

Radosław Smilgin

# ZAWÓD

01010001101010\_0110100110011101

# TESTER

Od decyzji do pierwszych kroków w pracy



Radosław Smilgin

**ZAWÓD**

**TESTER**

Od decyzji do pierwszych kroków w pracy



Projekt okładki i stron tytułowych **Grzegorz Laskowski**

Rysunki **Norbert Prochacki**

Wydawca **Łukasz Łopuszański**

Redaktor prowadzący **Jolanta Kowalczuk**

Redaktor **Joanna Cierkońska**

Skład wersji elektronicznej na zlecenie Wydawnictwa Naukowego PWN **Michał Nakoneczny / 88em.eu**

Zastrzeżonych nazw firm i produktów użyto w książce wyłącznie w celu identyfikacji.

Książka, którą nabyłeś, jest dziełem twórcy i wydawcy. Prosimy, abyś przestrzegał praw, jakie im przysługują. Jej zawartość możesz udostępnić nieodpłatnie osobom bliskim lub osobiście znanym. Ale nie publikuj jej w internecie. Jeśli cytujesz jej fragmenty, nie zmieniaj ich treści i koniecznie zaznacz, czyje to dzieło. A kopiując jej część, rób to jedynie na użytek osobisty.

Szanujmy cudzą własność i prawo

Więcej na [www.legalnakultura.pl](http://www.legalnakultura.pl)

*Polska Izba Książki*

eBook został przygotowany na podstawie wydania papierowego z 2016 r., (wyd. I)

Warszawa 2016

ISBN 978-83-01-18571-8

Wydanie I

Wydawnictwo Naukowe PWN SA

02-460 Warszawa, ul. Gottlieba Daimlera 2

tel. 22 69 54 321, faks 22 69 54 288

infolinia 801 33 33 88

e-mail: [pwn@pwn.com.pl](mailto:pwn@pwn.com.pl)

[www.pwn.pl](http://www.pwn.pl)

# Spis treści

## Wstęp

## Podziękowania

## 1. Konstrukcja książki

## 2. Testowanie w pigułce

Zadanie

## 3. Cykl życia oprogramowania

3.1. Pomysł (potrzeba)

3.2. Rozwój koncepcji

3.3. Planowanie

3.4. Analiza wymagań

3.5. Projektowanie

3.6. Rozwój aplikacji

3.7. Testowanie

3.8. Wdrożenie

3.9. Użycie i utrzymanie

3.10. Emerytura. Koniec życia

3.11. Inne fazy

## 4. Testowanie

4.1. Definicja testowania

Zadanie

4.2. Procesy testowania

4.2.1. Proces testowy wg BS7925-2

4.2.2. Proces testowy wg IEEE 829

4.2.3. Proces testowy wg ISO 29119

4.2.4. Proces testowy wg ISTQB

4.2.5. Podsumowanie

4.3. Błędy, defekty, awarie, incydenty, zdarzenia, bugi...

4.3.1. Uciekinierzy

4.3.2. Błędy popełniane przez testerów

4.3.3. Defekty powodują defekty

Zadanie

4.4. Jakość oprogramowania a użytkownik

4.5. Czym jest testowanie?

4.5.1. Proces oraz zapewnienie jakości

4.5.2. Weryfikować a walidować

4.5.3. Szkoła defektów kontra szkoła jakości

4.5.4. Testy automatyczne

Zadanie

4.6. Testowanie jest potrzebne

4.7. Testowanie jest nieskończone

Zadanie

4.8. O wyższości wczesnego testowania nad późnym

4.9. Ekonomia testowania

Zadanie

## **5. Dzielenie testowania**

5.1. Wprowadzenie

Zadanie

5.2. Czarna skrzynka i biała skrzynka

5.2.1. Testy białej skrzynki

5.2.2. Testy czarnej skrzynki

Zadanie

5.3. Testowanie funkcjonalne i нефunkcjonalne

5.3.1. Testy funkcjonalne

5.3.2. Testy нефunkcjonalne

5.3.3. Charakterystyki oprogramowania wg ISO 9126/ISO 25010

5.3.4. Charakterystyki oprogramowania wg TheTest Eye

5.3.5. Charakterystyki oprogramowania wg Jamesa Bacha

Zadanie

5.4. Testy potwierdzające

5.4.1. Retesty

5.4.2. Testowanie regresywne

Zadanie

5.5. Testowanie statyczne i dynamiczne

5.5.1. Testowanie statyczne

- 5.5.2. Testowanie dynamiczne
- 5.6. Zestawienie testów
- Zadanie

## **6. Zawód: tester**

- 6.1. Wprowadzenie
- 6.2. Edukacja testerska
  - 6.2.1. Edukacja szkolna
  - 6.2.2. Edukacja internetowa
  - 6.2.3. Edukacja przez praktykę
  - 6.2.4. Podsumowanie
- 6.3. Certyfikacja testerska
- 6.4. Testowanie oprogramowania ma swoich wrogów
- 6.5. Cechy miękkie testera
- 6.6. Trudne aspekty pracy testera
- 6.7. Kto może testować produkt?
- 6.8. Umiejętności twarde testera
- Zadanie
- 6.9. Posługiwanie się narzędziami i automatyzacja
- 6.10. Współpraca tester-programista
- 6.11. Rozwój testera w organizacji
- 6.12. Czego oczekuje się od testera na rynku pracy
- 6.13. Zawód z przyszłością
- 6.14. Zarobki testerów
- 6.15. Praca testera w innych publikacjach
- 6.16. Praca w charakterze testera
  - 6.16.1. Modele współpracy

## **7. Praktyka testowania**

- 7.1. Wprowadzenie
- 7.2. Podejścia do testowania
  - 7.2.1. Strategie wynikające z podziałów w testowaniu
  - 7.2.2. Strategia testowania oparta na modelu dostarczania
  - 7.2.3. Podejście negatywne do testów, czyli atak na oprogramowanie
  - 7.2.4. Podejście do testowania zależnie od dostępności specyfikacji
  - 7.2.5. Testowanie oparte na ryzyku
- 7.3. Planowanie
- 7.4. Testowanie

- 7.4.1. Element
- 7.4.2. Formularze
- 7.4.3. Funkcja
- 7.4.4. Logika lub proces

Przykłady

Zadanie

#### 7.5. Raportowanie

- 7.5.1. Subiektywna ocena jakości oprogramowania
- 7.5.2. Raport z testów
- 7.5.3. Raporty o defektach

#### 7.6. Przykładowe projekty

Projekt 1 – strona internetowa

Projekt 2 – strona internetowa z projektem

Projekt 3 – aplikacja internetowa z procesem wspierającym  
wytwarzanie i utrzymanie oprogramowania

Projekt 4 – testy edukacyjnej aplikacji desktopowej

Projekt 5 – testy aplikacji mobilnej

## **Bibliografia**

## **Przypisy**

# Wstęp

Przekazuję w tej książce wiedzę, którą sam chciałbym dysponować, kiedy zaczynałem pracować w zawodzie. Część informacji w niej zawartych może się Czytelnikowi wydać znajoma, ponieważ jest to poprawiona, przeredagowana i uspołniona wiedza, jaką od 2005 roku publikuję w prasie branżowej, zamieszczam na blogach i przedstawiam podczas prezentacji i szkoleń. Skoro część informacji można znaleźć w sieci, po co kupować tę książkę? Przygotowałem ją jako kompilację podstawowych informacji dla tych, którzy szukają drogi na skróty, by zostać testerami, i tych, którzy już zrobili pierwszy krok w tym kierunku.

Wciąż wiele osób pyta, jak zostać testerem. Ta publikacja to moja odpowiedź podana w pigułce. W mojej opinii pokazuje ona najprostszą i najkrótszą drogę do zawodu.

Materiał zawarty w książce został „przećwiczony” na początkujących adeptach testowania i testerach o niewielkim doświadczeniu. Chciałem im serdecznie podziękować za recenzje i uwagi oraz za udział w powstawaniu tej publikacji.

Do każdego bardziej praktycznego tematu próbowałem dodać zadania. Są opisane skrótowo, a ich rozwiązanie wymaga dużej dozy samodzielności i samozaparcia, a także własnej inicjatywy. Pewną pomocą dla tych, którzy wolą pójść na skróty, będą zadania, aplikacje i specyfikacje przygotowane na potrzeby mistrzostwa polski w testowaniu oprogramowania TestingCup, dostępne pod adresem [mrbuggy.pl](http://mrbuggy.pl). Jest tam wystarczająco dużo materiału dla osób, które chcą szukać defektów, uczyć się je raportować i podglądać pracę najlepszych.



# Podziękowania

Inspiracji do powstania mojego portalu [testerzy.pl](http://testerzy.pl) oraz w konsekwencji tej książki dostarczyły setki testerskich blogerów i autorów książek dotyczących lub dotykających testowania. Na świecie są setki wartościowych testerów, którzy swoją wiedzę dzielą się każdego dnia. Publikują ciekawe pomysły, informują o nowościach i sami je wymyślają. Każdy z nich w jakimś sensie zmusił mnie do myślenia o testowaniu. Z częścią z nich się nie zgadzam i próbuję zbijać ich argumentację. Część z nich otwiera mi oczy albo zmusza mnie do nowego spojrzenia na projekty, w których brałem udział, bądź na strategię testową, którą obrałem. To im chcę szczególnie podziękować, bo mieli wpływ na to, jakim testerem jestem dziś.

Chciałem również podziękować swoim współpracownikom. Jedni z nich czytali i recenzowali fragmenty książki, inni mnie inspirowali i wspierali w trakcie pisania, a sporo naszych wspólnych doświadczeń zostało w bardziej lub mniej oczywisty sposób zaprezentowane w tej publikacji.

W dużej części procesu rozwoju w zawodzie testera i pisania tej pracy musiała biernie uczestniczyć moja rodzina. Wiem, że niekiedy poświęcałem za dużo czasu pracy, a za mało spędzałem go w domu. Zbyt często zdarzało mi się być w nim tylko ciałem i rozmyślać o pracy. Może uświadomiłem to sobie odrobinę za późno, ale cieszę się, że dziś udało mi się osiągnąć równowagę. Dziękuję moim najbliższym, że wytrzymali ten czas poszukiwania i odnajdowania. Bez Was ta książka nigdy by nie powstała.

# 1. Konstrukcja książki

Książkę podzieliłem na dwie części. W pierwszej opisuję zawód testera od strony teoretycznej i omawiam aspekty testowania, z którymi możesz się spotkać w pierwszych miesiącach pracy. W drugiej zajmuję się praktyką i podaję swego rodzaju skróty oraz gotowe rozwiązania dla testerów. Dodałem również kilka przykładów, które pomogą ci się odnaleźć w pierwszej firmie, w jakiej przyjdzie ci pracować. Są to przykłady wzięte z życia i dzięki temu może nawet bardziej przydatne.

Chcąc pomóc „wzrokowcom”, wszędzie tam, gdzie grafika pomaga zrozumieć tekst, starałem się ją umieścić. Zastosowałem również tabele dla zwiększenia czytelności zbiorów danych – nie po to, by je pochłaniać podczas lektury, a raczej by je analizować na konkretnych przykładach. Natomiast te fragmenty, które pomagają zrozumieć szerszy kontekst omawianego tematu, ale nie należą do głównego wątku rozważań (np. definicje czy dygresje), zostały ujęte w ramki.

Definicja lub dygresja
Treść

## 2. Testowanie w pigułce

Nie staniesz się testerem tylko z powodu kupienia tej książki. Jesteś nim i zawsze byłeś. Testowanie jest naturalnym procesem i czy nazwiemy je uczeniem się, czy też eksperymentowaniem, nie ma to większego znaczenia dla samej czynności poznawania i podejmowania decyzji o testowanym obiekcie. Przedmiotem osądu może być wszystko, poczynając od wody i żywności, które są naszą podstawową potrzebą, przez narzędzia i produkty, których używamy każdego dnia, po narzędzie najbardziej skomplikowane, jakim jest oprogramowanie. Oceniamy też rzeczy niematerialne lub niedotykalne: obrazy, spektakl teatralny, film, muzykę. Można by tak wymieniać w nieskończoność. Czy jest to testowanie? Wszystko zależy od przyjętej definicji testowania. Jeśli uznamy, że jest nim **doświadczenie i dokonywanie oceny z jednoczesnym opisaniem rzeczy akceptowalnych oraz nieakceptowalnych**, to dlaczego nie?

Nasz osąd może być nie w pełni sprawiedliwy, może być niepoprawny, gdyż wpływa na niego (zbyt) wiele czynników, ale jeśli zadeklarujemy, że jest to nasza subiektywna opinia, to ma ona znaczenie. Im bardziej potrafimy ją zobiektywizować i zaprezentować w odniesieniu do potrzeb dużej grupy ludzi lub też jednej, ale za to ważnej dla wytworzenia danego obiektu osoby, tym większymi profesjonalistami jesteśmy.

Obiekt testów możemy zaakceptować, ale możemy go też odrzucić z informacją, że nie spełnia wymagań. Im bardziej konstruktywnie będziemy go krytykować, tym lepszymi staniemy się testerami.

Nie można być testerem wszystkiego, znać się na wszystkim i powiedzieć, że każdy obiekt oceni się tak samo dobrze. Im większą masz wiedzę wynikającą ze znajomości danego obiektu lub doświadczenia w jego używaniu, tym twoja opinia, również subiektywna, będzie bardziej wartościowa. Ludzie, którzy recenzują książki, sprzęt do biegania czy inwestycje samorządowe, to zazwyczaj

specjaliści, których wieloletnie badania danego obszaru upoważniają do wypowiedzania się na dany temat. Znalezienie swojej niszy jest również wybraniem własnej drogi rozwoju. Możesz się w niej wyspecjalizować, a znając ją lepiej niż ktokolwiek inny, osiągnąć status eksperta. Oczywiście nie musisz być w danej niszy twórcą. Możesz być wyłącznie krytykiem.

Wracając już do obszaru testowania oprogramowania, mogę śmiało powiedzieć, że jesteś testerem oprogramowania, jeśli potrafisz:

1. Uruchomić oprogramowanie i go używać.
2. Przekazać opinię na temat jakości oprogramowania, a na twój osąd składają się następujące informacje:
  - potwierdzenie, że dany obszar aplikacji działa,
  - potwierdzenie, że w danym obszarze znajdują się awarie,
  - zasygnalizowanie (na podstawie własnego przekonania), że coś nie działa tak, jak powinno, lub że są problemy, których nie umiesz jednoznacznie uznać za zachowanie poprawne czy niepoprawne.

Co prawda, jeśli potrafisz tylko tyle, to jesteś dopiero na początku drogi do osiągnięcia profesjonalizmu w tej dziedzinie, ale z całą pewnością pierwszy krok został wykonany. W książce tej rozwinę wszystkie wspomniane tu elementy i zakładam, że jeśli je zrozumiesz i dodatkowo efektywnie przerobisz zawarte w niej zadania, to po zakończeniu lektury możesz o sobie mówić „tester oprogramowania”.

---

## Zadanie

Czy myślisz: „Potrafię być testerem”?

Wykonaj następujące czynności:

1. Znajdź kogoś, kto tworzy lub dostarcza oprogramowanie.
2. Upewnij się, że ten ktoś (nawet byt wirtualny) chciałby usłyszeć twoją opinię o swoim produkcie.
3. Przekaż swoją możliwie najpełniejszą i konstruktywną opinię o produkcie.
4. (Opcja dodatkowa) Zbierz informację zwrotną o swojej opinii. Czy była potrzebna, wartościowa, konstruktywna itd.?

Na świecie jest wielu ludzi, którzy dostarczają oprogramowanie i cieszą się zarówno z pozytywnych, jak i mniej pochlebnych opinii. Jeśli zawiedziesz w pierwszym i drugim punkcie zadania, zastanów się, czy

masz ważną cechę każdego testera, czyli umiejętność identyfikowania potrzeb.

