

**Wydział Fizyki i Informatyki Stosowanej**

Projekt dyplomowa

**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**

Ambicją autora 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, twórca pracy spróbuje przetestować program, zanim go napisze. To właśnie ta paradoksalna idea stanowi podstawę metodyki Test-Driven Development (programowanie sterowane testami). Czy to jest nielogiczne? Nie trzeba się śpieszyć 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 głównym wzorcam 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[[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 dewelopera[[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 takie testy 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 przemyślono, dlaczego należy użyć TDD, jakie problemy, spotykające się w poprzednich projektach autora ta metodyka by pokonała. Podczas przygotowywania się do napisania pracy i czytania tematycznej literatury wymieniono kilka bolączek, które przez cały czas nie dawały spokoju oraz które TDD ma na celu wyleczyć:

* większość testów były manualne;
* jeżeli istniały testy automatyczne, to nie wykrywały one rzeczywiste problemy;
* testy automatyczne były pisane i przeprowadzane za późno, żeby wnieść użyteczny wkład w rozwój aplikacji;
* zawsze istniało 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 jakaś funkcjonalność została pominięta podczas przeprowadzenia biednego testowania[[5]](#footnote-5).

Oczywiście, TDD nie pokona magicznie wszystkie te problemy, ale istnieje nadzieja, że zostanie wyciągnięty niezmierzony postęp na tych poziomach tworzenia oprogramowania.

Także na początku wydzielono kilka zasadniczych zalet TDD. Wyszczególniono 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 należy skupić się na interakcji tego źródła z tzw. 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 przedstawiono tłumaczenie najbardziej znaczących zagadnień z ww rozporządzenia. Wiedząc fakt, że testy mogą służyć jako dokumentacja, w rozdziale poświęconym implementacji pokazano jak poniższe urywki najpierw realizują się w testach, a później implementują się w kodzie. Tym zdaniem autor chciał powiedzieć, że czytelnik tej pracy nie powinien się martwić, jeżeli jakieś rzeczy nie będą zrozumiałe na bieżąco.

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

Rozporządzenie określa, że pracownicy sfery medycznej, mające w dniu wejścia rozporządzenia w życie (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 (sekwencyjnie kolejną od aktualnej) 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 nadania drugiej kategorii kwalifikacyjnej a potwierdzenia kategorii kwalifikacji jest obowiązkowa.

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 itd.).

Aby nadać 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 nadania (potwierdzenia) kategorii kwalifikacyjnej, odbywa się w ciągu 5 lat. Warunkiem koniecznym do nadania (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 nadano 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;

1. **Implementacja**

Aplikacja powinna być stworzona przy pomocy programowania sterowanego testami. Oznacza to, że należy zacząć implementację od czystego projektu. Nie ma na razie nic oprócz dokumentu, który stanowi logikę biznesową programu. Czyli wypada iść z implementacją krok po kroku, czasami te kroki będą małe, czasami duże. Można pisać testy w taki sposób, że każdy z nich będzie wymagał dodania jedynej linii kodu i niewielkiej refaktoryzacji. Z drugiej strony można pisać testy tak skomplikowane, że będą wymagać dopisania setek linii kodu i godziennej refaktoryzacji. Nie ma lepszego podejścia z tych dwóch. W TDD powinno się 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 się zdobyć pewność siebie i wszystko będzie szło gładko, będzie można spróbować przejść na większą prędkość, czyli skorzystać z patternu „*Obvious Implementation*” – testy będą wymagały napisania większego kawałku kodu, jeżeli od razu wiadomo jak ten kod ma wyglądać. Ale zawsze należy wracać do mniejszych kroków, jeśli dostanie się głupi błąd tam, gdzie implementacja wydawała się być oczywista.

Cztery kolejne podrozdziały zostały napisane równolegle z napisaniem kodu źródłowego aplikacji, częściowo użytego w listingach tej części pracy. W ten sposób autor chciał pokazać rzeczywiste użycie TDD. Gdy dostaje się pewien problem (funkcjonalność do zaimplementowania), który należy rozwiązać od zera, dzieląc na podproblemy.

**4.1 Podstawa modelu**

Nazwano pierwszy krok implementacji programu jako „podstawa modelu”. Ponieważ tu zostało pokazane powstanie kilku podstawowych klas modelu aplikacji. Także zademonstrowano dodanie do tych klas wymaganych do przejścia testów pól oraz metod. Powinno to dać impuls do dalszego rozwoju logiki programu. TDD zakłada, że powinno się pisać tylko ten kod, którego wymaga test. Czyli daje 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 należy ustrukturyzować to jako listę podpunktów:

* + - 1. Jest data 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.

**Pasowałoby najpierw zająć się podpunktem 2, który wymusza na stworzenie pracownika zawierającego kategorię.** To zdanie brzmi jak test, przedstawiony na listingu 1.

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

Listing 1: Metoda testowa oznaczona adnotacją frameworku testowego JUnit

Teraz na tak prostym przykładzie zostanie pokazane jak działa pattern „*Assert first*”. Czyli najpierw dodaje się do testu metodę statyczną, sprawdzającą kategorię pracownika. Zostało pokazane to na listingu 2. I właśnie w tym momencie wymyśla się nazwy poszczególnych klas i pól.

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

Listing 2: Do metody testowej została dołączona metoda statyczna, porównująca dwie wartości

Następnie przy pomocy podpowiedzi kompilatora dodaje się nieistniejący obiekt, co widać na  
listingu 3.

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

Listing 3: W metodzie testowej został stworzony obiekt klasy Employee

Napisany właśnie test nawet się nie kompiluje, ale łatwo to poprawić. Najprościej to zrobić na sztywno dodając brakujące rzeczy, czyli klasę Employee, która zawiera pole category (patrz: lis. 4).

