raz na pięć lat. Dokumentację na potwierdzenie kategorii kwalifikacyjnej należy złożyć do komisji atestacyjnej nie później niż trzy miesiące przed wygaśnięciem kategorii kwalifikacyjnej. Certyfikacja do przypisania drugiej kategorii kwalifikacyjnej a potwierdzenie kategorii kwalifikacji jest obowiązkowe.

Z obowiązku certyfikacji zwolnieni są:

* pracownicy, którzy pracowali na odpowiednim stanowisku krócej niż rok;
* kobiety w ciąży;
* pracownicy, przebywające na długotrwałym (powyżej czterech miesięcy) leczeniu – na 6 miesięcy po wyjściu do pracy;
* osoby odbywające staż w szkole wyższej, studia doktoranckie – w okresie studiów, szkolenia;
* osoby przebywające na urlopie macierzyńskim, na urlopie wychowawczym dla dziecka do ukończenia trzeciego roku życia – na rok po zwolnieniu.

Kategorie kwalifikacyjne są przydzielane na podstawie kwalifikacji, odpowiednie z stanowiskiem pracownika (prowizor, prowizor-organizator, farmaceuta, lekarz organizujący opiekę zdrowotną, lekarz-ekspert).

Aby przypisać pracownikowi kategorię kwalifikacyjną, konieczne jest:

* druga kategoria kwalifikacyjna – mieć wiedzę teoretyczną oraz praktyczne umiejętności z zakresu działalności zawodowej, doświadczenie w pracy w odpowiedniej specjalności – co najmniej 3 lata, odbyć min. 100 godzin szkoleń zawodowych (dla pracowników o wykształceniu średnim medycznym lub farmaceutycznym – co najmniej 50 godzin);
* pierwsza kategoria kwalifikacyjna – mieć wiedzę teoretyczną oraz praktyczne umiejętności z zakresu działalności zawodowej, doświadczenie w pracy w odpowiedniej specjalności – co najmniej 6 lat, staż pracy w drugiej kategorii kwalifikacyjnej – co najmniej 3 lata, odbyć min. 160 godzin szkoleń zawodowych (dla pracowników o wykształceniu średnim medycznym lub farmaceutycznym – co najmniej 80 godzin);
* wyższa kategoria kwalifikacyjna – mieć wiedzę teoretyczną oraz praktyczne umiejętności z zakresu działalności zawodowej, doświadczenie w pracy w odpowiedniej specjalności – co najmniej 9 lat, staż pracy w pierwszej kategorii kwalifikacyjnej – co najmniej 3 lata, odbyć min. 160 godzin szkoleń zawodowych (dla pracowników o wykształceniu średnim medycznym lub farmaceutycznym – co najmniej 80 godzin).

Aby potwierdzić kategorię kwalifikacyjną pracownik powinien odbyć min. 160 godzin szkoleń zawodowych (dla pracowników o wykształceniu średnim medycznym lub farmaceutycznym – co najmniej 80 godzin).

Akumulacja wymaganego czasu szkoleń zawodowych, niezbędnych do przypisania (potwierdzenia) kategorii kwalifikacyjnej, odbywa się w ciągu 5 lat. Warunkiem koniecznym do przypisania (potwierdzenia) kategorii kwalifikacyjnej jest samodzielne opanowanie materiałów szkoleniowych, dla pracowników o wykształceniu wyższym medycznym w wymiarze co najmniej 80 godzin (dla pracowników z wykształceniem średnim medycznym lub farmaceutycznym - co najmniej 40 godzin). Pracownicy, którym przypisano kategorię kwalifikacyjną, którzy przeszli na nowe stanowisko służbowe, są dopuszczone do obrony na równorzędną kategorię kwalifikacyjną na nowym stanowisku pracy za dwa lata od daty rozpoczęcia pracy na nowym stanowisku. W tym czasie ważna jest kategoria kwalifikacyjna przypisana na poprzednim stanowisku pracy.