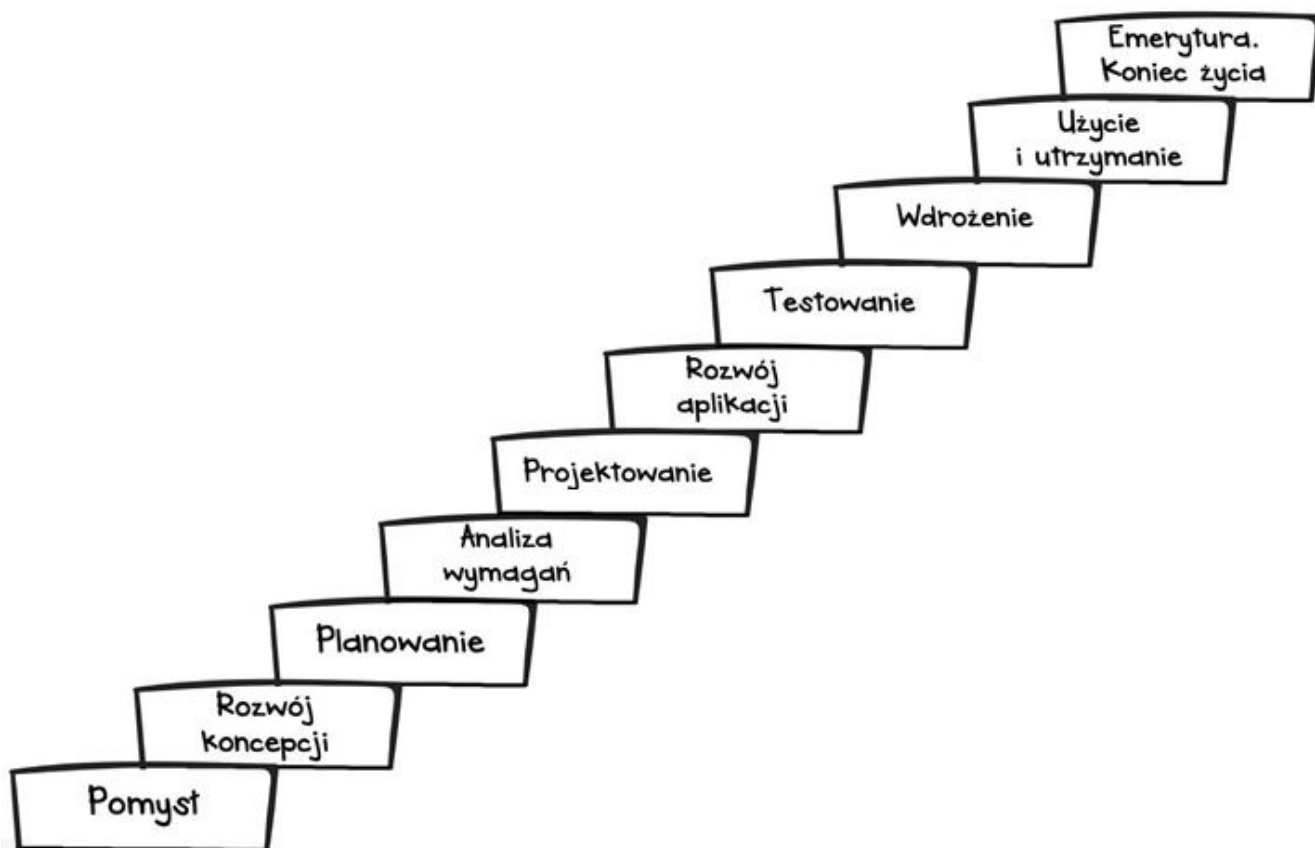
---

### 3. Cykl życia oprogramowania

SDLC (*software development life cycle*) to w wolnym tłumaczeniu cykl życia rozwoju oprogramowania. Opisuje on kolejne fazy powstawania oprogramowania. Jest to skomplikowany proces przetworzenia potrzeby użytkownika w produkt i utrzymywania tego produktu aż do jego wycofania. W większości faz cyklu życia oprogramowania znajdziemy zadania związane z zapewnieniem jakości i testowaniem. Formalny opis nie zagwarantuje powstania produktu wysokiej jakości, na pewno jednak pomoże usprawnić jego wytwarzanie. Jest to szczególnie ważne w dzisiejszych czasach, gdy od tworzonego oprogramowania może zależeć ludzkie życie w pośrednim i bezpośrednim znaczeniu. Dlatego opracowuje się procesy, metody i metodyki wytwarzania oprogramowania i zarządzania nim, które próbują opisać ten skomplikowany mechanizm tworzenia i utrzymania oprogramowania.

Skutki niepowodzenia projektu informatycznego mogą być różne i nie muszą się ujawniać tylko w czasie wytwarzania oprogramowania. Niektóre konsekwencje będą widoczne dopiero w późnych fazach użytkowania oprogramowania. Proces w tym przypadku pomaga zadbać nie tylko o odpowiedni odbiór produktu, lecz ma także zagwarantować niezawodne działanie w długim okresie jego użytkowania.

Mówiąc o procesie wytwórczym, mamy na myśli czynności, które trzeba wykonać, by przekształcić pomysł w produkt, czyli oprogramowanie. SDLC ma szerszy zakres. Jak widać to na rys. 3.1, uwzględnia zadania składające się na proces wytwarzania oraz inne aspekty, takie jak definiowanie pomysłu i jego rozwój, użycie i utrzymanie oprogramowania czy jego wycofanie. W całym procesie możemy wyróżnić dziesiątki faz, zaangażowanych ról, produktów i półproduktów. Przybliżę je, powołując się na popularne w kręgach testerskich źródła wiedzy.



**Rys. 3.1.** Cykl życia rozwoju oprogramowania

Zdefiniowane metodyki wytwarzania oprogramowania zazwyczaj służą do ujęcia w reguły samej fazy tworzenia oprogramowania i osiągnięcia większej przewidywalności w kwestii czasu i kosztów dostawy oprogramowania. Opisy takich metod jak model kaskadowy, model V, podejścia iteracyjne, DSDM, RUP, RAD, Scrum, XP pomijam. Nigdy w swoim zawodowym życiu nie spotkałem organizacji, która wdrożyłaby metodę w formie zdefiniowanej przez jej twórcę. Każdy z modeli, jaki widziałem, był daleko idącą modyfikacją wersji źródłowej. Czasami była to karykatura oryginału, a czasami praktycznie nowa metoda, będąca kolażem wielu innych podejść. Wychodzę z założenia, że dowolne metody powinny stanowić jedynie inspirację do stworzenia czegoś własnego i dopasowanego do naszych potrzeb. Można powiedzieć, że każda organizacja ma własną metodę i nie da się ich wszystkich opisać. Pewnym uproszczeniem byłoby przedstawienie reprezentantów metod, ale to można znaleźć w dziesiątkach publikacji na temat samych metod. Idąc na skróty, opisuję więc fazy

uniwersalnego cyklu życia oprogramowania bez wskazania na konkretną metodę czy metodykę.

### 3.1. Pomysł (potrzeba)

Stare przysłowie mówi: „Potrzeba matką wynalazków”. Wytworzenie oprogramowania zaczyna się od koncepcji. Głównym celem całego procesu będzie zrealizowanie pomysłu, to znaczy przekucie go w działające oprogramowanie. Dobry pomysł zaspokaja potrzebę dużej grupy ludzi. Czasami jednak dobra koncepcja wywołuje zupełnie nową, nieuświadomioną wcześniej potrzebę lub zachciankę. Jeśli tak nie jest, to aby produkt odniósł sukces, poza powodzeniem samego projektu wytworzenia oprogramowania, konieczne są również reklama i marketing. Czy uda się nam przekonać konsumentów, że produkt, którego nigdy nie widzieli i nie chcieli, jest im naprawdę potrzebny?

Głównym udziałowcem tej fazy SDLC będzie pomysłodawca. Może to być klient, który firmie wytwórczej zleca wytworzenie oprogramowania, lub pracownik firmy odpowiedzialny za rozwój jej portfolio. Jego obowiązkiem jest definiowanie nowych produktów czy usług, które mogą być świadczone na rynku.

<b>Klient, zlecający, sponsor</b>
Wymienione terminy, jak i wiele innych, będziemy stosować na określenie osób, które mają pomysł lub pomagają definiować pomysł do implementacji.

Pomysł nie bierze się z niczego. Zazwyczaj wynika z doświadczenia i wiedzy pomysłodawcy, jest także pochodną wielu analiz i wcześniejszych prób. Idee – nawet najbardziej rewolucyjne i śmiałe – nie są same w sobie fundamentem przyszłego produktu. Droga od koncepcji do celu jest długa i kręta. Wcześniej zdefiniowany pomysł zostanie zmodyfikowany do tego stopnia, że może w niczym nie przypominać pierwotnej koncepcji.

<b>Wytwórca, dostawca, producent</b>
Terminy te pojawiają się przy okazji organizacji wytwarzania oprogramowania i określają zazwyczaj firmę, która je dostarcza.



Musimy jeszcze zastanowić się nad mocą sprawczą i odpowiedzialnością poszczególnych ról w procesie wytwórczym. Czasami mówi się, że użytkownicy psują oprogramowanie. Posługując się jednak systemem na poziomie interfejsu, naprawdę trudno go zepsuć. Prawda jest taka, że system jest już zepsuty, kiedy użytkownik zaczyna z niego korzystać. Zawiera defekty, które ujawniły się po jego uruchomieniu.

Podobnie możemy zdefiniować możliwości pomysłodawców, którzy stanowią siłę napędową projektu. Mamy więc do czynienia z wynalazcami oraz odkrywcami. Odkrywca znajduje rzeczy, które już istnieją i jedynie pokazuje do nich drogę. Wynalazca to z kolei ktoś, kto tworzy coś unikatowego, co do tej pory przez nikogo nie było jeszcze zrealizowane. Te dwie osoby (lub zespoły) przejdą dwie różne drogi do komercyjnych premier swoich odkryć i wynalazków.

Odkrywca opiera się zazwyczaj na gotowych rozwiązaniach i korzysta z narzędzi zdefiniowanych przez kogoś innego znacznie wcześniej. Jego przewaga nad innymi wynika przede wszystkim z umiejętności dostrzegania nowych rozwiązań tam, gdzie pozostali widzą jedynie przeszkody. Przykładem odkrywcy będzie Krzysztof Kolumb, który był wystarczająco odważny (głupi?), by zmierzyć się z oceanem i odkryć nowy ląd. Dzięki wysiłkowi, który podjął, pokazał ludziom, że istnieje nowy świat. Czy świat ten powstał, kiedy marynarze Kolumba go dostrzegli? Oczywiście, że nie. Był tam od milionów lat, ale to on miał odwagę po niego sięgnąć.

### **Odkrywca Ameryki**

Osobnym zagadnieniem jest to, czy Kolumb rzeczywiście był pierwszy. Niektórzy historycy kwestionują nawet jego obecność tam. Nie zmienia to faktu, że każdego odkrywcę można zakwestionować i próbować wskazać kogoś, kto był odkrywcą „prawdziwym”. Póki spory historyków nie przycichną, ważne dla nas jest to, kto jako odkrywca funkcjonuje w masowej świadomości.

Projekt prowadzony dla „odkrywcy” nie ma charakteru innowacyjnego i może polegać na wzorcach czy przykładach z przeszłości. Może być realizowany wolniej, ponieważ opiera się na rozwiązaniach już znanych. Poufność danych i ich krytyczność nie są głównymi problemami. Odwrotnie jest w projektach „wynalazców”, gdzie wyprzedzenie konkurencji o dzień ma znaczenie. Liczy się

również utrzymanie informacji o projekcie możliwie najdłużej w tajemnicy. Czasami trzeba definiować pewne rzeczy zupełnie na nowo.

Wynalazca telefonu dzięki swojemu pomysłowi wpłynął na losy świata, jego rozwój i globalizację, jaką dziś obserwujemy. Komercyjne zastosowanie wynalazku uczyniło go również człowiekiem bardzo bogatym.

### **Wynalazca telefonu**

Do dziś istnieje wiele kontrowersji na temat rzeczywistego wynalazcy telefonu. Wynalazek jednak przypisuje się temu, kto go opatentował, czyli Alexandrowi Grahamowi Bellowi.

W projektach mamy do czynienia zarówno z odkrywcami, jak i wynalazcami. Ludzie, którzy znajdują sposób optymalizacji w istniejących już produktach, usługach czy procesach są raczej odkrywcami. Ci, którzy definiują zupełnie nowe koncepcje, będą wynalazcami. We współczesnym świecie trudno o jednego autora danej koncepcji, więc częściej zdarza nam się mówić o zespołach.

Zespoły lub osoby będące wynalazcami w dużo większym zakresie muszą szukać sposobów, które pozwolą testować pomysły. Jednym z nich jest prototypowanie lub jego uproszczona wersja – pretotypowanie. Jest to tworzenie niskobudżetowego modelu danego produktu (lub usługi), aby zweryfikować jego przydatność i potencjalnych użytkowników. Dzięki takiemu podejściu już we wczesnych fazach definiowania pomysłów możemy wyselekcjonować te z nich, które są skazane na porażkę. Oczywiście często się zdarza, że weryfikacja pomysłu jest pomijana lub zawodzi i wtedy otrzymamy tak epokowe wynalazki, jak żelazko z funkcją podgrzewania posiłków czy poduszka ze sztuczną ręką do przytulania.

Ta pierwsza faza nazywana jest czasami inicjalizacją. Jej pomyślne zakończenie, czyli zdefiniowanie pomysłu jest pierwszym krokiem do zamiany bytu niematerialnego w materialny.

## **3.2. Rozwój koncepcji**

Jeśli pomysł wejdzie w tę fazę, zazwyczaj oznacza to, że pojawił się sponsor, czyli człowiek gotowy zainwestować swój czas i pieniądze, by doprowadzić do wytworzenia produktu oraz jego komercyjnej premiery. Sponsor będzie zainteresowany przede wszystkim finansowym zyskiem ze sprzedaży. Aby mu go zagwarantować, potrzebna będzie również szeroka analiza przyszłego produktu.

Podstawowym elementem dla sponsora będzie ROI (*return of investment*), czyli zwrot z inwestycji. Relacja kosztów wytworzenia do potencjalnych zysków pomaga w podejmowaniu decyzji. Skuteczność działań tej fazy będzie zauważalna dopiero w końcowych fazach procesu twórczego. Musimy się pogodzić z faktem, że podjęcie decyzji odbywa się w niepewnych warunkach, gdy jeszcze nie wszystkie informacje ważne dla naszego biznesu są dostępne. Dodatkowo musimy uwzględnić niebezpieczeństwa, które mogą się pojawić w otoczeniu projektu twórczego. Wszelkiego rodzaju ryzyko będzie pochodziło nie tylko z zewnątrz (chodzi o takie aspekty, jak rynek czy polityka danego kraju). Głównym źródłem ryzyka w projekcie jest przede wszystkim czynnik ludzki. Decydujące będą dobór osób o właściwych kompetencjach i odpowiednie motywowanie do pracy, które stanowią o subtelnej, acz znaczącej różnicy między nieudanym a udanym projektem.

<b>Start-up</b>
Firma lub projekt, które cechują się niskobudżetowymi działaniami, dużym ryzykiem niepowodzenia przedsięwzięcia, ale też szansą na duży zwrot z inwestycji.

Absolutnie konieczne w naszych rozważaniach jest zdefiniowanie granic systemu, a przy tym granicy jego wykonalności. Czy produkt, który mamy na myśli, jest możliwy do zrealizowania i czy zadania, które ma wykonać, są osiągalne? Możemy założyć, że wytwarzamy start-up dla 5 mld użytkowników, ale czy nakłady finansowe na jego wytworzenie i zweryfikowanie poprawności nie będą zbyt duże? Zwłaszcza gdy zestawimy je z prawdopodobieństwem powodzenia biznesowego, mierzonego liczbą kupujących nasz produkt lub korzystających z niego. Jeżeli wiemy, że na świecie nie ma nawet tylu aktywnych użytkowników komputerów, to czy rzeczywiście możemy liczyć na aż tak liczną grupę odbiorców? Mówimy tu o zasięgu od najmłodszych po najstarszych przedstawicieli naszego gatunku. Część

z nich zamieszkuje co prawda uprzemysłowione tereny, ale część żyje w buszu, nie mając dostępu do elektryczności. Studium wykonalności (*feasibility study*), które określa realność takiego systemu, powie nam, co jesteśmy w stanie osiągnąć, a czego nie. Nałożone ramy pozwolą nam również na oszacowanie kosztów związanych z wytworzeniem produktu.