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

Listing 4: Klasa Employee, zawierająca publiczne pole typu *String* – category

Widać, że jest źle – istnieje pole publiczne w klasie. Ale teraz test się kompiluje i można zobaczyć jeden z kroków TDD w akcji. Napisano test, który nie przechodzi. Przyczynę można zobaczyć na listingu 5.

org.opentest4j.AssertionFailedError:

Expected :null

Actual :Kategoria X

Listing 5: Przyczyna nieudania się testu oraz oczekiwany i rzeczywisty parametry  
metody statycznej assertEquals

Najważniejsze jest, że istnieje postęp. Nieudany test to też wynik. Teraz należy oderwać się od globalnego celu „odbycie certyfikacji pracownikiem w terminie” i przełączyć się na mały kroczek z kategorią. Chce się zmusić ten test na przejście, a nie ma teraz na celu dostania pięknego rozwiązania. Najmniejsza zmiana, która sprawi, że test przebiegnie pomyślnie, została pokazana na listingu 6. Takie podejście jest jednym z wzorców TDD i nazywa się „Sfałszuj to” (ang. *Fake It*). Czyli chcemy zwracać stałą i stopniowo zamieniać ję na zmienną, dopóki nie otrzyma się prawdziwy kod.

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

Listing 6: Publiczne pole category w klasie Employee zostało zainicjalizowane

Ale jak można teraz zauważyć, istnieje duplikowanie pomiędzy testem a kodem realizacji. Nie pozwoli to napisać kolejny test i go przejść, jeżeli parametry wejściowe będą inne (np. inna kategoria). Można naprawić to w oczywisty sposób. Także należy pamiętać, że powinno się teraz poruszać małymi krokami, żeby nauczyć się takiego tempu i odczuć rytm TDD. Duplikowanie usuwa się poprzez dodanie konstruktora z odpowiednim parametrem, także można pozwolić sobie zrobić pole klasy prywatnym i wygenerować getter. Dodany kod został zademonstrowany na listingu 7.

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. }

Listing 7: Dodanie konstruktora z parametrem category, gettera do tego pola,  
które teraz już jest prywatne

Także należy wnieść zmiany do testu, co pokazane na listingu 8.

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

Listing 8: Obiekt teraz jest tworzony za pomocą konstruktora z parametrem, wartość pola category pobiera się pomocą gettera

Test dalej przechodzi pomyślnie, kod wygląda już lepiej. Można teraz przejść do kolejnych podpunktów problemu. Na przykład do podpunktu 1, dla przypomnienia brzmi on tak:

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

Należy gdzieś przechowywać tę datę, ponieważ jest to data wejścia rozporządzenia w życie i w przyszłości będzie potrzeba nie raz z niej skorzystać. Wybrano sposób przechowywania jako konstantę w już instniejącej klasie, pokazano to na listingu 9.

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

Listing 9: Publiczne niezmienne pole statyczne typu *LocalDate*, znajdujące się w klasie Employee i przechowujące datę wejścia rozporządzenia w życie

Dalej można przejść do reszty 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 wiadowo, 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 istnieje potrzeba informowania użytkownika o zbliżającym się ostatecznym terminie (ang. *deadlinie*) dla potwierdzenia kategorii pracownika. Zakłada się, że użytkownik będzie chciał samodzielnie wybierać preferowany termin wcześniejszego powiadomienia w dniach. Można pozwolić sobie trochę przyspieszyć i napisać większy test, pokazany na listingu 10. W tym teście został stworzony nowy obiekt typu NotificationTerm, oraz zostało sprawdzone, czy ilość dni między datami jest zgodna z wartością pola days obiektu notificationTerm. Jest to test jednostkowy, nie sprawdza się integracja pomiędzy warstwami aplikacji.

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. }

Listing 10: Test sprawdzający, czy ilość dni między datami jest zgodna  
z obiektem notificationTerm

Napisany właśnie test skończył się powodzeniem po dopisaniu odpowiednich klas, pól i metod nawet bez dodawania żadnych hardcode’owych wartości, ponieważ wszystko się inicjalizuje w miejscu. Właśnie takie podejście jest jednym z wzorców TDD i nazywa się „Oczywista implementacja” (ang. *obvious implementation*).

Teraz można byłoby przetestować klasę NotificationTerm, jako odwzorowanie tabeli w bazie danych. Będzie to test integracyjny z warstwą bazodanową, pasowało by wynieść ten temat w osobny podrozdział.

**4.2 Test integracyjny – warstwa danych**

W testach integracyjnych z warstwą danych należy skupić się na interakcji pomiędzy klasą-repozytorium a źródłem danych. Stworzono nową klasę testową, która jest skonfigurowana na potrzeby testów integracyjnych z warswą danych. Różnicą pomiędzy testami jednostkowymi a integracyjnymi jest to, że teraz obiekt klasy NotificationTerm wysyła się do bazy danych. A następnie tworzy się nowy obiekt, który inicjalizuje się znalezionym na podstawie id rekordem z bazy. W taki sposób sprawdza się integracja z warstwą danych. Kod klasy jest pokazany na listingu 11.

1. @RunWith(SpringRunner.class)
2. @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
3. @DataJpaTest
4. @ActiveProfiles("test")
5. public class DAOIntegrationTests {
7. @Autowired
8. private TestEntityManager entityManager;
10. @Test
11. public void testNotificationTermDAO() {
12. NotificationTerm notificationTerm = new NotificationTerm();
13. notificationTerm.setDays(365);
14. entityManager.persist(notificationTerm);
15. entityManager.flush();
16. NotificationTerm foundNotificationTerm = entityManager
17. .find(NotificationTerm.class, notificationTerm.getId());
18. assertEquals(365, foundNotificationTerm.getDays());
19. }

Listing 11: Klasa testowa skonfigurowana na potrzeby testów integracyjnych z bazą danych

W linijkach 2 oraz 3 listingu 11 dodałem adnotacje, które mówią klasie testowej, że będą to testy integracyjne z warstwą danych oraz, że chcę komunikować nie z wbudowanym, tylko z rzeczywistem źródłem danych. Także dobrą funkcjonalnością tak skonfigurowanej klasy testowej jest to, że zapisujące się podczas testów dane do bazy zostaną usunięte po zakończeniu testów. Czyli schemat bazy danych wróci do poprzedniego stanu. Ale zdecydowano skonfigurować inne źródło danych na potrzeby testów. A żeby klasa testowa wiedziała, że należy skorzystać z bazy testowej, dodano odpowiednią adnotację w linijce 4.

Także wywołuje się metodę flush w linijce 16, która powoduję rzeczywiste wykonanie zapytania w bazie. Pozwala 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, co jest pokazane na listingu 12.