**3.2 Wymagania**

Aplikacja powinna:

* pobierać ogólne dane pracowników firmy z pliku .csv, wyeksportowanego z programu księgowego «1С Бухгалтерия»[[10]](#footnote-10);
* pozwalać na ręczne wprowadzenie oraz modyfikację wszystkich potrzebnych danych dotyczących certyfikacji pracownika;
* obliczać oraz przedstawiać program certyfikacji osobno dla każdego pracownika zgodnie z zasadami opisanymi w odpowiednim dekrecie;
* z wyprzedzeniem informować osobę obsługiwającą aplikację (osobę odpowiadającą za certyfikację pracowników w firmie) o zbliżających się terminach dotyczących certyfikacji;
* generować potrzebną do certyfikacji pracownika dokumentację.

1. **Implementacja**

Aplikacja powinna powstać przy pomocy programowania sterowanego testami. Oznacza to, że zaczynam implementację od czystego projektu. Nie ma na razie nic oprócz dokumentu, który stanowi logikę biznesową programu. Czyli będę szedł z implementacją krok po kroku, czasami te kroki będą małe, czasami duże. Mogę pisać testy w taki sposób, że każdy z nich będzie wymagał dodania jedynej linii kodu i niewielkiej refaktoryzacji. Także mogę pisać testy tak skomplikowane, że będą wymagać dopisania setek linii kodu i godziennej refaktoryzacji. Nie ma lepszego podejścia z tych dwóch. W TDD powinniśmy umieć pracować na oba te sposoby.

Zacząć pasowałoby z mniejszych kroków, korzystając z patternów „Fake It” oraz „Triangulate”, żeby nauczyć się pracować w takim tempie i odczuć rytm TDD. Kiedy uda mi się zdobyć pewność siebie i wszystko będzie szło gładko spróbuję przejść na większą prędkość, czyli skorzystam z patternu „Obvious Implementation” – testy będą wymagały więcej kodu, a będę wiedział jak ten kod ma wyglądać. Ale zawsze będę wracał do mniejszych kroków, jeśli dostanę głupi błąd tam, gdzie implementacja wydawała się być oczywista.

**4.1 Podstawa modelu**

Nazwałem pierwszy krok implementacji programu jako „podstawa modelu”. Ponieważ tu stworzymy kilka podstawowych klas modelu aplikacji, dodamy tam wymagane do przejścia testów pola oraz metody. Da nam to impuls do dalszego rozwoju logiki programu.

TDD zakłada, że powinniśmy pisać tylko ten kod, którego wymaga test. Czyli daje mi to prawo brać poszczególne wycinki z dokumentu stawiącego logikę biznesową, pisać testy, a następnie kod, by przejść te testy. Pierwszy urywek brzmi tak:

Pracownicy sfery medycznej, mające w dniu 23 lipca 2021 r. obecną kategorię kwalifikacyjną, powinni odbyć certyfikację zawodową w celu potwierdzenia własnej kategorii w ciągu czterech lat i dziewięciu miesięcy od ww daty. Dokumenty potwierdzające odbycie certyfikacji zawodowej powinny być złożone do komisji nie później niż trzy miesiące do końca pięciu lat od ww daty[[11]](#footnote-11).

Teraz chciałbym ustrukturyzować to jako listę podpunktów:

* + - 1. Mamy datę punktu odniesienia.
      2. Odnosi się to tylko do pracowników mających na tę datę obecną kategorię.
      3. Pracownicy powinni odbyć certyfikację w ciągu czterech lat i dziewięciu miesięcy od tej daty.
      4. Dokumenty na potwierdzenie kategorii powinny trafić do komisji nie później niż trzy miesiące do końca pięciu lat od tej daty.