### 3.3. Planowanie

Niektóre czynności w procesie wytwarzania oprogramowania są jednorazowe albo powtarza się je niezmiernie rzadko. Druga część to czynności wielokrotnie powtarzane podczas całego cyklu życia. Teoretycznie zaplanowanie projektu informatycznego jest czynnością jednorazową, ale przeplanowanie, redefiniowanie lub też dostosowanie się do nieoczekiwanych (lub oczekiwanych acz niepewnych) zdarzeń jest już czynnością ciągłą.

Zakładamy, że produkt wytwarzany jest w rzeczywistym środowisku i mają na niego wpływ niezliczone czynniki zewnętrzne i wewnętrzne. W okresie wstępnego planowania mamy niepełne dane o możliwym przebiegu procesu tworzenia. Dane te będą spływały dopiero w trakcie jego realizacji. Jeśli informacje wymagają od nas zmiany planów, to musi zostać uruchomiony mechanizm pozwalający ponownie oceniać zasoby lub zarządzać czasem dostawy. Napływ informacji będzie ciągły, zmiany w planie będą również ciągłe.

Podczas planowania znany obszar pokrywamy estymatami o dużej dokładności. Dla obszaru nieznanego lub też dla ryzyka przyjmujemy założenia, które są jedynie przybliżonymi szacunkami. Elementami, które jako klienci chcielibyśmy uznać za stałe, są czas dostawy i budżet. Zakładamy datę premiery naszego produktu i dostosowujemy do tego mechanizm marketingowy, promocyjny czy też sprzedażowy. Oczywiście jest również, że raz zdefiniowany koszt produktu będziemy chcieli utrzymać do końca. Nikt nie lubi rozczarowań. Jeśli towar na półce opatrzony jest ceną X, to nie zakładamy, że przy kasie zapłacimy za niego 2X. Co więcej, bardzo się ucieszymy, jeśli otrzymamy rabat. Podobnie jest z projektem informatycznym – gdy sponsor zaakceptuje cenę Y, trudno mu będzie zrozumieć, że mimo naszej głębokiej analizy na końcu przyjdzie mu zapłacić więcej. Oczywiście w uzasadnionych

przypadkach, np. losowych, będzie w stanie przyznać dodatkowe środki, ale nie może to być więcej niż założony na początku bufor. Wskazuje on na opłacalność lub nieopłacalność inwestycji.

Sytuację, w której sponsor inwestuje dużo ponad założony wcześniej budżet, można porównać z postępowaniem gracza w kasynie, który im więcej przegrywa, tym bardziej chce się odegrać. Nie ma to nic wspólnego z logiką, a wynika z głębokiego przekonania, że szczęście musi się wreszcie do nas uśmiechnąć. Sponsor może więc przeinwestować, ignorując wstępne i aktualne analizy i na siłę doprowadzić projekt do niekoniecznie szczęśliwego finału.

Planowanie i poprawa planowania będzie więc umiejętnym żonglowaniem zasobami i terminami, tak by przy założonym budżecie i czasie dostarczyć wartościowy produkt.

Aspekt planowania zostanie szerzej omówiony w odniesieniu do samego testowania oprogramowania.

## 3.4. Analiza wymagań

Jest to faza, w której analityk (lub grupa) musi przekształcić pomysł w wymagania wobec systemu. Opierając się na wcześniejszych fazach oraz ideach pomysłodawcy, musimy stworzyć model lub dokument opisujący, jak produkt będzie wyglądał i działał po powstaniu. Zazwyczaj mówimy o wymaganiach funkcjonalnych, czyli zestawie funkcji danego produktu. Czasami są to również charakterystyki нефункционалне, takie jak użyteczność (*usability*) czy wydajność (*efficiency*).

W fazie analizy wymagań jest nadawany bieg procesom lub czynnościom ciągłym w procesie. Zaczynamy mówić o inżynierii wymagań jako o pozyskiwaniu wymagań i zarządzaniu nimi. Określa się też, przynajmniej wstępnie, mechanizm wprowadzania zmian, nazywany powszechnie procesem zarządzania zmianą.

### Definicje z zakresu analizy wymagań

**Wymaganie** to warunek oraz umiejętność potrzebna użytkownikowi do rozwiązania problemu lub osiągnięcia celu.

**Proces inżynierii wymagań** obejmuje zidentyfikowanie, przeanalizowanie i zatwierdzenie wymagań wobec systemu.

**Zarządzanie wymaganiami** to proces zarządzania zbieraniem i zmianami w wymaganiach podczas tworzenia systemu.

Grupa narzędzi możliwych do użycia podczas definiowania wymagań jest bardzo ograniczona. Służą one do pozyskania wymagań poprzez współpracę ze zleceniodawcą, np. możemy przeprowadzić wywiad z klientem lub poprosić o wypełnienie kwestionariusza. Ale możemy też zorganizować burzę mózgów, podczas której zbierzemy wszystkie wymagania (również dotychczas ukryte).

Notacja wymagań może być bardzo sformalizowana i oparta na normach, takich jak IEEE 1233 czy IEEE 830. Wymagania mogą być również przypadkami użycia lub też historyjkami użytkownika (*user stories*). Taki zapis zorientowany na użytkownika opisuje za pomocą przykładów lub scenariuszy, w jaki sposób produkt będzie używany przez jego docelowych odbiorców.

Błędy popełniane w tej ważnej fazie prowadzą do wytworzenia produktu, który nie spełnia oczekiwań klienta. Musimy pamiętać, że przekształcenie potrzeb w produkt musi angażować obie strony – klienta oraz wytwórcę. Zdefiniowanie odpowiedniego kanału komunikacyjnego umożliwi im pełniejsze zrozumienie. Staramy się przez to wyeliminować mnogość problemów takich jak np. nieprecyzyjne wymagania umożliwiające szeroką interpretację.

### **Interpretacja**

Największym niebezpieczeństwem w procesie wytwarzania oprogramowania jest możliwość wieloznacznego interpretowania zapisów – w zależności od wiedzy, doświadczenia czy nastroju. Podstawowym zadaniem w całej inżynierii wymagań jest więc wyeliminowanie niejednoznaczności. Uważa się, że jeśli osoba o umyśle ścisłym otrzyma niejednoznaczne zapisy wykonane przez ludzi o podejściu bardziej biznesowym, to ich interpretacja będzie zawsze różna.

Spotkamy się również z wymaganiami, które mogą być wewnętrznie sprzeczne. Dzieje się tak, jeżeli zachowanie produktu jest opisane w różny sposób w różnych miejscach dokumentu. Tworząc takie zapisy, balansujemy na krawędzi przepaści, którą jest niepowodzenie projektu.

Faza analizy wymagań będzie z naszego punktu widzenia istotna dla końcowego zadowolenia klienta. Musimy pamiętać, że nawet gdy

funkcjonalność wydaje się skończona i kompletna, to wymagania mogą nie być do końca określone. Zawsze pozostaje margines wymagań niezdefiniowanych, ważnych dla funkcjonowania biznesu zlecającego. Analityk biznesowy musi więc antycypować potrzeby klienta i pomóc mu je określać.

Nazwa odpowiedzialnych za wymagania
Wspomniany analityk biznesowy to tylko jedna z wielu nazw osób odpowiedzialnych za pomoc w określeniu wymagań. W literaturze przedmiotu spotyka się również określenie „specjalista inżynierii wymagań”.

## 3.5. Projektowanie

Projektowanie oprogramowania to etap przekładania wymagań funkcjonalnych na dokument projektowy. Właściwym i często używanym porównaniem jest odniesienie do budownictwa. Spisane potrzeby i wymagania klienta architekt musi przełożyć na projekt domu. Oczywiście pomijamy tutaj architektów artystów, mówimy o architektach rzemieślnikach. Ich celem jest sporządzenie szczegółowego opisu domu zgodnie z rządzącymi w tym obszarze regułami, a nie tworzenie nowych wartości. Każdy chce mieszkać w pięknym otoczeniu, a nie każdy chce mieszkać oryginalnie, co często wiąże się z pewnym eksperymentem (również kosztowym). Projekt domu docelowo ma zawierać rysunki techniczne z miarami i informacjami o materiałach. Musi uwzględniać nośność budynku, możliwe naprężenia związane z jego obciążeniem i wiele innych zmiennych istotnych dla powodzenia i bezpieczeństwa inwestycji. Projekt jest więc w tym przypadku technicznym i z punktu widzenia zlecającego mało przydatnym półproduktem. Nawet jeśli klient potrafiłby go zrozumieć, to gdyby nie musiał mieć planu architektonicznego, to zapewne nigdy by go nie kupił. Dobry projekt jest jednak istotny dla budowlanców, by mogli cegła po cegle zrealizować wizję klienta. Mam nadzieję, że czytelnik widzi tutaj analogię: budowlaniec to programista, architekt to projektant, a kierownik budowy to menedżer projektu.

W projektowaniu oprogramowania możemy wyróżnić wiele faz i wiele produktów. Mogą być to fazy powiązane z wytworzenie półproduktów, takich jak koncepcja projektu (*conceptual design*), projekt wstępny (*preliminary design*) czy projekt szczegółowy (*detail design*). Każdy z nich odgrywa rolę podstawową dla powodzenia projektu. Zła konstrukcja oprogramowania może powodować konieczność jego przebudowania, a to zwiększa koszty wytworzenia. Warto pamiętać, że na tym etapie pojawiają się pierwsze przesłanki do zaciągania długu technologicznego pokazanego na rys. 3.2. Oszczędności w tej fazie przełożą się na dodatkowe koszty wytworzenia, a nawet załamanie się całego projektu.

	Widoczne	Niewidoczne
Wartość pozytywna	Funkcja	Architektura
Wartość negatywna	Defekt	Dług technologiczny

Rys. 3.2. Dług technologiczny wg Philippe Kruchtena

#### Dług techniczny (technologiczny)

Dług zaciągamy zawsze, gdy w projektach informatycznych idziemy na skróty, tworzymy rozwiązania nieskalowalne i przeciwstawiamy sobie pojęcia wysokiej jakości i szybkiej dostawy. Budowanie oprogramowania, w którym od początku nie dbamy o architekturę systemową lub o kodowanie, powoduje wzrost naszego zadłużenia. Koszty te zazwyczaj nie są widoczne w aktualnej sytuacji, ale zostaną zdiagnozowane na późniejszych etapach SDLC.



## 3.6. Rozwój aplikacji

Elementy zdefiniowane we wcześniejszych fazach są przenoszone do oprogramowania w fazie jego rozwoju, czyli powstawania kodu źródłowego. Oczywiście oprócz przenoszenia funkcjonalności część wysiłku niestety poświęcimy również na nieświadome wprowadzanie do oprogramowania wcześniej popełnionych błędów.

W ramach rozwoju aplikacji tworzymy produkt. Nabiera on realnych kształtów i powoli staje się (mamy nadzieję) oczekiwanym rozwiązaniem. W tej fazie bardzo ważna jest współpraca wytwórców i osób zlecających wytworzenie. Zaangażowanie tych drugich i wola rewidowania produktu na wczesnych etapach jego tworzenia pozwala wprowadzać zmiany i poprawki, które nakierują go na właściwe tory. Na końcu tych torów jest wspólne zadowolenie wszystkich stron. Programiści tworzą dobry produkt, a klient otrzymuje to, czego naprawdę potrzebował.

## 3.7. Testowanie

W dużym uproszczeniu jest to dogłębna analiza produktu pod kątem dopasowania go do potrzeb klienta. Czy klient otrzyma(ł) produkt, który zamawiał. Książkę poświęcam właśnie temu zagadnieniu, więc czytelnik musi jeszcze trochę poczekać na rozwinięcie tego tematu. Należy jedynie wspomnieć, że z testowaniem jest jak z planowaniem. Zacznie się możliwie najwcześniej i zakończy wraz z końcem życia aplikacji.

## 3.8. Wdrożenie

Przekazanie produktu zleceniodawcy to nie jest tylko prosta seria zadań do wykonania. Nie ograniczamy się do instalacji w środowisku produkcyjnym i szkoleń dla użytkowników oprogramowania. Wiele decyzji musi zostać podjętych i wiele dodatkowych działań zrealizowanych, abyśmy świadomie zdecydowali o wdrożeniu aplikacji. Konsekwencje wcześniej podjętych działań widać właśnie w tej fazie.

Wdrożenie to w przypadku wielu projektów jedna z ostatnich czynności, a udany odbiór może rozpocząć celebrowanie sukcesu.

Przejsie na nowy system jest ostatnim etapem w cyklu rozwoju oprogramowania. Firma zaczyna korzystać z nowego programu, co może się odbywać jednocześnie z wykorzystaniem starego. Do poprawnego działania systemu niezbędne mogą być pewne zmiany lub aktualizacje, w tym m.in. usuwanie awarii sprzętu, poprawa i korygowanie defektów, właściwe zabezpieczenie oprogramowania oraz szkolenie pracowników i administratorów z obsługi systemu. Od tego momentu system zaczyna stawać się integralną częścią codziennej pracy zleceniodawcy.

### **3.9. Użycie i utrzymanie**

Każdy młody programista, wybierając swój zawód, dostrzega daleką perspektywę kreatywnej pracy twórczej, która ma na celu wytworzenie produktu. Jest oczywiste, że duża część z nich będzie tej szansy pozbawiona. Okazuje się, że istotne stają się projekty utrzymania oprogramowania, a nie jego wytwarzania. Faza rozwoju aplikacji to zazwyczaj 10% okresu jej życia. Oprogramowanie jest dużo częściej utrzymywane, co w uproszczeniu można by nazwać dostosowywaniem go do nowych i ciągle zmieniających się potrzeb rynku. Tak więc nie ma tworzenia czegoś od podstaw, a jest jedynie żmudne, odtwórcze zrozumienie pracy innego programisty i próba jej poprawienia.

Ważni w tej fazie są jednak użytkownicy. To oni z radością, przykrością lub obojętnością posługują się dostarczonym oprogramowaniem. Oni są też wyzwalaczem zmian i modyfikacji w oprogramowaniu. Zdarza im się znaleźć defekt, który przeszedł przez sito testów, lub też stwierdzić brak funkcji istotnej dla ich pracy z aplikacją. Oni więc, korzystając z pomocy helpdesku, zgłaszają problemy, które wspomniani tu programiści muszą skorygować. Użytkownicy mogą dostrzegać błędy popełnione w fazie analizy, podczas identyfikowania pewnych potrzeb. Mogą również wskazywać, co można zrobić, by produkt jeszcze lepiej wspierał ich pracę. Tu również, po zatwierdzeniu zmian przez sponsorów, pojawi się konieczność modyfikacji dokonywanej przez programistów.

Użycie i utrzymanie stanowi ważną część życia aplikacji. Jest to także bardzo trudne wyzwanie dla testerów.

### 3.10. Emerytura. Koniec życia

*Wszystko, co ma początek, ma też koniec.  
Wyrocznia, „Matrix Rewolucje”*

Posługuję się znanym cytatem, by wprowadzić odrobinę patosu na koniec rozdziału o SDLC. Pomysł raz zdefiniowany, rozwijany, implementowany – z sukcesem lub nie – na końcu musi odejść. Koniec życia aplikacji to zazwyczaj także koniec oferowanego dla niej wsparcia lub też wyparcie jej przez nowszą wersję. Przykładów takiego rozwiązania można znaleźć mnóstwo, wystarczy przypomnieć los systemu operacyjnego Windows XP i jego koniec spowodowany pojawieniem się Windows 7 czy późniejszego 8, 10 itd. Być może jest na świecie kilku zapaleńców, którzy uznają, że FireFox w wersji 1.0 nigdy nie zostanie pobity przez nowsze wersje. Stanowią oni jednak nieliczącą się mniejszość, która w końcu będzie musiała migrować do nowszej wersji albo wymrze w sposób naturalny.

#### **Migrować**

Migracja jest popularnym w świecie informatycznym pojęciem. Migrujemy do nowszej wersji istniejącego oprogramowania albo do nowego narzędzia. Migrujemy również nasze dane do nowego systemu.

Wycofywana aplikacja z zasady nie powinna być permanentnie usuwana, gdyż nigdy nie wiadomo, czy kiedyś się jeszcze nie przyda. Przechowuje się ją, na wypadek gdyby ta aktualna aplikacja zawiodła.