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. }

Listing 12: Kod klasy NotificationTerm, która jest encją odwzorowania  
obiektowo-relacyjnego (skrót ang. *ORM*)

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

**4.3 Rozszerzenie modelu**

Z wymagań aplikacji pośrednio wynika fakt, że pracownik może być aktywny lub nieaktywny. Zatem pasuje dodać kolejny test (patrz: lis. 13).

1. @Test
2. public void testEmployeeActive() {
3. Employee employee = new Employee();
4. employee.setActive(true);
5. assertTrue(employee.isActive());
6. }

Listing 13: Metoda testowa, sprawdzająca, czy pracownik jest aktywny

Teraz na tak prostym przykładzie zostanie pokazane zastosowanie kolejnego wzorcu TDD „Triangulacja” (ang. *Triangulate*). Jak wynika z teorii, najpierw należy zastosować hardcode’ową wtyczkę, zwracającą stalą, czyli skorzystać z wzorcu fałszywej implementacji (patrz: lis. 14).

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

Listing 14: Metoda isActive w klasie Employee zwracająca stałą

Test przechodzi pomyślnie. Ale co się stanie, jak zostanie napisany kolejny test, pokazany na listingu 15. Także na listingu 15 zostały dodane metody przygotowywujące dane do testowania oraz czyszczące, ponieważ te dane będą się często powtarzać dla różnych testów.

1. private Employee employee;
2. @BeforeEach
3. public void before() {
4. employee = new Employee();
5. }
6. @AfterEach
7. public void after() {
8. employee = null;
9. }
11. @Test
12. public void testEmployeeActive() {
13. employee.setActive(true);
14. assertTrue(employee.isActive());
15. }
16. @Test
17. public void testEmployeeNotActive() {
18. employee.setActive(false);
19. assertFalse(employee.isActive());
20. }

Listing 15: Metody before oraz after, a także dwie metody testowe, sprawdzające,  
czy pracownik jest aktywny

Jeden z testów z listingu 15 przechodzi pomyślnie, a drugi nie. Przyczyną jest to, że metoda isActive zwraca teraz stałą true. Napisano drugi test, który inicjalizuje pole active wartością przeciwną, czyli false. W taki sposób udało się podejść do przypadku testowego z drugiej strony, czyli skorzystać z triangulacji. Pozwoliło to zobaczyć, że klasa Employee ma teraz wadę – metodę zwracającą stałą. Można to w bardzo prosty sposób zlikwidować, co zostało zademonstrowne na listingu 16.

1. public class Employee {
2. // ...
3. private boolean active;
5. public void setActive(boolean active) {
6. this.active = active;
7. }
9. public boolean isActive() {
10. return active;
11. }
12. }

Listing 16: Część kodu klasy Employee, zawierająca prywatne pole active oraz  
odpowiedni getter oraz setter

Właśnie w tak prostym przykładzie było tylko dwie wartości graniczne – true oraz false. Natomiast zademonstrowany wzorzec „*Triangulate*” jest bardzo przydatny w sytuacjach, gdy logika jest bardzo skomplikowana. W takich przypadkach pasuje triangulować problem z kilku stron, żeby złapać sens i napisać poprawny kod.

**4.4 Wprowadzenie zmian**

Czytając opis logiki biznesowej aplikacji zauważono, że pracownik może posiadać kategorię z konkretnie wyróżnionej listy: druga, pierwsza, wyższa. Także należy pamiętać, że w podpunkcie 4.1 operowało się 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. Także trzeba będzie zmienić kod testów, ponieważ na razie kategoria jest typu String. Po dokonaniu niezbędnych zmian oraz po poprawieniu błędów kompilacji wszystkie testy przechodzą pomyślnie, a nowy typ wyliczeniowy oraz jego zastosowanie w klasie Employeewidać na listingu 17.

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

Listing 17: Typ wyliczeniowy Category oraz jego zastosowanie w klasie Employee

**4.5 Bezpieczeństwo aplikacji**

Na chwilę odchodząc od tematu TDD, autor chciałby zademonstrować jak jest skonstruowana warstwa bezpieczeństwa aplikacji. Nie jest to ani bezpośrednim tematem pracy, ani bardzo ważnym wymogiem programu. Ale twórca pracy zdecydował zaimplementować to w celu wykorzystania w dalszych etapach rozwoju aplikacji, kiedy powstanie potrzeba rozdzielenia użytkowników na role oraz rozbicia schematu na więcej niż jedną, korzystającą z aplikacji, firmę medyczną.

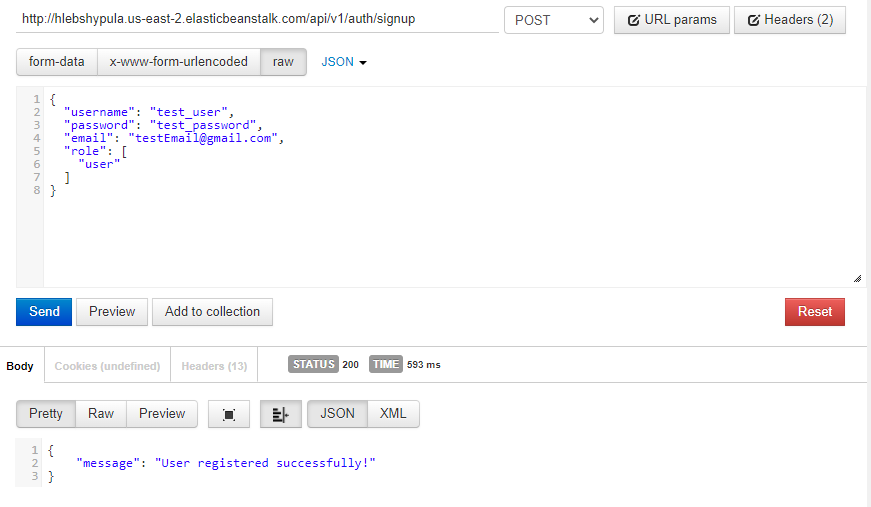
Cześć serwerowa polega na uwierzytelnianiu Spring Boot JWT (JSON Web Token) przez Spring Security co zapewnia uwierzytelnianie oparte na tokenach i autoryzację opartą na rolach. Także została zaimplementowana pełna walidacja wartości wejściowych przy rejestracji.