**Chciałbym się najpierw zająć podpunktem 2, który wymusza mi stworzyć pracownika zawierającego kategorię.** Brzmi jak test:

1. @Test
2. public void testEmployeeCategory() {
3. }

Teraz na tak prostym przykładzie chciałbym pokazać jak działa pattern „Assert first”, żeby potem było wiadomo o czym mówię na bardziej skomplikowanych przykładach. Czyli najpierw dodaję metodę statyczną sprawdzającą kategorię pracownika do testu. I właśnie w tym momencie wymyślam nazwy poszczególnych klas i pól:

1. @Test
2. public void testEmployeeCategory() {
3. **assertEquals(employee.category, "Kategoria X");**
4. }

Następnie przy pomocy podpowiedzi kompilatora dodaję nieistniejący obiekt:

1. @Test
2. public void testEmployeeCategory() {
3. **Employee employee = new Employee();**
4. assertEquals(employee.category, "Kategoria X");
5. }

Napisany właśnie test nawet się nie kompiluje, ale łatwo to poprawić. Jak to zrobić najprościej – na sztywno dodać brakujące rzeczy:

1. public class Employee {
2. public String category;
3. }

Widzę, że jest źle – mamy pole publiczne w klasie. Ale teraz test się kompiluje i możemy zobaczyć jeden z kroków TDD w akcji. Napisaliśmy test, który nie przechodzi:

org.opentest4j.AssertionFailedError:

Expected :null

Actual :Kategoria X

Najważniejsze jest, że mamy postęp. Teraz mamy oderwać się od globalnego celu „odbycie certyfikacji pracownikiem w terminie” i przełączyć się na mały kroczek z kategorią. Chcemy zmusić ten test na przejście, a nie interesuje nas teraz piękne rozwiązanie. Najmniejsza zmiana, która sprawi, że test przebiegnie pomyślnie, wygląda tak:

1. public class Employee {
2. public String category **= "Kategoria X";**
3. }

Ale jak można teraz zauważyć, mamy duplikowanie pomiędzy testem a kodem realizacji. Widzę jak to naprawić, jest to oczywiste. Także muszę pamiętać, że poruszamy się teraz małymi krokami, żeby nauczyć się takiego tempu i odczuć rytm TDD. Usuwam duplikowanie przez dodanie konstruktora z odpowiednim parametrem, także mogę pozwolić sobie zrobić pole klasy prywatnym i wygenerować getter:

1. public class Employee {
2. **private** String category;
4. public Employee(**String category**) {
5. **this.category = category;**
6. }
8. public String getCategory() {
9. return category;
10. }
11. }

Także powinienem wnieść zmiany do testu:

1. @Test
2. public void testEmployeeCategory() {
3. Employee employee = **new Employee("Kategoria X");**
4. assertEquals(employee.**getCategory()**, "Kategoria X");
5. }

Test dalej przechodzi pomyślnie, kod wygląda już lepiej. Mogę teraz przejść do kolejnych podpunktów. Wybieram podpunkt 1, dla przypomnienia brzmi on tak:

**Mamy datę punktu odniesienia jako 23.07.2021 r.**

Chcemy gdzieś przechowywać tę datę, ponieważ jest to data wejścia naszego rozporządzenia w życie i będziemy chcieli nie raz z niej skorzystać. Wybieram sposób przechowywania jako konstantę w już instniejącej klasie.

1. public class Employee {
2. public static final LocalDate ACT\_ENTRY\_INTO\_FORCE\_DATE = LocalDate.of(2021,7,23);
3. //...
4. }

Dalej przechodzimy do podpunktów 3 oraz 4:

* + - 1. **Pracownicy powinni odbyć certyfikację w ciągu czterech lat i dziewięciu miesięcy od tej daty.**
      2. **Dokumenty na potwierdzenie kategorii powinny trafić do komisji nie później niż trzy miesiące do końca pięciu lat od tej daty.**