Oprogramowanie wycofywane zastępowane jest nowszą wersją lub inną aplikacją. W obu przypadkach wyzwaniem dla zespołu wdrożeniowego jest odpowiednia migracja danych do nowego systemu czy zweryfikowanie poprawności działania nowego oprogramowania w porównaniu do starej wersji.

### 3.11. Inne fazy

W cyklu życia są jeszcze inne fazy, działania i poddziałania, których nie opisałem, co nie zmienia faktu, że w niektórych projektach i dla niektórych osób będą to ich podstawowe zadania w całym procesie. Zaliczają się do nich m.in.:

- negocjacje kontraktu – zdefiniowanie, za jaką cenę produkt zostanie wytworzony i do czego zobowiązują się obie strony;
- zarządzanie ryzykiem – ciągle definiowanie, monitorowanie i zapobieganie ryzykom lub ich skutkom (minimalizowanie ryzyka jest pracą w ramach szeroko pojętego zarządzania jakością);
- zarządzanie zgłoszeniami – standardowe zarządzanie defektami oraz zmianami;
- zapewnienie jakości – czynność związana z definiowaniem procesów, procedur oraz wzorców dokumentacji.

## 4. Testowanie

### 4.1. Definicja testowania

Każdy obszar, w którym przyjdzie ci, czytelniku, kiedykolwiek pracować, powinien zostać przez ciebie zdefiniowany. Definicja pomaga odpowiedzieć sobie na pytanie: „Co należy do moich obowiązków?” Światłe głowy starają się ciągle określić, czym jest testowanie, więc twoja praca powinna być prosta. Tak niestety nie (do końca) jest. Jak w wielu innych obszarach, również w testowaniu są sprzeczne interesy, różne szkoły, organizacje i podejścia do problemu. Każda z nich linię definicji próbuje przeciągnąć na swoją stronę, jednocześnie wmawiając nam, że druga strona kompletnie się myli i nie ma (bo nie może mieć) racji. Co gorsza, niektóre organizacje zrzeszające testerów same sobie przeczą w definiowaniu testowania. Inni z kolei definiują testowanie tak szeroko, że trudno je odróżnić od wytwarzania oprogramowania. Przypomina to nieco targ, na którym przekupki próbują przelicytować konkurencję argumentami, że „Moje jest lepsze od innego, bo ma więcej”. Każdy będzie nam wmawiał, że to jego definicja jest najpełniejsza lub jedyna poprawna. W ogólnym chaosie komunikacyjnym tester lub adept testowania musi poznać ofertę wielu organizacji definiujących testowanie, aby wybrać własną drogę.

W tym ogólnym rozgardiaszu ludzie i firmy funkcjonują od lat. W rezultacie niektórzy pracodawcy nie widzą różnicy między testowaniem a definiowaniem procesów, a testerzy w ich firmach są nazywani specjalistami ds. zapewnienia jakości, inżynierami jakości czy też QA-owcami (od ang. *quality assurance* – czyli zapewnianie jakości).

Testowanie doczekało się pewnego standardu, ale nie doczekało się standaryzacji, co niekoniecznie musi być problemem. Musimy definicję testów wyłuskać z tego, co aktualnie mamy, a także skorzystać z wiedzy pobocznej. Wspomniane zapewnienie jakości, programowanie oraz zarządzanie projektami mają swoje definicje. Testowanie próbuje

funkcjonować na styku tych obszarów, możemy więc wykorzystać je do określenia granic testowania.

Każdego z nas prędzej czy później czeka konieczność znalezienia odpowiedzi na pytanie, czym jest dla niego testowanie. Pytań będzie wiele, a odpowiedzi nie będą jednoznaczne.

- Czy testowanie jest szukaniem defektów?
- Czy testowanie jest weryfikacją poprawności działania funkcjonalności?
- Czy testowanie jest używaniem i ocenianiem?
- Czy testujemy po to, aby pokazać, że coś działa?

„Ostateczna definicja testowania” jest tworem mitycznym i prawdopodobnie nigdy nie uda się jej sformułować. Przyjdzie dzień, w którym powiemy sobie: „Tak, mam TO! Określiłem definicję testowania”. Niestety po upływie pewnego czasu, po nowych doświadczeniach uświadomimy sobie, że jednak „Nie, to nie było TO”. Może w takim razie nie warto się męczyć? Może warto popłynąć z nurtem? Każdy kolejny pracodawca przedstawi własne postrzeganie testowania, definiując zakres obowiązków na naszym stanowisku, a my po prostu zaakceptujemy to, co otrzymamy. Jest to jakieś rozwiązanie, konformistyczne do granic, ale zawsze rozwiązanie.

Część z nas zostanie na wysokim poziomie ogólności i uzna, że „testowanie jest czynnością w cyklu życia oprogramowania” albo „testowanie jest procedurą wykonywania testów” lub inną nic nieznaczącą zbitką słów. Definicje ogólne mają mniejszą szansę, by się zdezaktualizować, bo nie niosą żadnej informacji ani niczego konkretnie nie opisują. Mamy też szansę łatwo znaleźć „wyznawców” naszej definicji. Analizując masowego odbiorcę i informacje przekazywane mu przez media, łatwo możemy określić własne zasady formułowania definicji. Musi to być krótki komunikat, zrozumiały i docierający do największego grona. Im więcej w nim konkretów, tym większe pole do negocjowania i do zwykłej polemiki.

„Testowanie jest ważną częścią każdego procesu wytwórczego”. Lepiej nie mówić „najważniejsze”, bo łatwo zantagonizować środowisko. Ta definicja nie jest realnie definicją, ale skoro my powiemy, że nią jest, to niech nasi oponenti męczą się, by ją negocjować.

„Testowanie jest jakością” to bzdura dla wszystkich, którzy coś o testowaniu wiedzą, ale dla nieświadomych brzmi jak hasło wyborcze.

Nie ma niczego znaczyć, ma być łatwe do zapamiętania.

„Testowanie jest sprawdzeniem” – i niech mi ktoś powie, że nie jest.

Dla wielu osób powyższa analiza jest problemem wyssanym z palca. A co mnie obchodzi jakaś tam definicja? Dla tych osób definicja staje się wirtualna i brzmi: „Testowanie”. Oni znają odpowiedź na każde pytanie np.:

- Czym się zajmujesz na co dzień? „Testowaniem”.
- Co to znaczy? „Testuję”.
- A co testujesz? „To, co wymaga testowania”.
- A jak to robisz? ... i tak dalej.

Powinniśmy zaakceptować również takie podejście.

Na drodze do definicji znajdziemy wielu doradców. Nie warto przytaczać każdej definicji, jaka kiedykolwiek się pojawiła, warto za to pracować nad tą jedną, która cię urzekła. Dla mnie jest to zdanie wypowiedziane przez Jamesa Bacha:

*Testing is an infinite process of comparing the invisible to the ambiguous in order to avoid the unthinkable happening to the anonymous.*

Czyli w wolnym tłumaczeniu:

*Testowanie jest niekończącym się procesem polegającym na porównywaniu tego, co niewidzialne, do tego, co wieloznaczne, a wykonuje się je po to, aby coś, co jest niemożliwe do przewidzenia, nie dotknęło tych, których nie znamy z imienia.*

Po pierwszym czytaniu tego zdania, wydaje się, że mamy do czynienia ze zwykłym bełkotem. Po głębszej analizie okazuje się ono niezmiernie ciekawym opisem wypaczonej logiki, jaka towarzyszy testowaniu.

- Pokazuje ona, w jakim środowisku przychodzi nam pracować i jak trudne jest nasze zadanie.
- Wiemy, że nigdy nie skończymy naszej pracy, ale ją zaczynamy i kontynuujemy.
- Kiedy dochodzimy do końca, wiemy jeszcze mniej niż na początku.
- Mimo wielu testów większości zdarzeń w testowanym urządzeniu i tak nigdy nie wykonamy.

- Dokumentacja, która ma informować, jak działa system, w większości ma defekty i zamiast rozwiązywać problemy, wprowadza jeszcze więcej zamieszania.
- Mimo że reprezentujemy „użytkownika końcowego”, niczego o nim nie wiemy (albo wiemy niewiele) i nie jesteśmy w stanie w pełni go zrozumieć.
- Im dłużej testujemy oprogramowanie, tym więcej defektów powinniśmy znajdować, a tak naprawdę więcej ich akceptujemy i omijamy.
- Chodzimy na kursy, które mają nas nauczyć dobrych praktyk, a kiedy się ich nauczymy, nikt nie pozwala nam ich wdrożyć.
- Mamy głębokie przeświadczenie, że robimy rzeczy ważne, ale na końcu okazuje się, że nasza praca idzie na marne, bo ktoś nad nami akceptuje niską jakość oprogramowania.

Może w interpretacji tej definicji posunąłem się za daleko, ale wykorzystuję każdą okazję, by wylać żale powiązane ze złym traktowaniem testowania. Na czym one polegają i czemu służą, opiszę w kolejnych rozdziałach.

### **To zależy...**

Testowanie oprogramowania jest obszarem opisywanym na wiele sposobów, ale jest jedna uniwersalna odpowiedź, która pomaga wyjść z dowolnej opresji, jeśli opresją jest trudne pytanie o testy. Ta odpowiedź brzmi: „To zależy...”. Przykłady:

Kiedy skończysz testować? To zależy od tego, ile defektów będę znajdować.

Jaką strategię wybierasz? To zależy od kontekstu i analizy otoczenia projektu.

Czy znajdziesz wszystkie defekty? To zależy od tego, jaki dostanę budżet.

Ogólnie w testowaniu wszystko zależy od wielu czynników, które musimy wziąć pod uwagę i na końcu wskazać odpowiedź. Czy będzie ostateczna? To zależy... Również definicja testowania zależy od pewnych okoliczności okołoprojektowych, oczekiwań klientów, wiedzy testerów i wielu innych czynników.

Oto kilka propozycji definicji testowania przesłanych przez czytelników portalu [testerzy.pl](http://testerzy.pl) w odpowiedzi na pytanie: „Czym jest dla mnie testowanie?”.



[...] sztuka psucia wszystkiego i szukania dziury w całym, nawet jeśli gdzie indziej to coś działa. (Mariola)

[...] próba wyciśnięcia jakości z bitwy między developerami a testerami. Im krwawsza walka, tym wyższa jakość! Na szczęście w tej bitwie nie ma przegranych. (Marek)

[...] to ratowanie świata przed zawaleniem: Ratujemy producentów oprogramowania przed niemiłymi konsekwencjami oddania do użytku oprogramowania zawierającego nieznane błędy. Ratujemy klientów, którym ma być dostarczone oprogramowanie, przed niemiłym zawodem związanym z możliwością otrzymania oprogramowania niespełniającego ich wymagania. Tym samym sprawiamy, że otaczający nas świat jest piękniejszy (lepszey jakości). (Maciej)

Testowanie to proces mający na celu stworzenie używalnego i funkcjonalnego produktu z owoców pracy programistów. (Piotr)

I na koniec podejście, z którym testerom trudno się pogodzić i na które nie możemy (jako testerzy) pozwolić. To myślenie ludzi, których Alfred Pennyworth z filmu „Mroczny rycerz” określa krótko: *Niektórzy chcą tylko patrzeć, jak świat płonie*. To ci, którzy mówią, że testowanie oprogramowania jest niepotrzebne, a ich definicja testów zawiera takie sformułowania jak „marnotrawstwo”, „bezwartościowy wysiłek” czy „brak wartości dodanej”. Im uwagi w tej książce poświęcę możliwie jak najmniej.

### **A co z definicją jakości?**

Oto kilka definicji znalezionych w różnych źródłach:

*Jakość*

1. «wartość czegoś»
2. filoz. «istotne cechy przedmiotu wyróżniające go spośród innych» – Słownik języka polskiego

*Jakość produktu/usługi to miara braku wad w tym produkcie, a wadą jest każda negatywna cecha produktu – negatywna z punktu widzenia klienta – której klient ma się prawo nie spodziewać – autor nieznany*

*Pewien stopień doskonałości – Platon*

*Jakość (jak piękno) jest sądem wartościującym, wyrażonym przez użytkownika. Jeśli nie ma takiego użytkownika – nie ma takiego sądu – Platon*

*Zgodność ze specyfikacją, czyli zero braków – Wikipedia*

*Jakość to stopień, w jakim zbiór inherentnych właściwości spełnia wymagania – Norma PN-EN 9000:2001*  
*Przewidywany stopień jednorodności i niezawodności przy możliwie niskich kosztach i dopasowaniu do wymagań wyniku – William E. Deming*  
*Jakość to zgodność z wymaganiami – Philip B. Crosby*  
*Jakość to doskonałość, której nie da się osiągnąć, lecz do której trzeba uporczywie zdążać – Lao Tsu*  
Warto zwrócić również uwagę na trzy definicje ludzi nam współczesnych, którzy zajmują się zapewnieniem jakości i testowaniem:  
*Jakość jest wartością dla kogoś – Jerry Weinberg*  
*Jakość jest wartością dla kogoś ważnego – James Bach*  
*Jakość jest wartością dla kogoś w pewnym czasie – Michael Bolton*

## Zadanie

A jaka jest twoja definicja testowania? Może powielać istniejące definicje, może być połączeniem wielu z nich, ale zachęcam cię do określenia, co rozumiesz i chcesz rozumieć przez testowanie. Zapamiętaj tę definicję, a jeśli pamięć masz krótką, to stwórz z niej tapetę na pulpit albo napisz na kartce i przyklej nad biurkiem. Im dłużej będziesz testować, tym bardziej zmieniać się będzie twoja definicja. Pamiętaj o wszystkich od początku do końca.

## 4.2. Procesy testowania

Procesowe podejście do testowania nie musi oznaczać „ciężkich” metod i metodyk. Większości z nas proces testowania kojarzy się jednoznacznie z dziesiątkami dokumentów i setkami procedur, których poznanie i przestrzeganie wymaga się od pracowników. Na szczęście to reguła, którą da się obejść. Proces testowy może być również „lekki”. Posługuję się pojęciem procesu, aby przedstawić podstawowe czynności i obowiązki testera oprogramowania.

### Co i gdzie

W niektórych organizacjach zmusza się pracowników do pamięciowego przyswajania procesów, co nieuchronnie musi kojarzyć się z czasami

szkolnymi. To z kolei przypomina mi żart z brodą:  
*Student wie wszystko. Magister wie, w których książkach jest wszystko.  
Doktor wie, gdzie jest biblioteka. Profesor wie, gdzie jest doktor.*  
Legenda:  
Student = pracownik  
Magister = bezpośredni przełożony pracownika  
Doktor = kierownik projektu lub specjalista ds. jakości, zazwyczaj odpowiedzialny za przestrzeganie podejścia procesowego w organizacji  
Profesor = szef działu informatycznego

Sednem książki jest pokazanie, jak definiowano, definiuje się i będzie się definiować procesy testowe. Choć branża IT jest relatywnie młoda, to wiele podejść procesowych udało się już szczegółowo opisać, a część nawet zanegować i odrzucić. Celem tego rozdziału jest pokazanie najbardziej uniwersalnych procesowych zasad testowania oprogramowania. Przedstawiając wiele procesów, chcę pokazać, z jednej strony, podobieństwo między nimi, a z drugiej, jak testowanie może wyglądać.

#### **4.2.1. Proces testowy wg BS7925-2**

Proces testowy zawarty w nigdy nie opublikowanej oficjalnie normie BS7925-2 dotyczy, jak i sama norma, testów jednostkowych. Poświęcam czas na jej przedstawienie, bo ma charakter ponadczasowy. Mimo upływu ponad dekady od czasu pojawienia się ostatniej (ciągle jedynie szkicowej) wersji proces ten nie stracił aktualności, choć niektóre elementy mogą trącić myszką (podobnie jak sformułowanie „trącić myszką”). Omawiając proces, postaram się odejść od komponentowego charakteru dokumentu i skupić się na elementach, które mogą być stosowane również w innych obszarach testowania.

Na proces składają się następujące czynności:

- Planowanie testów, czyli odniesienie ich strategii oraz dokumentacji projektowej do kontekstu aplikacji. Każde odstępstwo od tych dokumentów należy opisać z uzasadnieniem. Plan zawiera również specyfikowanie interakcji między poszczególnymi programami w ramach wykonania testów.