Ta warstwa aplikacji została przetestowana za pomocą narzędzia do wysyłania REST zapytań Postman. Na rysunku 1 została pokazana odpowiedź serwera przy próbie rejestracji użytkownika z błędnymi danymi wejściowymi, wysyłając zapytanie POST do punktu końcowego (and. endpoint) /api/v1/auth/signup. W tym przypadku polem z błędną wartością jest nazwa użytkownika. Dostaje się systemową wiadomość, że rozmiar wartości nazwy użytkownika musi być w przedziale od 5 do 32 znaków. Wiadomość jest w języku rosyjskim, ponieważ jest generowana automatycznie w języku systemu.



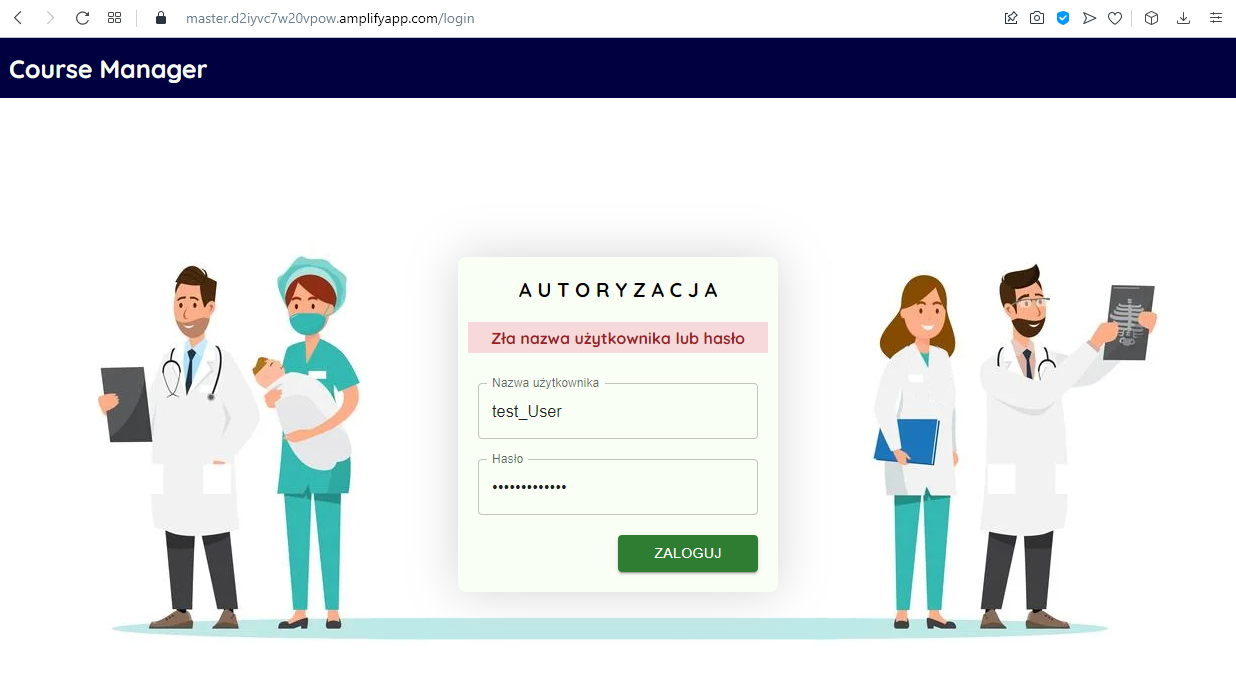
Rysunek 1: Odpowiedź serwera przy próbie rejestracji użytkownika z błędną nazwą użytkownika

Natomiast przy rejestracji z poprawnymi danymi dostaje się wadomość o udanej rejestracji użytkownika, co jest pokazane na rysunku 2.



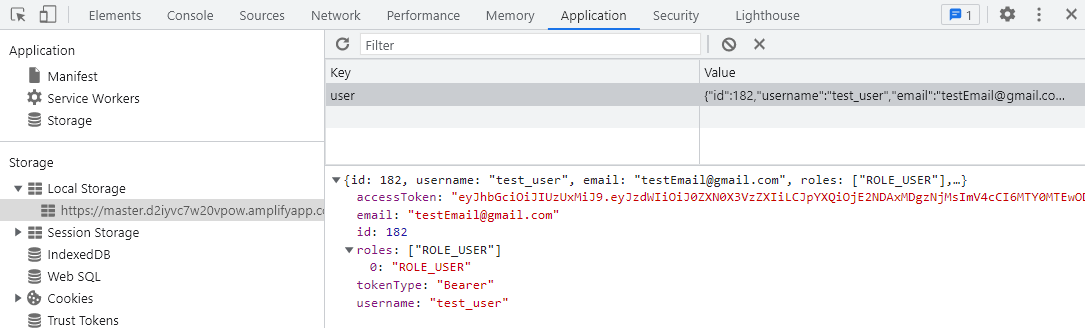
Rysunek 2: Poprawna rejestracja użytkownika

Po stronie klienta zostało zaimplementowane okno logowania (patrz: rys. 3). Przy próbie wejścia na stronę z błędnymi danymi użytkownik dostaje komunikat o błędzie.

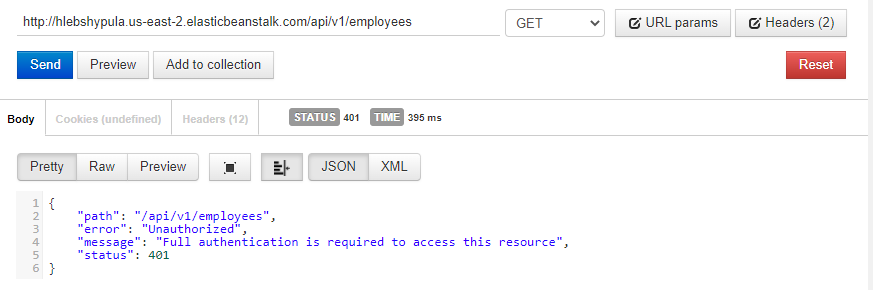


Rysunek 3: Okno logowania wraz z komunikatem o błędzie

Po poprawnym logowaniu, serwer wysyła w odpowiedzi dane użytkownika, które są zapisywane do *Local Storage* przeglądarki w odpowiedniej domenie (patrz: rys. 4). Jednym z pól zwróconych w odpowiedzi jest token JWT, który przechowuje w siebie zahaszowane dane i służy do uwierzytelniania użytkownika podczas zapytań do punktów końcowych, które są skonfigurowane na przyjęcie zapytań tylko od zautoryzowanych użytkowników lub od użytkowników określonych roli. Token załącza się do zapytania jako nagłówek. Dalej po stronie serwera przy otrzymaniu zapytania odbywa się sprawdzenie poprawności tokenu w specjalnie napisanej klasie tzw. filterze. Przy próbie zapytania z błędnym tokenem, filter to rozpozna i rzuci odpowiedni wyjątek (patrz: rys. 5).