Jeżeli wiem, że to się odnosi tylko do pracowników, którzy na moment wejścia rozporządzenia w życie mają obecną kategorię, to logiczne jest, że klasa odwzorowująca pracownika (*Employee*) ma zawierać pole na przechowywanie daty potwierdzenia (nadania) kategorii. Także zgodnie z wymaganiami aplikacji będę musiał informować użytkownika o zbliżającym się ostatecznym terminie (deadlinie) dla potwierdzenia kategorii pracownika. Zakładam, że użytkownik będzie chciał samodzielnie wybierać preferowany termin wcześniejszego powiadomienia w dniach. Mogę pozwolić sobie trochę przyspieszyć i napisać większy test:

1. @Test
2. public void testEmployeeDeadlineNotification() {
3. Employee employee = new Employee();
4. employee.setCategory("Kategoria X");
5. employee.setCategoryAssignmentDeadlineDate(Employee
6. .ACT\_ENTRY\_INTO\_FORCE\_DATE.plusYears(5));
8. employee.setDocsSubmitDeadlineDate(Employee
9. .ACT\_ENTRY\_INTO\_FORCE\_DATE.plusYears(4).plusMonths(9));
11. NotificationTerm notificationTerm = new NotificationTerm();
13. notificationTerm.setDays(365);
14. assertEquals(Math.abs(DAYS.between(employee
15. .getCategoryAssignmentDeadlineDate(), LocalDate.of(2025, 7, 23))),
16. notificationTerm.getDays());
18. notificationTerm.setDays(200);
19. assertTrue(Math.abs(DAYS.between(employee
20. .getDocsSubmitDeadlineDate(), LocalDate.of(2025, 7, 23)))
21. > notificationTerm.getDays());
22. }

 Napisany właśnie test skończył się powodzeniem po dopisaniu odpowiednich klas, pól i metod nawet bez dodawania żadnych hardcode’owych wartości, ponieważ wszystko się definiuje w miejscu. Dlatego chciałbym przetestować realne użycie klasy NotificationTerm, kótra będzie odwzorowywać tabelę w bazie danych. Jest to test integracyjny z warstwą bazodanową, a myśle, ż ten temat można wynieść jako osobny, dlatego opiszę sposób postępowania w kolejnym podrozdziale.

**4.2 Test integracyjny – warstwa danych**

W testach integracyjnych z źródłem danych skupię się na interakcji pomiędzy klasą-repozytorium a źródłem danych. Stworzyłem nową klasę testową, którą skonfigurowałem na potrzeby testów integracyjnych z warswą danych, dodałem test identyczny do ostatniego testu z poprzedniego podpunktu. Jedyną różnicą jest to, że teraz wysyłam obiekt klasy *NotificationTerm* do bazy, a następnie tworzę nowy obiekt, który inicjalizuję znalezionym na podstawie *id* rekordem z bazy dancyh. Oczywiście dodałem jeszcze metody przygotowywujące dane do testowania oraz czyszczące, ponieważ te dane będą się często powtarzać dla różnych testów. Klasa wygląda tak:

1. @RunWith(SpringRunner.class)
2. @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
3. @DataJpaTest
4. public class DAOIntegrationTests {
6. @Autowired
7. private TestEntityManager entityManager;
9. private Employee employee;
10. private NotificationTerm notificationTerm;
12. @Before
13. public void before() {
14. employee = new Employee();
15. employee.setCategory("Kategoria X");
16. employee.setCategoryAssignmentDate(LocalDate.of(2020,5,5));
17. employee.setCategoryAssignmentDeadlineDate(Employee
18. .ACT\_ENTRY\_INTO\_FORCE\_DATE.plusYears(5));
19. employee.setDocsSubmitDeadlineDate(Employee
20. .ACT\_ENTRY\_INTO\_FORCE\_DATE.plusYears(4).plusMonths(9));
22. notificationTerm = new NotificationTerm();
23. }
25. @After
26. public void after() {
27. employee = null;
28. notificationTerm = null;
29. }
31. @Test
32. public void testNotificationTermDAO() {
33. notificationTerm.setDays(365);
34. entityManager.persist(notificationTerm);
35. entityManager.flush();
36. NotificationTerm foundNotificationTerm = entityManager
37. .find(NotificationTerm.class, notificationTerm.getId());
38. assertEquals(Math.abs(DAYS.between(employee
39. .getCategoryAssignmentDeadlineDate(), LocalDate.of(2025, 7, 23))),
40. foundNotificationTerm.getDays());
42. notificationTerm.setDays(200);
43. entityManager.persist(notificationTerm);
44. entityManager.flush();
45. foundNotificationTerm = entityManager
46. .find(NotificationTerm.class, notificationTerm.getId());
47. assertTrue(Math.abs(DAYS.between(employee
48. .getDocsSubmitDeadlineDate(), LocalDate.of(2025, 7, 23)))
49. > foundNotificationTerm.getDays());
50. }
51. }

W linii 3 dodałem adnotację z parametrem, która mówi, że chcę komunikować nie z wbudowanym, tylko z rzeczywistem źródłem danych.

Podczas wykonywania testu w linijkach 34 oraz 43 wysłałem obiekt do bazy, w linijkach 36 oraz 45 pobrałem go z bazy, a po zakończeniu testu baza danych została zwrócona do początkowego stanu z pustą tabelą *notification\_term*. Mianowicie tak działa metoda testowa znajdująca się w klasie z adnotacją @DataJpaTest (3 linijka), jest to bardzo użyteczna funkcjonalność.

Także wywołuję metodę *flush* w linijkach 35 oraz 44, która powoduję rzeczywiste wykonanie zapytania w bazie. Pozwala mi to uzyskać *id* wysłanego przed chwilą obiektu na potrzeby dalszego odnalezenia go w bazie.

W celu poprawnej kompilacji oraz przejścia testu do klasy NotificationTerm zostały dodane kilka niezbędnych adnotacji. Teraz kod klasy wygląda tak:

1. @Entity
2. public class NotificationTerm {
4. @Id
5. @GeneratedValue(strategy = GenerationType.AUTO)
6. @Column(columnDefinition = "serial")
7. private long id;
9. private int days;
10. // getters, setters
11. }

Podsumowując podrozdziały 4.1 oraz 4.2 mogę powiedzieć, że udało mi się rozłożyć jeden wielki problem terminowej certyfikacji pracowników na 4 podproblemy, co pozwoliło mi skupić się osobno nad każdym i zrealizować to w miarę płynnie i bez stresu. Nie napisałem żadnej linii kodu, której by nie wymagał test. Dlatego mogę powiedzieć, że kod działa bezpiecznie, co wzbudza pewność siebie i chęć kontynuować programowanie kolejnych zagadnień logiki biznesowej.

**4.3 Rozszerzenie modelu**

Z wymagań wynika, że pracownik może być aktywmy lub nieaktywny. Zatem chciałbym dodać kolejny test:

1. @Test
2. public void testEmployeeActive() {
3. assertTrue(employee.isActive());
4. }

Teraz chciałbym na tak prostym przykładzie pokazać zastosowanie patternu fałszywej realizacji – „Fake it”. Jak wynika z teorii, chcę zastosować hardcode’ową wtyczkę, zwracającą konstantę, a za każdym krokiem zamieniać ję na zmienną, aż dopóki nie otrzymam prawdziwej realizacji. Najpierw chciałbym wykonać najmniwjszy krok, który sprawi, że test się wykona pomyślnie. Robię to poprzez dodanie odpowiedniej metody, w której zwracam hardcode’ową wartość:

1. public class Employee {
2. // ...
3. public boolean isActive() {
4. return true;
5. }
6. }