- Specyfikowanie testów to projektowanie przypadków testowych na podstawie technik predefiniowanych i określonych w planie testów.
- Uruchomienie testów to zestaw czynności wykonanych na już działającej aplikacji. Warto dopowiedzieć, że norma BS podkreśla wagę tworzenia i wykonania przypadków testowych, które będzie można w przyszłości wielokrotnie powtarzać.
- Zapisywanie wyników testów to nie tylko raportowanie defektów, lecz przede wszystkim zbieranie informacji o uruchomieniu danego przypadku testowego. Można uznać, że szczegółowa informacja o uruchomieniu przypadku testowego z sukcesem (bez ujawnienia defektu) jest największą korzyścią w odniesieniu do testów opartych na eksploracji, o których będzie mowa później. Informacja ta przekłada się na miary będące równocześnie podstawą do definiowania kryterium zakończenia testów. Sam defekt, definiowany jako różnica między zachowaniem oczekiwanym a aktualnym, powinien być poddany szczegółowej analizie. Wynikiem analizy może być informacja o defekcie w specyfikacji testowej, ale także informacja o konieczności poprawienia defektu i zaplanowanie testów potwierdzających poprawne działanie oprogramowania po poprawce.
- Sprawdzenie kryterium zakończenia testów. Kryteria oczywiście muszą być zdefiniowane z wyprzedzeniem. W przypadku niespełnienia kryteriów po już zakończonej fazie uruchomienia i zapisaniu wyników musimy zidentyfikować braki w oprogramowaniu lub samych testach. Może to być przyczynkiem do uruchomienia całego procesu od nowa. Braki w testach przekładają się czasami na konieczność przygotowania dodatkowych przypadków testowych, tak by osiągnąć zakładany poziom pokrycia testami.

Proces testowy zgodny z normą BS 7925-2 jest pokazany na rys. 4.1.

Planowanie i określanie końca testów całego oprogramowania zgodnie ze standardem wykonuje się jednorazowo, a specyfikowanie, uruchamianie i zapisywanie są realizowane iteracyjnie.

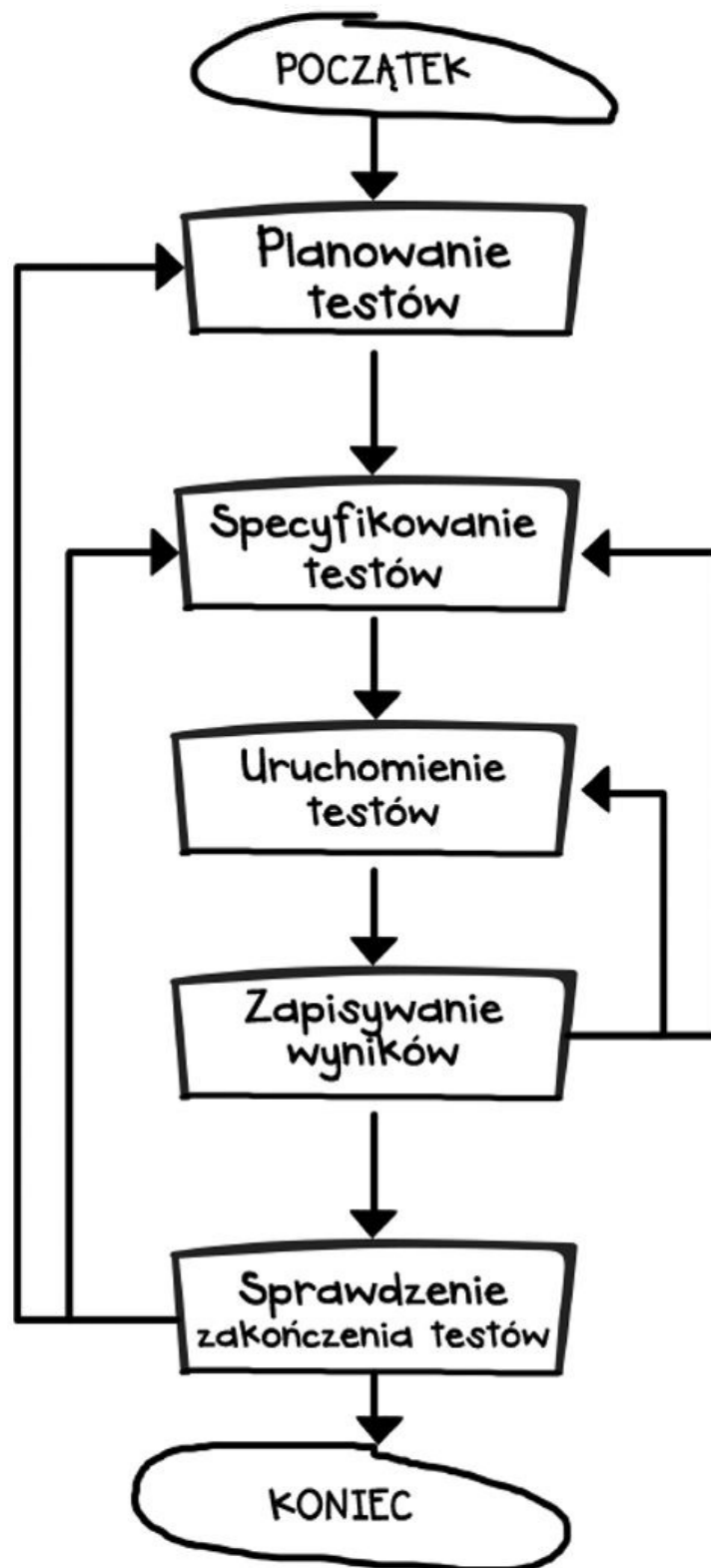
## Iteracja

To kolejne słowo-klucz we współczesnym świecie IT. Mamy iterację w kodzie źródłowym w postaci pętli powtarzanych czynności. Mamy również iterację w prowadzeniu projektów informatycznych, gdzie

pewien zbiór czynności będzie powtarzany regularnie w celu wytworzenia oprogramowania działającego i dopasowanego do klienta.

Strzałki na rys. 4.1 informują, że czynności procesu trzeba wykonywać w kolejności, ale równocześnie wskazują na konieczność uzyskania informacji zwrotnej. Wymusza to powrót do czynności wykonanych poprzednio:

- Niespełnienie kryterium zakończenia testów zmusza do ponownego planowania i uruchomienia procesu wytwarzania od nowa lub do dopisania kolejnych testów.
- Wynik testu może być przyczynkiem do modyfikacji specyfikacji testowej lub do kolejnego uruchomienia oprogramowania w celu sprawdzenia dostarczonej poprawki.



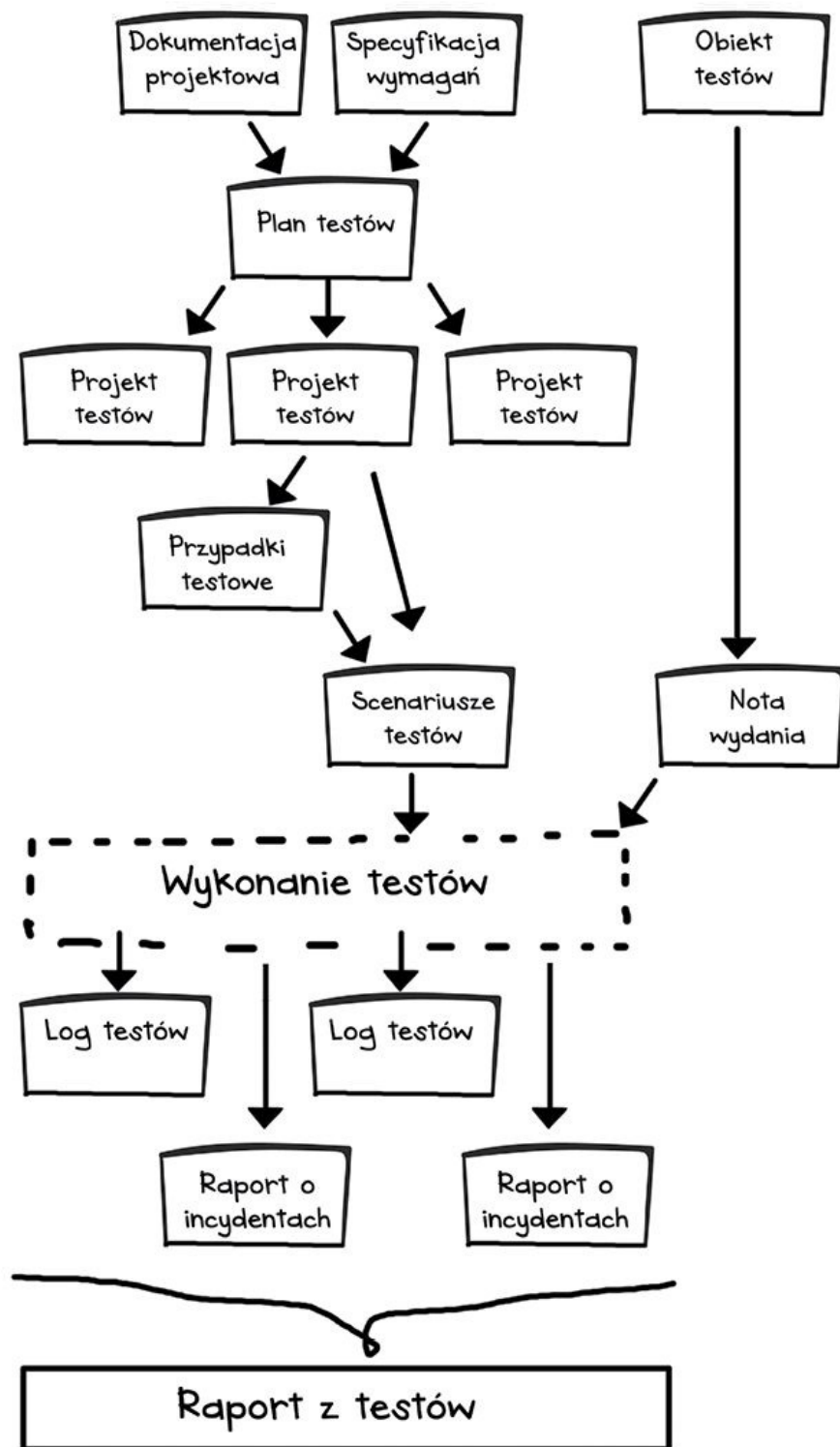
**Rys. 4.1.** Proces testowy wg normy BS-7925

Opis procesu, choć skromny, pokazuje bardzo wyraźnie nie tylko czynności testowe, ale również produkty wytwarzane w ramach samego testowania i przygotowań do niego.

#### **4.2.2. Proces testowy wg IEEE 829**

Norma IEEE 829 określa dokumentację powstającą w trakcie testowania oprogramowania. Pełna angielska nazwa brzmi „Standard for Software and System Test Documentation” i dotyczy dwóch wersji o tej samej nazwie, ale z inną datą publikacji. Pierwsza pochodzi z roku 1998 i została już oficjalnie wycofana, a druga, z 2008 roku, jest wersją obowiązującą. Dezaktualizacja normy IEEE 829-1998 i jej ogólnodostępna w internecie publikacja była przyczyną znacznego wzrostu zainteresowania jej zawartością. International Software Testing Qualification Board i Information System Examination Board, czyli dwa ciała oferujące najbardziej rozpoznawalne ścieżki certyfikacyjne dla testerów na świecie, ciągle promują już nieaktualną wersję normy, traktując ją jako podstawę przygotowania do egzaminów.

Norma ta, która *de facto* prezentuje wzorce dokumentacji, może być wytyczną do procesowego opisania podejścia do testowania jakości oprogramowania tak jak pokazano to na rys. 4.2. Jeśli przyjrzymy się przez lekko przymknięte oczy grafice prezentującej relacje między dokumentami w procesie testowym (wersja 1998), to dostrzeżemy wyraźnie (lub też nie), że na całościowy obraz składają się informacje wejściowe, wyniki czynności oraz produkt końcowy, będący informacją o jakości oprogramowania. Ilustruje to więc pewien proces. Ponieważ sama norma może uchodzić za zabytek, a użyte w niej pojęcia za martwy język, w wizualizacji posłużyłem się pojęciami popularnymi w polskich projektach testowych. Zastępuję oryginał z pełną świadomością jego nieprzystawalności do realiów współczesnego IT.



Rys. 4.2. Proces inspirowany IEEE 829 (interpretacja własna)



Informacją wejściową dla powstania dokumentów w normie są: dokumentacja projektowa, specyfikacja wymagań opisująca testowany obiekt oraz sam obiekt testów. Dokument projektowy powinien nam powiedzieć, co mamy zrobić i w jakim czasie. Opis obiektu testowego powie nam, jak powinien się on zachowywać, a analiza obiektu pozwoli nam to zachowanie zweryfikować. Mamy więc do czynienia z informacją pełną, pozwalającą każdemu rozgarniętemu testerowi przystąpić do działania. Kolejne elementy grafiki to przede wszystkim półprodukty, ale również czynności wykonywane w trakcie samego testowania. Zaczynamy od standardowego planowania stanowiącego podstawę wszystkich innych czynności. Sposób projektowania i wytworzenia przypadków testowych i scenariuszy testowych jest raczej dowolny i sama norma pozostawia tu sporo miejsca na własne dywagacje. Wykonanie testów nie jest dokumentem, a wspomnianą wcześniej aktywnością. Ten stan pośredni ma oddzielać część przygotowania do testowania od określania jakości produktu. Dzięki takiej wizualizacji nasz „proces” zachowuje ciągłość mimo pewnej niekonsekwencji w notacji. Log testowy (tłumaczony na polski jako dziennik testowy) oraz raporty z incydentów stanowią wstęp do najważniejszego dokumentu opisującego działanie bądź niedziałanie systemu – raportu z testów.

### **Procedura a scenariusz**

Wszyscy adepci ISTQB (International Software Testing Qualification Board) poziomu podstawowego mają kłopot z procedurą. Dla części osób wystarczającym wyjaśnieniem istoty procedury będzie jej tożsamość ze scenariuszem testowym. Dokładna definicja z nomenklatury normy IEEE 829 mówi: *Dokument określający ciąg akcji umożliwiający wykonanie testu*. Należy jednak pamiętać, że samemu scenariuszowi testów znacznie bliżej jest do zbioru testów (*test suite*) niż do procedury.

Rozwinięcie tematu podejścia procesowego normy IEEE 829 stanowi wersja 2008, która ma rozdział dodatkowo nawiązujący bezpośrednio do procesów testowych. Warto wspomnieć, że ta norma odwołuje się do innej normy, opisującej procesy cyklu życia oprogramowania (IEEE/EIA 12207.0-1996). Zgodnie z obiema normami testowanie wspiera swoimi czynnościami następujące procesy:

1. Zarządzanie, w którym czynności testowe sprowadzają się do zarządzania testowaniem, w tym planowania i wspierania kierownictwa projektu.
2. Przejęcie/przyjęcie, w którym zadania testerskie dotyczą wsparcia przy estymacji wysiłku testowego i definiowania kryteriów odbioru.
3. Dostawy, w której główną czynnością jest planowanie z definiowaniem metryk i określanie poziomu złożoności oraz krytyczności oprogramowania.
4. Rozwój to już obszar silnie naszpikowany działaniami testerskimi, stąd też wiele czynności, takich jak:
  - rewidowanie specyfikacji wymagań, budowanie matrycy śledzenia,
  - przygotowanie planów, identyfikacja ryzyk,
  - wytworzenie procedur i przypadków testowych,
  - wykonanie testów,
  - instalacja oprogramowania i raportowanie.
5. Operowanie/używanie, w którym główne czynności to zadania związane z testowaniem już działającego oprogramowania.
6. Utrzymanie – czynności to zarówno testy, jak i analiza anomalii w działaniu systemu.

#### **Zadania w IEEE 829 – 2008**

W ramach każdej czynności testerskiej wyróżnia się jedno lub więcej zadań testerskich, które z kolei opisane są za pomocą informacji wejściowych oraz informacji wyjściowych. Takie podejście wskazuje na mocne procesowe podstawy normy.

Warto podkreślić, że norma IEEE 829, choć nie jest normą procesową, to dość skutecznie próbuje uwzględnić mapowania procesu testowania na proces wytwarzania. Przez wiele lat norma pokazywała prostą kolejność działań w testowaniu, którą można stosować jako wytyczne do samego uruchomienia i prowadzenia testowania. Znam niejedną organizację traktującą tę normę jako główny przewodnik i mapę dla testerów, by mogli się odnaleźć w płątaniu wzorców i labiryncie definicji.