Rysunek 4: Dane użytkownika zapisane do *Local Storage* przeglądarki



Rysunek 5: Odpowiedź serwera przy próbie zapytania z błędnym tokenem

Kod źródłowy żadnego kawałku warstwy bezpieczeństwa nie został zademonstrowany, ponieważ jest dość skomplikowany, a zgodnie z autorem nie jest to bezpośrednim tematem niniejszej pracy.

**4.6 Edycja danych**

Podczas napisania pierwszych testów związanych z edycją danych pracownika, napotkano problem. Przy ścisłej walidacji pól klasy Employee (patrz: lis. 18), znika możliwość częściowo edytować stan obiektu.

1. @Entity
2. @CategoryDatesNotNull
3. public class Employee {
5. @Id
6. @GeneratedValue(strategy = GenerationType.AUTO)
7. @Column(columnDefinition = "serial")
8. Long id;
10. /// Main info
11. /// Id from accounting app
12. @NotNull(message = "foreignId cannot be null")
13. Long foreignId;
14. @NotBlank(message = "fullName cannot be blank")
15. private String fullName;
16. @NotNull(message = "hiringDate cannot be null")
17. private LocalDate hiringDate;
18. @NotBlank(message = "jobFacility cannot be blank")
19. private String jobFacility;
20. @NotBlank(message = "position cannot be blank")
21. private String position;
23. /// Category
24. private String qualification;
25. private Category category;
26. private String categoryNumber;
27. private LocalDate categoryAssignmentDate;
28. private LocalDate categoryAssignmentDeadlineDate;
29. private LocalDate docsSubmitDeadlineDate;
30. private LocalDate categoryPossiblePromotionDate;
32. /// Courses
33. @OneToMany(mappedBy = "employee", fetch = FetchType.EAGER)
34. @JsonIgnore
35. private Set<Course> courses;
36. private int courseHoursSum;
38. /// Exemption
39. private Exemption exemption;
40. private LocalDate exemptionStartDate;
41. private LocalDate exemptionEndDate;
42. private boolean exemptioned;
44. /// Active / inactive
45. private boolean active;
47. /// Education
48. private Education education;
49. private String eduName;
50. private LocalDate eduGraduationDate;
52. /// statyczne pola, konstruktory, gettery, settery ...
54. }

Listing 18: Pola klasy Employee oznaczone adnotacjami walidacji i frameworku Hibernate

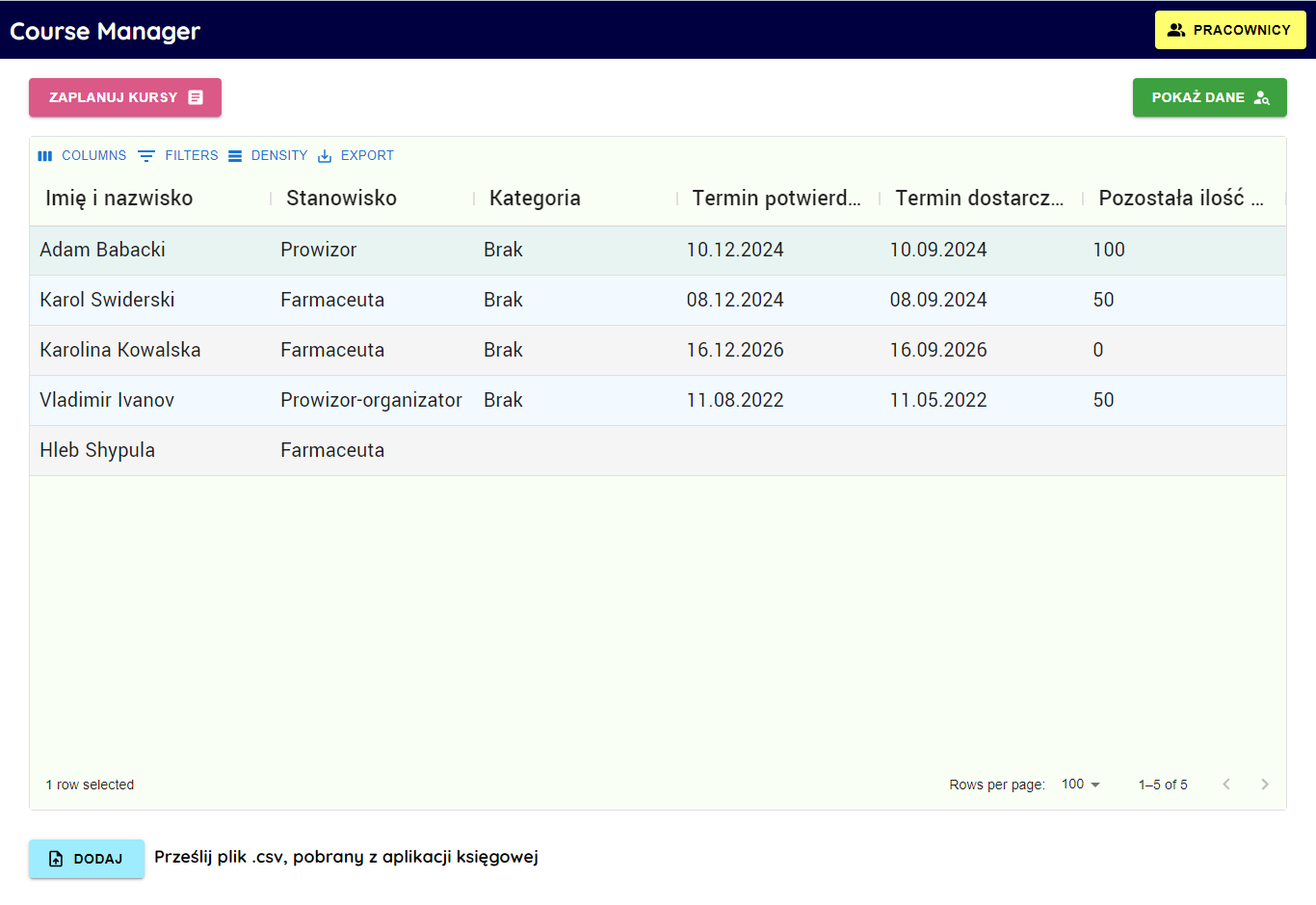
Jak widać na listingu 18, adnotacjami walidacji (@NotNull, @NotBlank) oznaczone są tylko pola w sekcji Main info. Ponieważ te pola są inicjalizowane na początku życia obiektu i są z nim związane przez cały cykl życia. Wartości tych pól przychodzą z zewnątrz (z aplikacji księgowej) i na ich podstawie tworzy się nowy obiekt pracownika. Natomiast pola z innych sekcji (Category, Exemption itd.) na początku powinny mieć wartość null. Użytkownik aplikacji powinien mieć możliwość osobno edytować każdą z tych sekcji stanu obiektu.