Test przechodzi pomyślnie. Ale przecież nie zadziała to przy „nieaktywnym” pracowniku, wartość *true* jest hardcode’owa. Czyli problemem jest duplikowanie pomiędzy kodem a testem – *assertTrue* oraz *return true*. Poprawia się to poprzez kolejny mały krok. Dodaję do klasy Employee odpowiednie pole, ale zostawiając hardcode’ową wartość:

1. public class Employee {
2. // ...
3. private boolean active = true;
5. public boolean isActive() {
6. return active;
7. }
8. }

Teraz chciałbym całkowicie zlikwidować duplikaty wartości poprzez dodanie prawdziwej zmiennej oraz settera. Także powinienem zainicjalizować pole poprzez wywołanie settera w metodzie przygotowującej dane do testowania. Po dokonaniu zmian test dalej kończy się powodzeniem, a nie mamy teraz żadnego duplikowania wartości. Końcowy kod wygląda następująco:

1. @BeforeEach
2. public void before() {
3. employee = new Employee();
4. employee.setActive(true);
5. // ...
6. }
8. @Test
9. public void testEmployeeActive() {
10. assertTrue(employee.isActive());
11. }
12. public class Employee {
13. // ...
14. private boolean active;
16. public void setActiv(boolean active) {
17. this.active = active;
18. }
20. public boolean isActive() {
21. return active;
22. }
23. }

**4.4 Wprowadzenie zmian**

Czytając opis logiki biznesowej aplikacji zauważam, że pracownik może posiadać kategorię z konkretnie wyróżnionej listy: druga, pierwsza, wyższa. Także pamiętam, że w podpunkcie 4.1 operowałem na pracownikach mających obecną kategorię. Czyli logiczne jest stwierdzenie, że pracownicy bez kategorii też będą występować w aplikacji. Naprasza się tutaj typ wyliczeniowy z czterema wartościami. W mojej sytuacji będę chciał zmienić testy, ponieważ na razie mam kategorię jako *String*. Po dokonaniu niezbędnych zmian oraz po poprawieniu błędów kompilacji wszystkie testy przechodzą pomyślnie, a nowy typ wyliczeniowy oraz jego zastosowanie w klasie *Employee* wygląda następująco:

1. public enum EmployeeCategory {
3. HIGHEST("Wyższa"),
4. FIRST("Pierwsza"),
5. SECOND("Druga"),
6. NONE("Brak");
8. private final String label;
10. EmployeeCategory(String label) {
11. this.label = label;
12. }
14. @Override
15. public String toString() {
16. return label;
17. }
18. }
19. public class Employee {
20. private EmployeeCategory employeeCategory;
21. // ...
22. }

1. K. Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion 2014, s. 10. [↑](#footnote-ref-1)
2. <https://bit.ly/3xRUGCE>, [dostęp: 14.08.2021]. [↑](#footnote-ref-2)
3. Tamże. [↑](#footnote-ref-3)
4. <https://bit.ly/3m94iqK>, [dostęp: 15.08.2021]. [↑](#footnote-ref-4)
5. V. Farcic, A. Garcia, *Test-Driven Java Development*, Packt Publishing Ltd., Birmingham 2015, s. 2. [↑](#footnote-ref-5)
6. <https://bit.ly/3z4S7hW>, [dostęp: 15.08.2021]. [↑](#footnote-ref-6)
7. <https://bit.ly/3bh6sO8>, [dostęp: 27.10.2021] [↑](#footnote-ref-7)
8. <https://bit.ly/3pJeKqn>, [dostęp: 27.10.2021] [↑](#footnote-ref-8)
9. <https://bit.ly/2ZeP7Ty>, [dostęp: 20.10.2021]. [↑](#footnote-ref-9)
10. <https://bit.ly/30Z26tf> [↑](#footnote-ref-10)
11. Rozporządzenie ministerstwa zdrowia Białorusi z dnia 28 maja 2021 r., s. 1. [↑](#footnote-ref-11)