### **4.2.3. Proces testowy wg ISO 29119**

Norma ISO 29119 to najnowszy standard testowania oprogramowania. Mimo kontrowersji wokół samego dokumentu warto podkreślić, że dość szeroko opisuje ona podstawowe i zaawansowane aspekty testowania.

Norma ISO/IEC 29119 Software Testing jest pierwszą, kompleksową próbą opisu testowania oprogramowania. Pierwsze prace nad nią rozpoczęły się w maju 2008 roku i na rynku pojawiła się w 2014 roku. Proces testowy jest jedną z części standardu.

Dokument jest podzielony na cztery części:

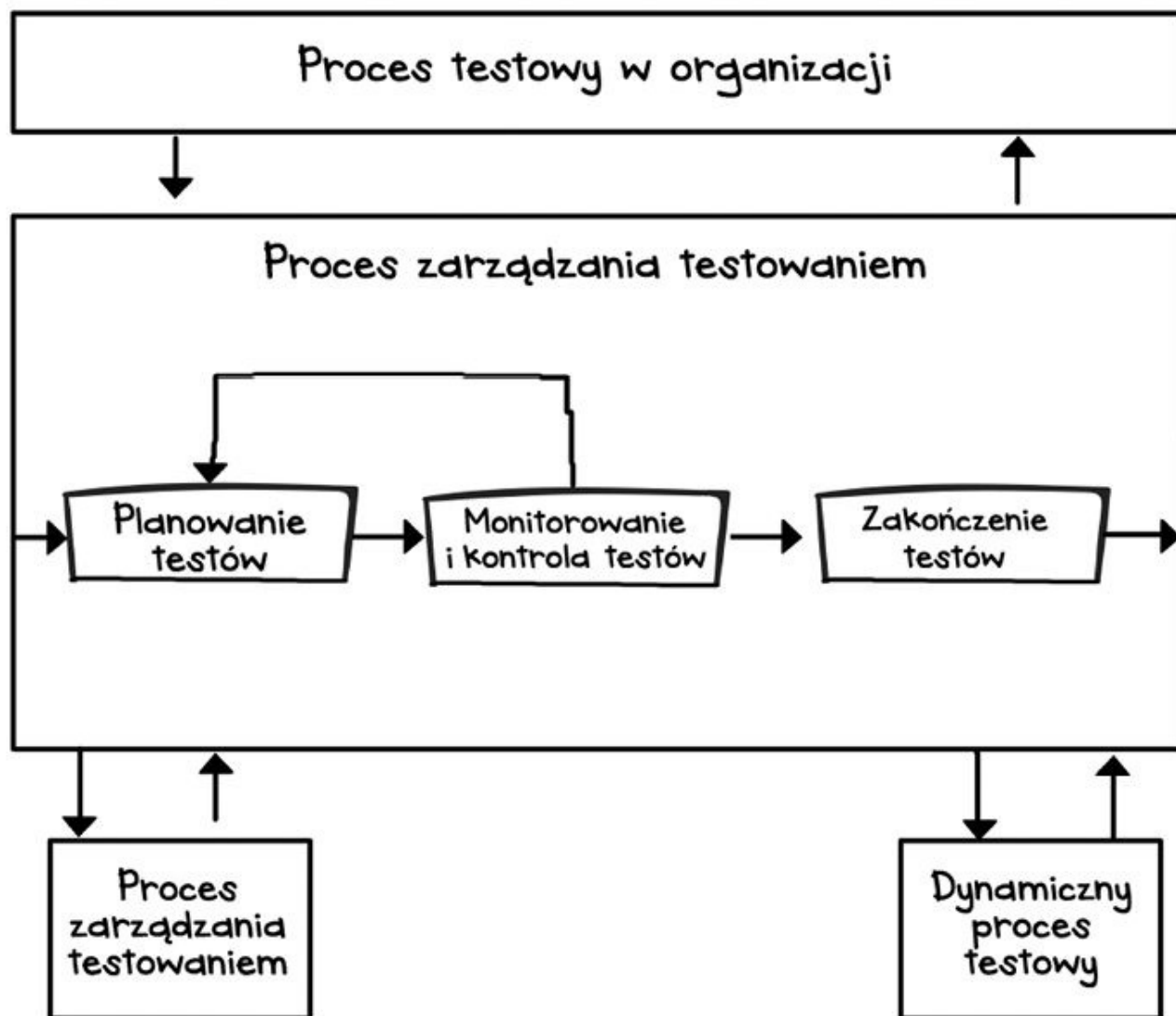
1. „Concept and Vocabulary”, czyli koncepcja i pojęcia słownikowe. Koncepcja testowania odnosi się do wysokopoziomowych aspektów testowania, takich jak wdrożenie testów czy testowanie jako części procesu wytwarzania oprogramowania. Słownik odwołuje się do najpopularniejszych pojęć testerskich ujętych w normach: IEEE 1028 – opisującej proces przeglądu, IEEE 829 – opisującej dokumentację testową, BS 7925-1 – klasycznego słownika testerskiego czy słownika pojęć ISTQB.
2. „Test Process”, czyli część poświęcona elementom składającym się na procesowe podejście do testowania oprogramowania. Tym standardem zajmiemy się szerzej.
3. „Test Documentation” odnosi się przede wszystkim do dokumentacji testerskiej opisanej w IEEE 829.
4. „Test Techniques”, ostatni rozdział, opisuje techniki testowe i w dużej mierze odwołuje się do BS-7925-2.

Jak widać, norma nie próbuje definiować testowania od nowa, lecz korzysta z wiekowych, co prawda, ale wartościowych poprzedników. Pierwszym ciekawym zabiegiem w normie ISO jest podzielenie procesu na trzy warstwy: organizacyjną, zarządczą i dynamiczną, którą możemy również nazwać operacyjną.

Relacje między warstwami są szczegółowo opisane, by wyjaśnić potrzebę definiowania więcej niż jednej warstwy. W obszarze organizacyjnym wytwarzane są dokumenty, których nie definiujemy osobno w każdym projekcie. Są to różnego rodzaju dokumenty wzorcowe, których wypełnienie będzie zalecane bądź wymagane, a także ogólne wytyczne do testowania w organizacji, które osobie zarządzającej jakością w obszarze jednego projektu pozwolą odpowiednio zdefiniować na przykład zakres testowania. W procesie testowania uwzględniono również potrzebę dostosowywania poziomu

organizacyjnego do zmieniających się warunków projektowych. Model procesowy zawiera informację zwrotną na temat jakości i stopnia zastosowania wzorców dokumentów w projekcie. Ma to pozwolić na ciągłe podnoszenie jakości dokumentacji poprzez odpowiednie zarządzanie informacjami wynikającymi z praktycznego użycia wzorców.

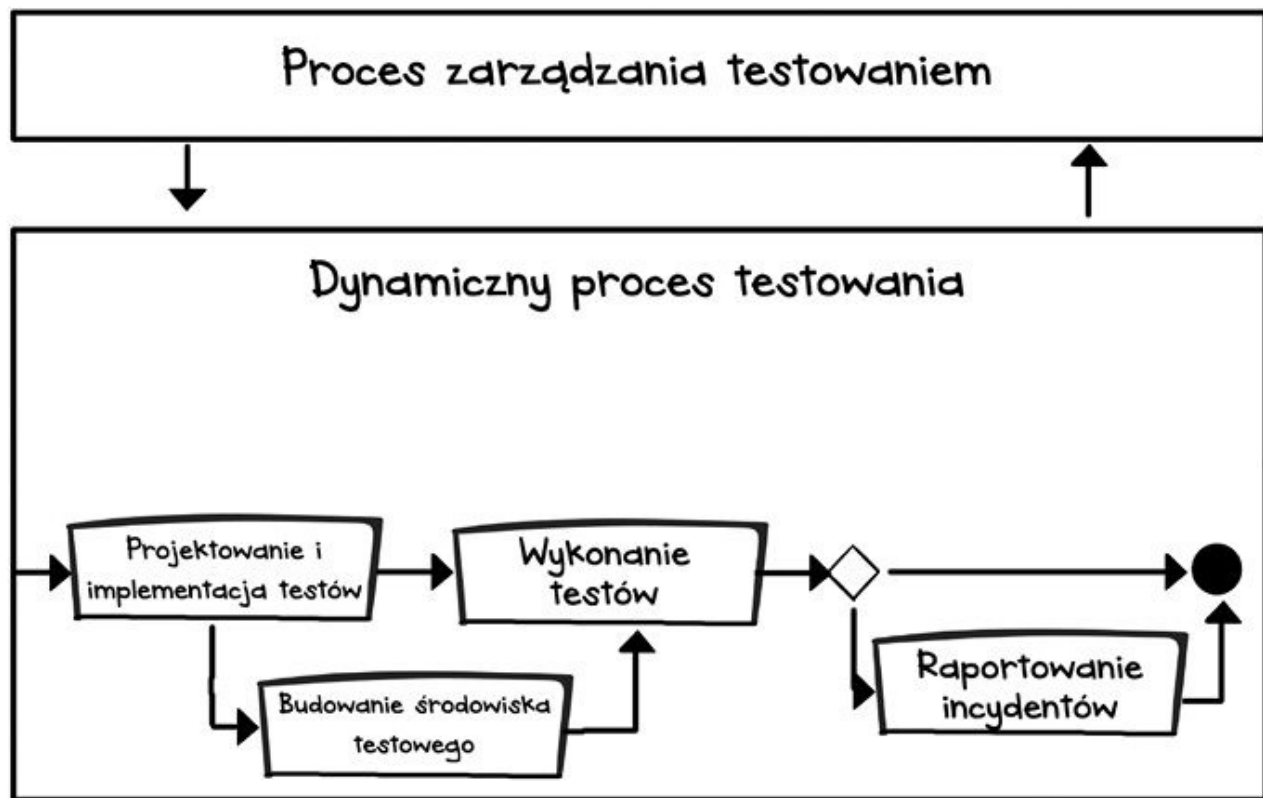
Zarządzanie w procesie jest uproszczone do maksimum i opisane na rys. 4.3. Zawiera jedynie trzy grupy aktywności: planowanie, wspólnie monitorowanie i kontrola oraz określanie zakończenia. Istnieje również określenie informacji zwrotnej z monitorowania, które może wpłynąć na plan testów. Jest oczywiste, że okoliczności projektowe mogą doprowadzić do zmian w planie. Niedostosowywanie dokumentu do nowych okoliczności spowoduje, że planowanie okaże się bezwartościowe i nie będzie stanowić punktu odniesienia np. dla określenia końca testów. Stąd też duża potrzeba dostosowywania harmonogramu testów czy też matrycy ryzyk do aktualnych informacji płynących z projektu. Cały proces ma tylko dwa produkty – plan testów i raport podsumowujący. Jak wynika z praktyki projektowej, jest to zazwyczaj dokumentacja przygotowywana przez kierowników testów. Zarządzanie testowaniem nie ogranicza się do generowania ton papieru, ale polega również na kierowaniu testerami. Sterowanie w tym przypadku jest wspierane informacjami płynącymi z procesu dynamicznego. Zbierane są w nim miary, np. liczba wykonanych przypadków testowych i rezultaty lub informacje o krytyczności defektów. Odpowiedzialnością kierownika jest tutaj przesyłanie odpowiednich dyrektyw kontrolnych. W ten sposób mocno powiązано poziom testowania operacyjnego i zarządzanie testowaniem.



Rys. 4.3. Zarządzanie procesem testowym wg ISO 29119

Najniższy poziom, na którym pracuje większość z nas, to poziom dynamiczny pokazany na rys. 4.4. Tutaj pisze się przypadki testowe i na podstawie analizy dokumentacji definiuje się środowisko testowe. Opis procesu testowego czytelnie pokazuje krytyczne wymagania dotyczące rozpoczęcia samego testowania, do którego wejściem są dwa dokumenty: specyfikacja testowania jako zbiór przypadków testowych oraz raport o przygotowaniu środowiska testowego. Wynikiem wykonania testów jest raport z testów, który z jednej strony może informować o znalezieniu incydentu, a z drugiej – o poprawnej

weryfikacji funkcjonowania oprogramowania. W obu przypadkach informacje te, jak napisano wcześniej, trafiają do kierownika testów.



Rys. 4.4. Dynamiczny proces testowy wg ISO 29119

#### 4.2.4. Proces testowy wg ISTQB

ISTQB jest znane jako organizacja rozwijająca ścieżki certyfikacyjne zapoczątkowane przez ISEB, czyli Information Systems Examination Board. W związku z ograniczoną dostępnością opisów procesów testowych funkcję wyroczni może pełnić proces opisany w podręczniku z kursu przygotowującego do egzaminów na certyfikat ISTQB. Można uznać, że stał się on czymś w rodzaju standardu. Warto podkreślić, że proces opisywany na poziomie podstawowym tego kursu różni się od procesu opisywanego na poziomie zaawansowanym.

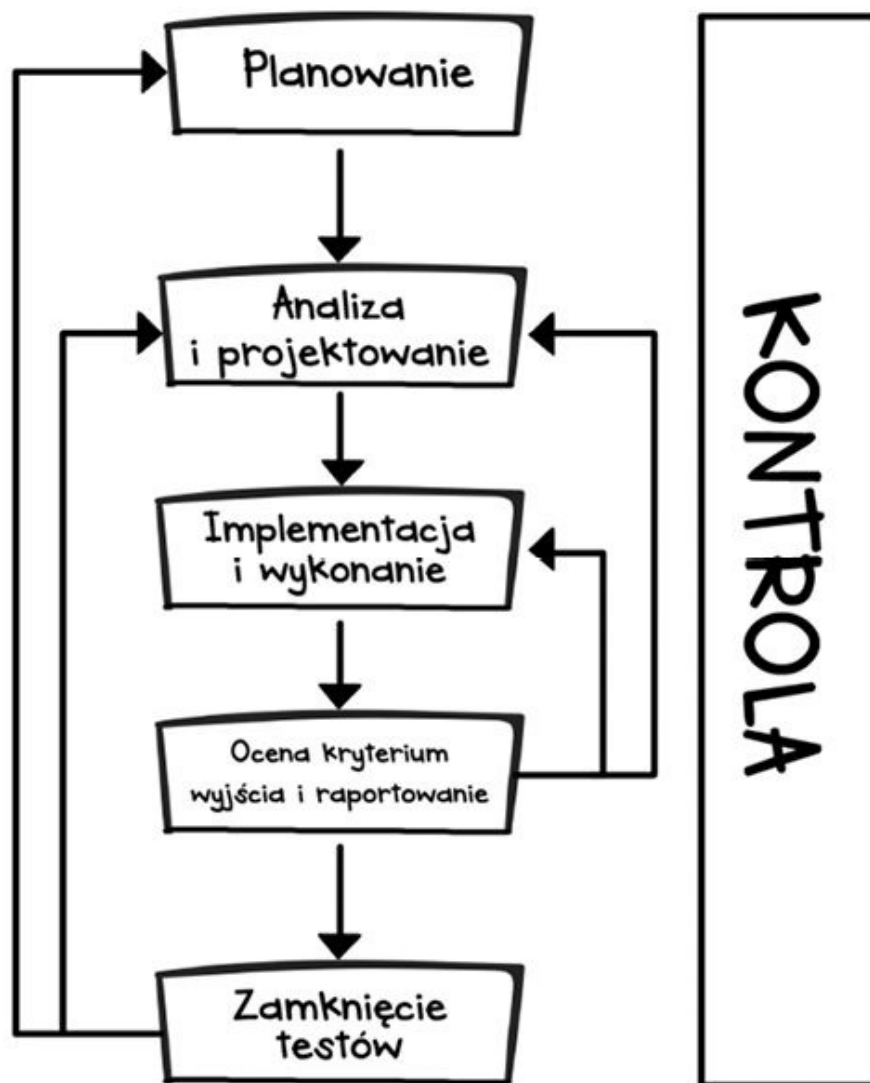
Na proces z poziomu podstawowego przedstawiony na rys. 4.5 składają się następujące czynności:

- planowanie i kontrola,

- analiza i projektowanie,
- implementacja i wykonanie,
- ocena kryterium wyjścia i raportowanie,
- zamknięcie testów.