Typowym sposobem edycji stanu obiektu w aplikacjach internetowych opartych na REST API jest wysyłanie zapytania PUT, zawierającego w ciele nowy stan obiektu, który nadpisuje stan poprzedni.

Teraz problem jest taki, że, korzystając z zapytań PUT, nie ma możliwości oznaczyć jakieś pole z sekcji Category adnotacją @NotNull, ponieważ przy próbie edycji sekcji Exemption nadal następuje nadpisanie wszystkich pól obiektu. Na przykład jeżeli pole jest null i nie ma potrzeby zmiany wartości, to wartość nadal się zmieni z null na null. A to właśnie wywoła naruszenie ograniczenia, które nakłada się na pole poprzez adnotację @NotNull.

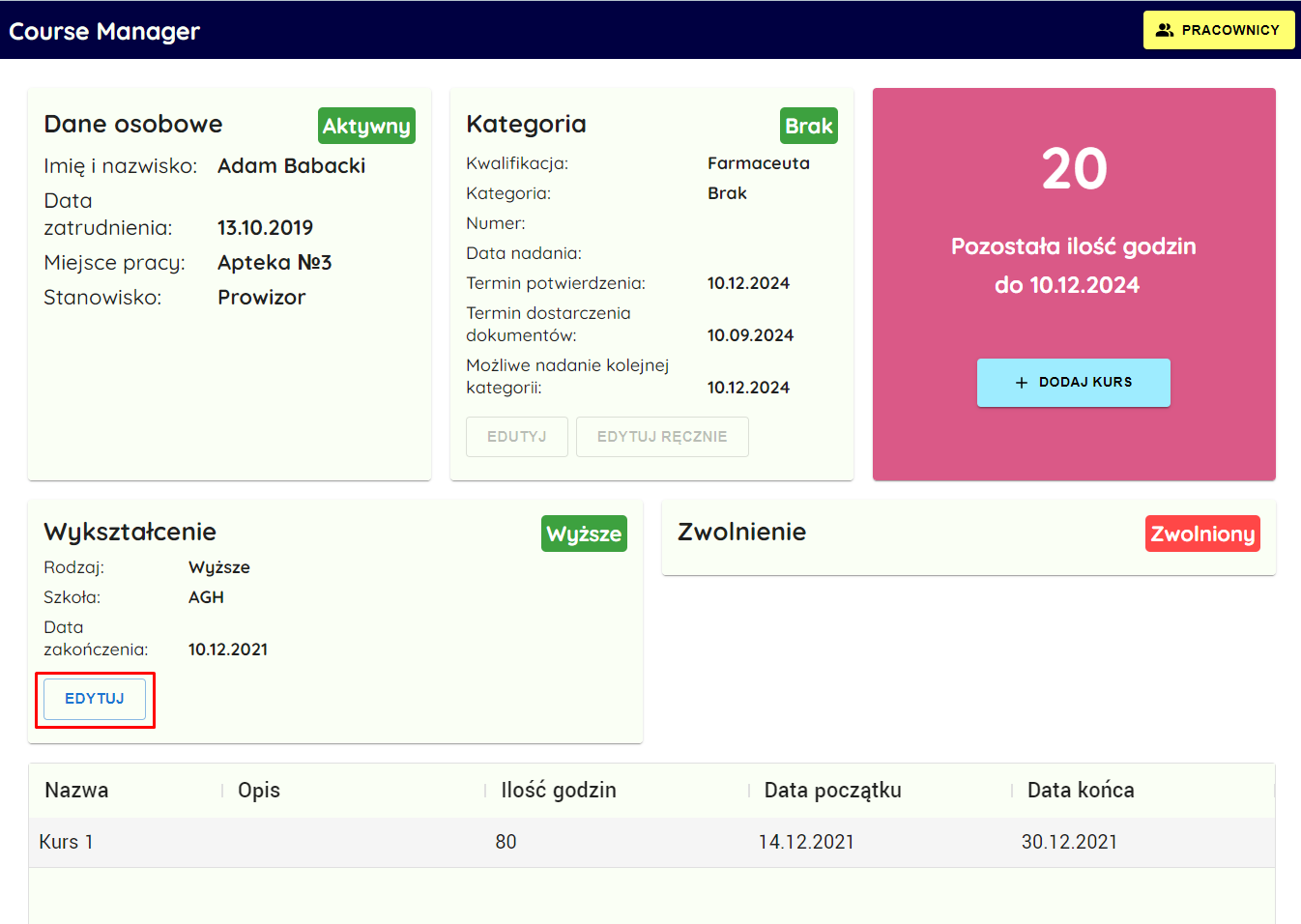
Przeglądając wzorce rozwiązujące ten problem, zdecydowano skorzystać z modyfikowanego wzorcu, polegającego na obiekcie transferu danych (ang. *Data Transfer Object, DTO*). Polega on na tym, że do częściowej modyfikacji stanu obiektu używa się zapytanie PATCH, które w ciele zawiera DTO dla odpowiedniej sekcji obiektu klasy Employee. W dalszej części podpunktu przebieg działań zostanie pokazany na jednym z pięciu przykładów użycia tego podejścia w aplikacji.

Chcąc zmodyfikować wykształcenie pracownika, użytkownik powinien wybrać osobę z listy (patrz: rys. 6) i wejść na stronę z danymi.