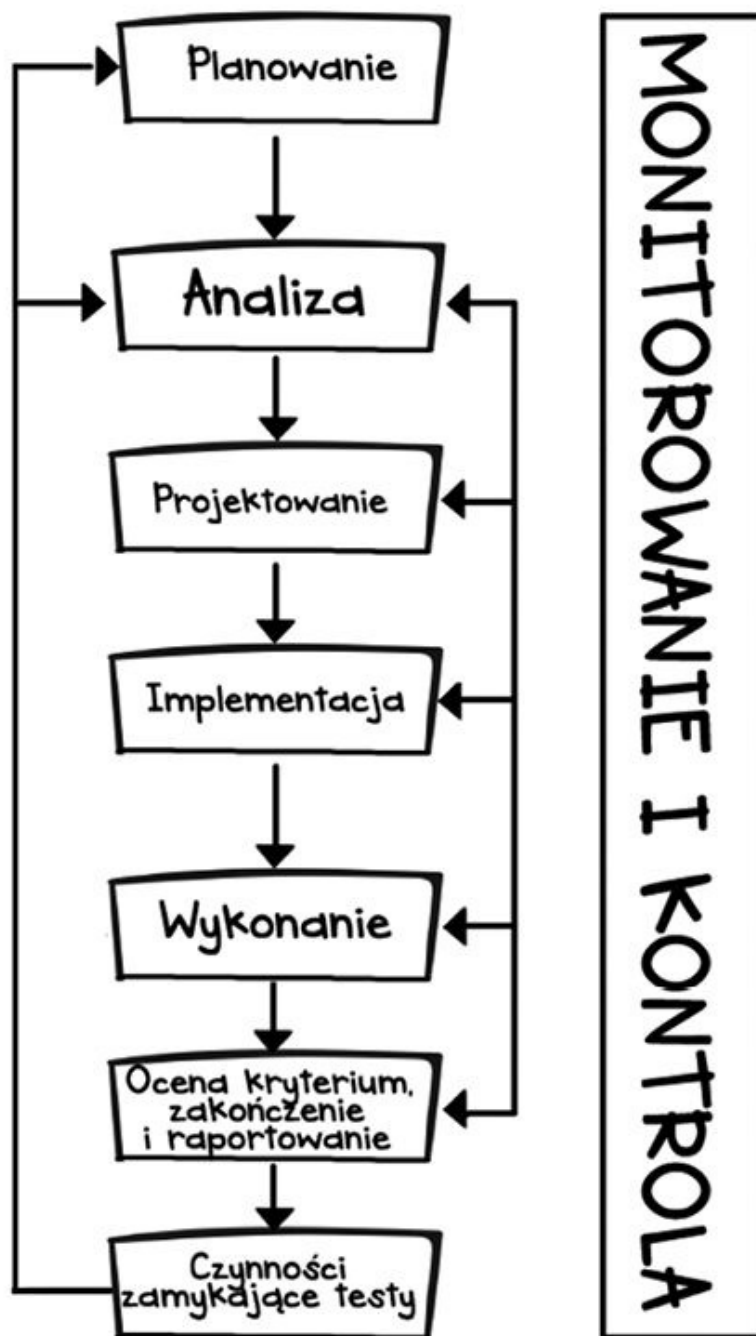
Proces z poziomu zaawansowanego to już modyfikacja, która rozdziela (wcześniej połączone) czynności w ramach procesu. Trzeba przyznać, że nowsze podejście pokazane na rys. 4.6 jest bardziej praktyczne i lepiej oddaje doświadczenia zespołów testerskich.

Zawartość materiałów szkoleniowych jest pewną interpretacją, na którą łatwo się powołać, ale którą równie łatwo podważyć. Dlaczego? Chociażby dlatego, że proces opisany w materiałach ISTQB opiera się na standardzie BS7925-2 opracowanym do celów testów komponentowych i jego szybka implementacja w inne obszary wymaga poważniejszych badań i analiz.



**Rys. 4.5.** Proces testowania zgodny z ISTQB poziomu podstawowego (interpretacja własna)





**Rys. 4.6.** Proces testowania zgodny z ISTQB poziomu zaawansowanego (interpretacja własna)

Tak opisane podejście procesowe ma swoje mocne i słabe strony. Mocną stroną procesu nazywanego przez twórców prostym będzie niewielka liczba kroków z wysokopoziomowym opisem, ułatwiającym aplikację w wielu środowiskach. W opisie podano, że w organizacji są dwie podstawowe role: kierownik testów, który odpowiada za

planowanie, nadzór, definiowanie zakończenia testów, raportowanie i czynności kończące testowanie, oraz tester, odpowiedzialny za proces na poziomie czynności operacyjnych.

Słabe strony mocno wpływają na ogólny odbiór procesu. Trudno między innymi zdefiniować poziom odpowiedzialności za czynności, które nie są jasno powiązane. Szczególnie dla czynności ciągłej w procesie – kontroli – nie zdefiniowano ani momentu zaangażowania się, ani momentu reakcji kontrolnej. Podobnie nie widać w opisie procesu jednoznacznych produktów poszczególnych faz – co musi powstać, a co może być wytworzone.

Dużym problemem dla osób poszukujących rozwiązań jest brak mapowania procesu testowego na wytwarzanie oprogramowania. Gdzie jest wejście produktów z innych procesów, w tym z procesu wytwórczego? W którym miejscu to my przekazujemy swoje produkty? Gdzie są czynności takie jak retest, testy regresywne czy testowanie wymagań? Nie możemy jednak oczekiwać cudów, skoro cały opis procesu liczy trzy strony.

#### **4.2.5. Podsumowanie**

Przedstawione opisy procesów pochodzą głównie ze źródeł teoretycznych i standardów. Trzeba przyznać, że samym procesem testowania zajmuje się niedużo źródeł o praktycznym zacięciu – może dlatego, że w praktyce proces jest dopasowany do działań organizacji i kontekstu samego produktu.

### **4.3. Błędy, defekty, awarie, incydenty, zdarzenia, bugi...**

W tej publikacji często posługuję się interpretacjami i definicjami przekazanymi przez *Słownik wyrażeń związanych z testowaniem*. Nie dlatego, że uznaję go za doskonałe źródło wszelkiego typu definicji, ale dlatego, że potrzebujemy spójności w wyrażaniu się. Możesz, drogi czytelniku, sam definiować pojęcia, ale musisz w takim wypadku wziąć na siebie zadanie zrzutowania moich definicji na twoje. Zawarte w słowniku definicje w następujący sposób pokazują źródło pojawiania

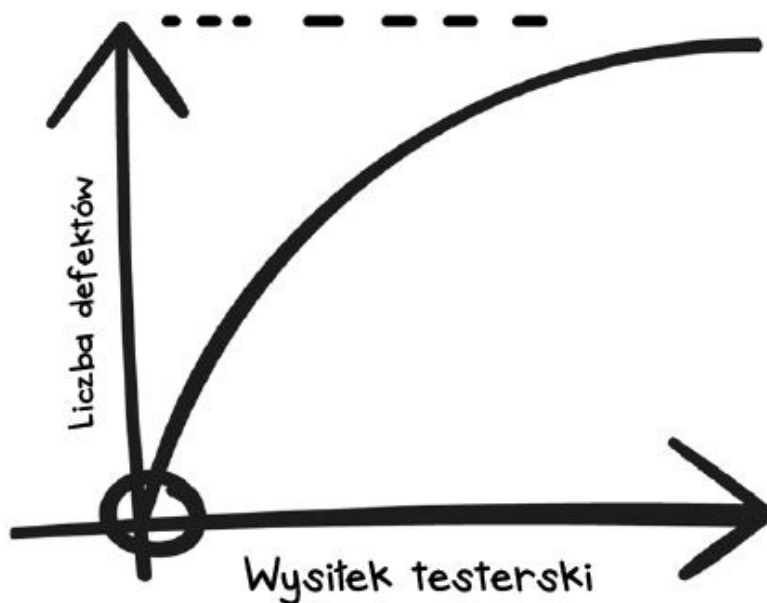
się AWARII oprogramowania. Pierwszym krokiem jest popełnienie przez człowieka BŁĘDU – od złej analizy, przez niepoprawnie skonstruowaną dokumentację, po pomyłki podczas samego programowania. Błędy te wprowadzają do półproduktów w tym do oprogramowania DEFEKTY. Defekty znajdują się również w specyfikacji, a przez to później w kodzie źródłowym i doprowadzają do AWARII oprogramowania podczas jego uruchomienia. Pokazane to zostało na rys. 4.7.

Jest to oczywiście jedna z możliwych nomenklatur testowania. W mediach, w tym w publikacjach prasowych, ludzie, którzy nie zajmują się zawodowo testowaniem, często stwierdzają: „W programie znaleziono błąd”. Zgodnie z tym, co już powiedziano, nie jest to poprawne, bo w oprogramowaniu, zwłaszcza w produkcie końcowym, znajduje się awarie.



**Rys. 4.7.** Cykl od błędu do awarii (interpretacja własna)

Pod względem językowym lepszym określeniem wydaje się „popełniono błąd”. Czy oprogramowanie samo w sobie może popełnić błąd? Raczej nie, poprawność czy niepoprawność jego działania zależy od człowieka, ale „znalezienie błędu” na tyle przyjęło się w języku laików, że nie warto z nim walczyć. My jednak trzymamy się następującego ciągu: błąd → defekt → awaria.



**Rys. 4.8.** Wizualizacja skończonej liczby defektów w oprogramowaniu

Choć podczas testowania nigdy nie znajdzie się wszystkich usterek, zakładamy, że liczba defektów w oprogramowaniu jest skończona, jak pokazano na rys. 4.8. Stwierdzenie to będzie poprawne w odniesieniu do jednej wersji oprogramowania uruchomionego w określonym dla niego środowisku.

Celem testowania jest próba określenia konsekwencji błędów popełnianych przez innych oraz unikanie popełniania własnych. Wiąże się to z podejściem, które może być kontrowersyjne, ale pomaga testerowi w pracy: „Znaleziony defekt jest zazwyczaj dobry, a na pewno jest lepszy od defektu nieznanego po stronie wytwórcy oprogramowania”. Oczywiście zależy nam na tym, aby defektów było możliwie najmniej, ale znaleziony można świadomie naprawić, a nieznanego może (choć nie musi) kiedyś spowodować dużo kłopotów. Ludzie są omylni i zawsze będą popełniali błędy, zadaniem testerów zaś jest wskazywanie konsekwencji błędów. To zajęcie wymaga dużych umiejętności tzw. miękkich, opisanych w kolejnych rozdziałach.

## **Błędy**

Mądrości ludowe w powiedzeniach i przysłowiach oraz sformułowane przez filozofów pomogą nam przyjąć odpowiednie nastawienie do

błędów. Co prawda, żaden z autorów przytoczonych tu zdań na pewno nie myślał o testowaniu oprogramowania, ale ze względu na uniwersalność przekazu mieszczą się one w obszarze naszej pracy:

*Błądzić jest rzeczą ludzką (Errare humanum est) – Seneka Starszy*

*Błądzenie jest rzeczą ludzką, ale dobrowolne trwanie w błędzie jest rzeczą diabelską – św. Augustyn*

*Człowiek, który nie robi błędów, zwykle nie robi nic – Edward John Phelps*

*Dopiero płacąc za błędy, uświadamiamy sobie ich cenę – Sławomir Kuligowski*

*Doświadczenie to nazwa, którą każdy nadaje swoim błędom – Oscar Wilde*

*Jest cechą głupoty dostrzegać błędy innych, a zapominać o swoich – Cyceon*

*Kto mało myśli, błądzi wiele – Leonardo da Vinci*

*Mali ludzie czerpią olbrzymią radość z błędów i gaf popełnianych przez ludzi wielkich – Arthur Schopenhauer*

*Naukowiec jest niczym mimoza, gdy sam popełni błąd, i niczym ryczący lew, gdy odkryje błąd zrobiony przez kogoś innego – Albert Einstein*

*Na wojnie ten wygrywa, kto najmniej błędów popełnia – Napoleon Bonaparte*

*Są ludzie, którzy nie popełniają błędów. To ci, za których myślą inni – Henryk Jagodziński*

*Śpiesz się powoli, a unikniesz błędów – Sofokles*

*Uczmy się na cudzych błędach, bo sami wszystkich popełnić nie zdążymy – Joanna Chmielewska*

*Wartość pomyłki jest jedynie czasami niewątpliwa; nie każdy, kto do Indii płynie, Amerykę odkrywa – Erich Kästner*

Osobną kategorię elementów, jakie raportujemy, są INCYDENTY, niekiedy nazywane ANOMALIAMİ lub po prostu PROBLEMAMI. Czasami po prostu wiemy, że znaleźliśmy awarię oprogramowania, która wymaga naprawienia, czasami jednak nie potrafimy jednoznacznie określić, czy zachowanie aplikacji jest poprawne, czy niepoprawne. Nie możemy sobie oczywiście pozwolić na nieraportowanie o takim zdarzeniu, więc zgłaszamy je jako incydent wymagający dalszego zbadania. Tester może nie wiedzieć, czy ma do czynienia z awarią, czy też problemem, ponieważ:

- specyfikacja jest niejasna lub nie istnieje;

- obszar funkcjonalny lub нефunkcjonalny nie został w pełni zdefiniowany;
- zdarzenie nastąpiło raz lub następuje niedeterministycznie i niełatwo je zreprodukować;
- aplikacja pozornie zachowuje się tak, jak powinna, ale nasz testerski nos mówi nam, że coś jest nie tak;
- inne przyczyny.

Awarie i incydenty zazwyczaj nie mają różnych form zapisu w narzędziach do raportowania defektów. Możemy więc pozostać przy jednym formacie raportowania.

Do wspomnianych pojęć warto jeszcze dodać „sugestie”. W przypadku projektu to bardzo grząski grunt. Sugestie mogą być źle przyjmowane przez innych członków zespołu projektowego, ponieważ główną rolą testera oprogramowania jest potwierdzenie implementacji funkcji, częściowe potwierdzenie (funkcja działa, ale ma defekty) i zanegowanie implementacji funkcji (funkcja nie działa). Natomiast sugestia jest wejściem w inny obszar, czyli próbą definiowania, co warto by – jak nam się wydaje – poprawić w interesie klienta. Problem polega na tym, że sugestia w wielu przypadkach jest oparta na subiektywnym odczuciu. Coś nie zostało w pełni zdefiniowane w specyfikacji i zbyt często dajemy sobie prawo do uznania, że powinno zostać zmienione. Nie zrozum mnie źle, czytelniku, często sugestie są dobre, ale nie dajmy się zwariować. Szukamy defektów, potwierdzamy, że działa, a definiowanie tego, co „warto by zrobić inaczej”, zostawmy odbiorcom końcowym. Chyba że ktoś poprosi nas nie tylko o testowanie, ale również o zasugerowanie zmian. W takim wypadku sugestie są w pełni uzasadnione.

### **4.3.1. Uciekinierzy**

Naszym głównym celem podczas testowania jest niedopuszczenie do sytuacji, w której defekty zostaną zauważone dopiero przez klienta w czasie użytkowania oprogramowania. Defekty takie można nazwać „uciekiniernami” (taka nazwa pojawiła się w blogu Erica Jacobsena [testthisblog.com](http://testthisblog.com)) i spośród wszystkich defektów oprogramowania to właśnie one są naszym największym wrogiem, z nimi przede wszystkim walczymy. To one, mimo pól minowych analizy statycznej, zasieków

wykonywanych testów i obrony testerskiej „przedzierają się” do użytkownika końcowego. Nie ma właściwie dużego znaczenia, czy defekt jest krytyczny, ważny czy trywialny dopóty, dopóki jesteśmy świadomi jego istnienia. Poważne kłopoty pojawiają się, jeśli nie wykryjemy go (ich) przed wydaniem oprogramowania klientowi.

„Uciekinier” stanowi zupełnie inną kategorię defektu, uzupełniając jednocześnie wszystkie nam znane. Pojawi się nie w raportach z testów, ale jako skarga na infolinii, towar zwrócony do sklepu czy żądanie zapłacenia kary umownej. „Uciekinier” ma dużą wagę biznesową. Może powodować utratę dobrego imienia w oczach klienta lub też stratę przy kolejnym produkcie, który znajdzie mniej nabywców.