Rysunek 6: Strona z listą wszystkich pracowników

Na stronie z danymi użytkownik powinien kliknąć na przycisk „Edytuj” na kartce z  
wykształceniem. Ten przycisk jest oznaczony czerwonym kwadracikiem na rysunku 7.



Rysunek 7: Strona z danymi pracownika oraz oznaczony na czerwono  
przycisk do edycji wykształcenia

Następnie użytkownik powinien wprowadzić dane w formie oraz kliknąć na przycisk  
„Zatwierdź” (patrz: rys. 8).



Rysunek 8: Forma do edycji wykształcenia pracownika

W tym momencie ze strony klienta na serwer wysyła się zapytanie PATCH do punktu końcowego /api/v1/employees/{id}/education poprzez wbudowane Fetch API. Kod źródłowy funkcji wysyłającej zapytanie jest umieszczony na listingu 19.

1. export const patchEmployeeEducation = (id, patch) => {
2. return patchEmployee(id, patch, "/education");
3. };
5. const patchEmployee = (id, patch, path) => {
6. return fetch(API\_BASE\_URL + "/" + id + path, {
7. method: "PATCH",
8. body: JSON.stringify(patch),
9. headers: Object.assign(
10. {},
11. { "Content-type": "application/merge-patch+json" },
12. authHeader()
13. ),
14. })
15. .then((response) => {
16. if (response.ok) {
17. return response.json();
18. } else {
19. throw new Error(response.clone().json());
20. }
21. })
22. .catch((error) => {
23. console.log(error);
24. });
25. };

Listing 19: Funkcje odpowiadające za wysyłanie zapytania PATCH  
w celu edycji wykształcenia pracownika

Po stronie serwera zapytanie przyjmuje się w metodzie, pokazanej na listingu 20.

1. @PatchMapping(path = "/{id}/education", consumes = "application/merge-patch+json")
2. public ResponseEntity<Employee> patchEmployee(@PathVariable Long id,
3. @RequestBody @Valid EmployeeEducationPatchDto employeeEducationPatchDto) {
4. return ResponseEntity.ok(employeeDataService.patch(id, employeeEducationPatchDto));
5. }

Listing 20: Metoda w klasie REST kontrolera, przyjmująca zapytanie PATCH

Jak widać na listingu 20, wywołuje się metoda patch serwisu[[12]](#footnote-12) EmployeeDataService, która przyjmuje jako parametry id pracownika oraz DTO wykształcenia (patrz: lis. 22). DTO wykształcenia zawiera tylko te pola z klasy Employee, które odpowiadają za sekcję Education z listingu 18. Także na tym etapie można dodać adnotacje, odpowiadające za walidacje tych pól. Teraz już to nie przeszkodzi innym sekcjom klasy Employee, ponieważ jest wyosobnione w oddzielną klasę. A żeby zwalidować obiekt, należy oznaczyć go adnotacją @Valid, jak to jest zrobione na listingu 20 w 3 linijce. Kod źrodłowy DTO jest pokazany na listingu 21.

1. public class EmployeeEducationPatchDto extends EmployeePatchDto {
3. @NotNull(message = "Education cannot be null")
4. private Education education;
5. @NotBlank(message = "Education name cannot be blank")
6. private String eduName;
7. @NotNull(message = "Education graduation date cannot be empty")
8. private LocalDate eduGraduationDate;
9. /// getters, setters
10. }

Listing 21: Kod DTO EmployeeEducationPatchDto wraz z adnotacjami walidacji

1. private void patchEmployeeEducation(Employee employee, EmployeePatchDto employeePatchDto) {
2. BeanUtils.copyProperties(employeePatchDto, employee);
3. }

Listing 22: Kod metody patch serwisu EmployeeDataService, odpowiadający za kopiowanie wartości DTO do obiektu klasy Employee

W taki, opisany powyżej, sposób osiągnięta została częściowa edycja stanu obiektu klasy Employee nie tracąc mocy walidacji. Zademonstrowano to na prostszym przykładzie, żeby nie przeładowywać tekst pracy kodem źródłowym. Ale warto powiedzieć, że na przykład DTO dla sekcji Category z listingu 18 zawiera na poziomie klasy dwie własnoręcznie napisane adnotacje walidacji, ponieważ logika tej sekcji jest dość skomplikowana i systemowe adnotacje z nią nie radziły.

Natomiast testy do tego kawałku logiki, które były napisane zgodnie z założeniami TDD przed napisaniem rzeczywistego kodu, są pokazane na listingach 23 oraz 24.

Pierwszy test na listingu 23 od linijki 25 sprawdza, że ograniczenia na pewno nie są naruszone przy walidacji poprawnego DTO EmployeeEducationPatchDto. Natomiast drugi test od linii 32 sprawdza, czy naruszenie nałożone adnotacją @NotNull na pole education z listingu 21 na pewno zostało złapane. Podczas testów walidacja przeprowadza się ręcznie poprzez wywołanie walidatora wstrzykniętego w linijce 7 za pomozą adnotacji @Autowired. Natomiast w rzeczywistym użyciu obiektów walidacja odbywa się automatycznie po oznaczeniu obiektu adnotacją @Valid, jak już było powiedziane wyżej.

1. /// Unit tests
2. @SpringBootTest
3. @ActiveProfiles("test")
4. public class EmployeeEducationPatchDtoTests {
6. @Autowired
7. Validator validator;
9. private EmployeeEducationPatchDto employeeEducationPatchDto;
11. @BeforeEach
12. public void before() {
13. employeeEducationPatchDto = new EmployeeEducationPatchDto();
14. employeeEducationPatchDto.setEducation(Education.HIGHER);
15. employeeEducationPatchDto.setEduName("AGH");
16. employeeEducationPatchDto.setEduGraduationDate(LocalDate.of(2020, 5, 5));
17. }
19. @AfterEach
20. public void after() {
21. employeeEducationPatchDto = null;
22. }
24. @Test
25. public void When\_Valid\_Then\_ViolationsEmpty() {
26. Set<ConstraintViolation<EmployeeEducationPatchDto>> violations = validator.validate(employeeEducationPatchDto);
28. assertTrue(violations.isEmpty());
29. }
31. @Test
32. public void When\_EducationIsNull\_Should\_ThrowConstraintViolationException() {
33. employeeEducationPatchDto.setEducation(null);
35. Set<ConstraintViolation<EmployeeEducationPatchDto>> violations = validator.validate(employeeEducationPatchDto);
37. assertEquals("Education cannot be null", violations.iterator().next().getMessage());
38. }
39. /// inne testy
40. }

Listing 23: Testy jednostkowe dla DTO EmployeeEducationPatchDto

1. /// Unit tests
2. @ExtendWith(MockitoExtension.class)
3. @SpringBootTest
4. @ActiveProfiles("test")
5. public class EmployeeDataServiceTests {
7. @Mock
8. EmployeeRepository employeeRepository;
9. @Mock
10. CourseService courseService;
11. @Mock
12. EmployeeValidationService employeeValidationService;
13. @Mock
14. EmployeeExemptionService employeeExemptionService;
15. @Mock
16. EmployeeFilteringService employeeFilteringService;
17. @Mock
18. EmployeeCategoryService employeeCategoryService;
20. private EmployeeDataService employeeDataService;
22. private Employee employee;
24. @BeforeEach
25. public void before() {
26. employeeDataService = new EmployeeDataService(employeeRepository, courseService, employeeValidationService,
27. employeeExemptionService, employeeFilteringService, employeeCategoryService);
29. employee = new Employee();
30. employee.setForeignId(1L);
31. employee.setFullName("Test");
32. employee.setHiringDate(LocalDate.of(2020, 5, 5));
33. employee.setJobFacility("Apteka 9");
34. employee.setPosition("Farmaceuta");
36. Optional<Employee> optionalEmployee = Optional.of(employee);
37. doReturn(optionalEmployee).when(employeeRepository).findById(any(Long.class));
38. }
40. @AfterEach
41. public void after() {
42. employee = null;
43. }
45. @Test
46. public void Should\_PatchEducation() {
47. EmployeeEducationPatchDto employeeEducationPatchDto = new EmployeeEducationPatchDto();
48. employeeEducationPatchDto.setEducation(Education.HIGHER);
49. employeeEducationPatchDto.setEduName("AGH");
50. employeeEducationPatchDto.setEduGraduationDate(LocalDate.of(2020, 5, 5));
52. BeanUtils.copyProperties(employeeEducationPatchDto, employee);
53. doReturn(employee).when(employeeRepository).save(any(Employee.class));
55. Employee newEmployee = employeeDataService.patch(any(Long.class), employeeEducationPatchDto);
57. assertNotNull(newEmployee);
58. assertEquals("AGH", newEmployee.getEduName());
59. }
60. }

Listing 24: Testy jedostkowe dla serwisu EmployeeDataService

Na listingu 24 widać wykorzystanie obiektów fikcyjnych oznaczonych adnotacją @Mock frameworku Mockito. W linijce 26 tworzy się nowy obiekt serwisu poprzez konstruktor, gdzie wprowadza się te obiekty. Oznacza to, że przeprowadzane w tej klasie testy są testami jednostkowymi, a nie integracyjnymi, które by sprawdzały integrację tego serwisu z innymi warstwami aplikacji. Po oznaczeniu obiektu adnotacją @Mock można zdefiniować zachowanie tego obiektu w pewnej sytuacji. W linijkach 37 oraz 53 zdefiniowano, co ma się zwrócić po wywołaniu odpowiedniej metody podczas wykonania testu.

Ale żeby móc skorzystać z takiego typu fikcyjnych obiektów, należy w klasie, do której planuje się wstawić te obiekty, użyć wstrzykiwania zależności poprzez konstruktor[[13]](#footnote-13). W taki sposób zwiększa się hermetyzacja klasy, a także pojawia się możliwość użyć fikcyjnych obiektów podczas testowania, jak było pokazane powyżej. Ta technika jest pokazana na listingu 25, który zawiera kawałek kodu źródłowego serwisu EmployeeDataService. W linijce 12 znajduje się konstruktor oznaczony adnotacją @Autowired, który inicjalizuje zależności klasy na podstawie parametrów.

1. @Service
2. public class EmployeeDataService {
4. private final EmployeeRepository employeeRepository;
5. private final CourseService courseService;
6. private final EmployeeValidationService employeeValidationService;
7. private final EmployeeExemptionService employeeExemptionService;
8. private final EmployeeFilteringService employeeFilteringService;
9. private final EmployeeCategoryService employeeCategoryService;
11. @Autowired
12. public EmployeeDataService(EmployeeRepository employeeRepository, CourseService courseService,
13. EmployeeValidationService employeeValidationService, EmployeeExemptionService employeeExemptionService,
14. EmployeeFilteringService employeeFilteringService, EmployeeCategoryService employeeCategoryService) {
15. this.employeeRepository = employeeRepository;
16. this.courseService = courseService;
17. this.employeeValidationService = employeeValidationService;
18. this.employeeExemptionService = employeeExemptionService;
19. this.employeeFilteringService = employeeFilteringService;
20. this.employeeCategoryService = employeeCategoryService;
21. }
22. /// inne metody
23. }

Listing 25: Kawałek kodu serwisu EmployeeDataService, demonstrujący technikę  
wstrzykiwania zależności poprzez konstruktor

**4.7 Design aplikacji**

Przeprowadzając niezmierzoną ilość refaktoryzacji na początku tworzenia aplikacji, zauważono pełną regularność w relacjach między komponentami programu. Zastanawiając się nad tym tematem, autor doszedł do pomysłu, że design aplikacji – to abstrakcyjna piramida. Zaczynając od małych, ale nie mniej ważnych elementów logiki biznesowej, szkielet aplikacji rozwija się do dużych fragmentów, które łączą mniejsze kawałki i rządzą tą logiką.

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/3sust5M>, [dostęp: 20.12.2021] [↑](#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)
12. Serwis – klasa oznaczona adnotacją @Service, która implementuje logikę biznesową aplikacji. [↑](#footnote-ref-12)
13. Wstrzykiwanie zależności poprzez konstruktor – technika wstrzykiwania zależności, oznaczająca, że przekazanie wymaganych komponentów to klasy odbywa się w momencie tworzenia instancji. Osiąga się poprzez oznaczenie konstruktora adnotacją @Autowired. W przeciwieństwie do wstrzykiwania zależności poprzez pole, gdzie zależności są przypisywane bezpośrednio do pól, które są oznaczone adnotacją @Autowired. [↑](#footnote-ref-13)