Nie bez powodu ocenia się zespoły wytwórcze miarą: (liczba „uciekierów”/liczba wszystkich znalezionych defektów) \* 100%. I nie bez powodu każdy wytwórca marzy o liczbie „uciekierów” równej zeru.

Jeśli jednak defekty przedostaną się do klienta, musimy się zastanowić nad gęstością oczek w siatce do ich wychwytywania, nad poprawnością procesu ich eliminowania itp. Czy mogliśmy zrobić coś, co zatrzymałoby „uciekiera”? Jaki wysiłek oznacza to dla naszej organizacji i kto będzie odpowiedzialny za to, aby dany defekt nie wyciekł ponownie?

Dążący do doskonałości tester ma świadomość tego, że część defektów wycieknie jako konsekwencja jego niepełnej wiedzy biznesowej lub braku kompetencji w pewnych obszarach technicznych. W swoim dążeniu do zredukowania liczby uciekinierów uczenie się na popełnionych błędach będziemy traktowali jako najlepszą formę doskonalenia.

Kiedy jednak krytyczny defekt „wycieka” do klienta, pojawia się sytuacja, którą można nazwać kryzysem. Obowiązują więc zasady pokrewne do zarządzania kryzysem. Należy przede wszystkim dbać o odpowiednie i rzetelne informowanie o problemie. Aby nie tworzyć chaosu komunikacyjnego, informacje należy przekazywać przez „rzecznika”. Informacja musi dotyczyć nie tylko obecnej sytuacji, ale również prognoz na przyszłość. Nie wolno unikać deklaracji co do dalszego rozwoju wypadków ani składać pochopnych deklaracji. Podstawowe reguły informowania o problemie są następujące:

- Nie zaprzeczaj. Negowanie tego, że defekt wystąpił, jest reakcją naturalną, ale należy jej się wystrzegać.
- Reaguj szybko. Zanim zareaguje konkurencja czy prasa.
- Pokaż, że nie tylko widzisz problem, ale również rozumiesz jego wagę i konsekwencje.
- Kanały komunikacji utrzymuj otwarte. Informuj o postępach prac. Odpowiadaj na pytania.
- Przepraszaj.
- Powiedz: „To się już więcej nie wydarzy” i zrób wszystko, aby już do tego nie doszło.

Nasza praca polega na szukaniu rozwiązań, które zminimalizują liczbę „uciekierów”. Powinniśmy wdrożyć analizę przyczyny „uciekiera” i możliwości narzędziowe, procesowe i ludzkie jego powstrzymania.

Sam temat raportowania o defektach zostanie szerzej opisany w części praktycznej tej książki.

### **4.3.2. Błędy popełniane przez testerów**

Również testerzy oprogramowania popełniają błędy. Typowy dla nich jest tzw. błąd zakwalifikowania działania poprawnego jako niepoprawne, który pojawia się, gdy mimo braku defektu tester zgłasza raport o incydencie. Ten typ błędów jest uciążliwy dla innych członków zespołu, analityków czy programistów, bo zmusza ich do poszukiwania problemu, którego realnie nie ma.

Innym typem błędu testera jest nieraportowanie o defekcie, tam gdzie on się pojawił. Istnieje wiele źródeł takiego błędu, takich jak zwykła nieuwaga lub niejednoznaczna specyfikacja produktu. Bez względu na powód przypadki takie mogą być krytyczne dla projektu, ponieważ z jednej strony marnują budżet przeznaczony na testy, a z drugiej dają złudne poczucie poprawności działania, podczas gdy w rzeczywistości pojawia się problem.

Błędy te przekładają się w konsekwencji na wspomnianych już „uciekierów”. Doskonalenie testera powinno się zacząć od analizy własnych zgłoszeń uznanych przez innych członków projektu za „niedefekt” i tych błędów, które mieliśmy okazję wychwycić, ale z jakiegoś powodu tego nie zrobiliśmy.



### 4.3.3. Defekty powodują defekty

Czasami zdarza się również, że defekty powodują defekty. Z tym stwierdzeniem wiąże się kilka prawd.

- Niepoprawne zachowanie aplikacji ogólnie możemy nazwać awarią. Nie wnioskujemy, skąd wiemy, że zachowanie jest niepoprawne. Generalnie akceptujemy, że mamy wystarczającą wiedzę, aby uznać coś za „awarię”. Jeśli w aplikacji pojawi się awaria, to może ona mieć wiele konsekwencji dla samej aplikacji, np. może powodować inne awarie. Źródłem awarii jest defekt. O defekcie wynikającym z defektu mówimy wówczas, gdy eliminacja defektu (źródłowego) spowoduje, że również powiązane z nim defekty przestają istnieć. Przykład: niepoprawna walidacja formularza może powodować, że nie uda się go uzupełnić i zakończyć mimo poprawności danych. Usunięcie defektu walidacji powoduje, że defekt formularza (defekt wynikający z defektu) rozwiązuje się automatycznie.
- Analizując oprogramowanie, czasami widzimy dużą zależność między niektórymi defektami. Część tych zależności wynika ze wspólnego źródła pomyłki, część to konsekwencja stosowania techniki programistycznej kopiuj – wklej. Niepoprawny kod jest kopiowany do wielu miejsc (które czasami wydają się zupełnie niepowiązane). Znalezienie wzorca defektu pomaga wyeliminować go z wszystkich miejsc, w których został „wklejony”.
- Poprawki defektów mogą powodować nowe defekty. Dopóki liczba defektów naprawianych jest większa niż liczba defektów wprowadzanych przez poprawki, dopóty możemy mówić o postępie jakości oprogramowania.

Możemy więc mówić o czymś w rodzaju wzorca defektów. Ich świadomość i zdolność wykrywania będzie kolejnym (ważnym) aspektem rozwoju testera oprogramowania.

---

#### Zadanie

Przeprowadź prosty eksperyment poznawczy. W dowolnej aplikacji wskaż awarię i spróbuj udowodnić, że naprawdę jest ona warta naprawienia. Na jakie źródła, autorytety czy dokumenty się powołasz? Jak udowodnisz osobie odpowiedzialnej za dany problem, że awaria musi zostać poprawiona.

---

## 4.4. Jakość oprogramowania a użytkownik

We współczesnym świecie oprogramowanie jest prawie wszędzie, a tam, gdzie go jeszcze nie ma, na pewno wkrótce się pojawi. Postrzeganie oprogramowania i jego znaczenia nie może się obyć bez próby analizy użytkownika i kontekstu jego funkcjonowania. Dobrą próbą zrozumienia użytkownika jest sporządzenie jego opisu, czyli tzw. osoby.

Persona
Persona jest opisem fikcyjnego użytkownika, która twórcom oprogramowania ma pomagać zrozumieć jego potrzeby i oczekiwania względem oprogramowania.

Oto kilka przykładów takich opisów.

Edyta Kowalska – 16 lat, uczy się w szkole publicznej, ma wielu znajomych, a jej główne kanały komunikacji to telefon komórkowy, komunikatory i portale społecznościowe.

*Edyta zaczyna swój dzień o uruchomienia komputera. Jej głównym celem jest sprawdzenie nowych wiadomości w skrzynce pocztowej, nowych informacji i komentarzy w portalu społecznościowym oraz sprawdzenie „dostępności” znajomych. W przypadku niedostępności serwisu społecznościowego (np. z powodu defektu oprogramowania) Edyta ma poczucie oderwania od rzeczywistości i znacznego pogorszenia się standardu jej życia. Zmiana rutyny i nawyku codziennego sprawdzania informacji w serwisie może spowodować depresję oraz uczucie osamotnienia.*

Może dla wielu osób to stwierdzenia to przesada, ale rodzice współczesnych nastolatek oraz nastolatków rozumieją, że nie jest to przypadek odizolowany czy nierealny.

John Smith – lat 23, żołnierz armii amerykańskiej, operator rakiet średniego zasięgu. Wykształcenie średnie, planowane studia inżynierskie.

*Mimo młodego wieku Johnowi powierzono obsługę najnowocześniejszej broni. Podczas kolejnej wojny będzie miał możliwość uśmiercania terrorystów znajdujących się*

*w najodleglejszych rejonach świata. Defekt w oprogramowaniu może oznaczać dla Johna różnicę między zabiciem „złych” a zbombardowaniem szpitala czy sierocińca. Konsekwencje defektu mogą pociągnąć za sobą reperkusje nie tylko dla samej psychiki Johna, ale również złą prasę, nienawiść ludności tubylczej, a w ostateczności – rozszerzenie konfliktu narodowego.*

Czy się z tym zgadzamy, czy też nie, na świecie oprogramowanie służy nie tylko do podtrzymywania życia, ale również do jego skracania. Niedziałanie oprogramowania może się zakończyć uśmiercaniem niewinnych ludzi.

Igor Daszczenko – lat 33, mieszkaniec Moskwy, biznesmen prowadzący 13-osobową firmę kurierską.

*Dla Igora i jego pracowników telefon komórkowy jest podstawowym narzędziem pracy. Otrzymywanie zamówień od klientów i ciągle śledzenie trasy paczek odbywa się dzięki systemowi A-GPS. Dzięki terminowości i ciągłej informacji o przesyłkach firma Igora stale zyskuje nowych klientów. W razie awarii sieci komórkowej Igor traci w każdej minucie. Z jednej strony, nie może obsłużyć zamówień spływających od klientów, z drugiej – jego pracownicy mimo obecności w pracy nie są w stanie wykonywać swojej pracy. Konsekwencje to utrata zysków i zamknięcie biznesu.*

W standardzie person są również zdjęcia, które mają nam pomóc jeszcze lepiej „wczuć się” w danego użytkownika.

Każda z tych „osób” (w lekko przerysowany sposób) zależy od oprogramowania. Większość z opisanych przypadków to tzw. problemy pierwszego świata i może być nie do pojęcia dla ludzi pamiętających czasy bez komputera. Nie zmienia to faktu, że wraz ze zwiększającą się penetracją naszej rzeczywistości przez oprogramowanie uzależniamy się od poprawnie działającego i dostępnego oprogramowania.

Testowanie jest więc procesem złożonym i jednym z narzędzi ochrony użytkownika przed defektami i awariami. Ocena jakości, której dokonujemy, nie może pomijać użytkownika. Jakość jest subiektywna i jej odczucie zależy od osoby, która z oprogramowaniem ma styczność. Zdanie testera zawsze będzie miało mniejszą wartość niż zdanie potencjalnego lub rzeczywistego (reprezentatywnego) użytkownika.

## **4.5. Czym jest testowanie?**

Rozprawmy się z kilkoma mitami testowania i znajdziemy odpowiedź na kilka pytań. Jak pokazałem we wcześniejszym rozdziale, nie ma uniwersalnej definicji testowania, a droga, jaką obierzemy, musi zależeć od tego, dokąd zmierzamy. Dalej podaję kilka istotnych elementów pomagających znaleźć własną drogę w testowaniu lub zaakceptować jedną z proponowanych przez ekspertów i mądre głowy.

#### 4.5.1. Proces oraz zapewnienie jakości

Testowanie to jedna z wielu czynności (podprocesów) w procesie wytwarzania oprogramowania, która ma pomóc zapewnić najwyższą jakość dostarczanego produktu. Mnogość definicji przesuwają samą czynność testowania w zupełnie inne obszary. Załóżmy dwie propozycje definicji testowania.

Definicja wąska: *Testowanie jest sprawdzeniem poprawności i zgodności implementacji oprogramowania z założeniami spisanyymi w formie dokumentacji lub z założeniami zlecającego wytworzenie oprogramowania.*

Definicja szeroka: *Testowanie zapewnia poprawność i zgodność implementacji oprogramowania z założeniami spisanyymi w formie dokumentacji lub też z założeniami zlecającego wytworzenie oprogramowania przez sprawdzenie poszczególnych lub wszystkich produktów i półproduktów procesu wytwarzania oprogramowania.*

Jak widać, definicja wąska jest częścią definicji szerokiej, uwzględniając testy jedynie końcowego produktu oprogramowania. Definicja szeroka jest już znacznie bliżej definicji zapewnienia jakości w procesie wytwarzania oprogramowania. Zakładamy jednak, że istnieje pewna ważna granica oddzielająca te dwa zestawy czynności. Testowanie oprogramowania zapewnia jakość na poziomie pojedynczego projektu lub produktu. Zapewnienie jakości rozumiane jako tłumaczenie angielskiego terminu *quality assurance* (QA) zapewnia jakość w całej organizacji. W wielu publikacjach pojawi się również stwierdzenie, że na zapewnienie jakości oprogramowania składają się czynności raczej prewencyjne niż wykrywające defekty, z kolei testowanie oprogramowania bardziej stawia na wykrywanie niż zapobieganie. Praktyk testowania oprogramowania może jedynie z politowaniem pokiwać głową na takie sformułowanie. Procesy QA i testowanie mają zarówno cele detekcyjne, jak i prewencyjne i możemy

spokojnie przyjąć, że testowanie zawiera się w procesach zapewnienia jakości (rys. 4.9).

Możemy więc założyć, że zapewnienie jakości jest pojęciem o wiele szerszym od testowania nazywanego również kontrolą jakości (*quality control*). W organizacjach mniej formalnych lub niedojrzałych procesowo te dwa elementy są traktowane jako jeden. Choć określenia „nieformalny” oraz „niedojrzały” mogą brzmieć pejoratywnie, należy uwzględnić, że same w sobie nie mają takiego charakteru. Decyzja o redukcji formalizmów albo świadoma rezygnacja z podejścia procesowego jest jedną z możliwych metod prowadzenia projektu.



**Rys. 4.9.** Testowanie a zapewnienie jakości

Zapewnienie jakości ma „wyższe cele” od testowania. Kiedy testowanie koncentruje się na pojedynczym produkcie, zapewnienie jakości ma wysokopoziomowo dbać o jakość wszystkich wytwarzanych produktów.

Możemy wyróżnić następujące elementy typowe dla zapewnienia jakości:

- definiowanie i poprawianie procesów,
- pilnowanie zgodności działań projektowych z procesem,

- definiowanie miar do oceny procesu i mierzenie jakości,
- brak kontroli jakości produktu końcowego,
- odpowiedzialność inżyniera (kierownika/menedżera) zapewnienia jakości.

Z kolei testowanie:

- koncentruje się na dostawie oprogramowania,
- szuka defektów,
- waliduje produkt pod kątem zgodności z wymaganiami klienta,
- należy do zakresu odpowiedzialności testera.

Przy jawnym rozdzieleniu tych czynności oczywiste jest, że testowanie może zgłaszać uwagi do procesu, ale samo nie redefiniuje procesu i zazwyczaj nie ma wpływu na jego ostateczny kształt. Trudno też zgodzić się ze stwierdzeniem, że testowanie ulepsza produkt. Nawet jeśli tak, to nie bezpośrednio. Na pewno pozwala znaleźć defekt, zbudować zaufanie (albo je podważyć) do produktu. Niestety bezpośrednio, jako metoda detekcyjna, nie wpływa na zapobieganie defektom. Dopiero analiza zgłoszeń testerskich może wskazać obszary poprawy lub też pomóc innym członkom zespołu uczyć się na własnych błędach. Zestawienie podstawowych cech i czynności dla testowania i zapewnienia jakości pokazano w tab. 4.1.

Testowanie, podobnie jak Ziemia, nie jest płaskie. Jest wielowymiarowe, a jego poszczególnych składowych możemy wyróżnić wiele:

- wymiar czasowy – ile mamy czasu na testowanie;
- wymiar finansowy – jakie mamy środki na finansowanie testowania;
- wymiar produktowy lub półproduktowy – co składa się na końcowy produkt, jaki produkt testujemy i w jaki sposób;
- wymiar ludzki – w jaki sposób pozyskujemy siły do testowania;
- inne.

Wracając do szerokiej definicji testowania oprogramowania, pokrywa się ona z tzw. trójkątem jakości lub też trójkątem projektowym pokazanym na rys. 4.10. Na rogach trójkąta znajdują trzy elementy: czas

(jak szybko?), pieniądz (jak tanio?) oraz jakość (jak dobrze?). Czasami element „jakość” zastępowany jest przez „zakres” lub „funkcjonalność”.

Czarna kropka na prezentowanym rysunku jest produktem. Miejsce i przemieszczanie się kropki w trójkącie traktujemy jako wytyczne dotyczące produktu i/lub sposobu prowadzenia projektu. Odległość kropki od rogów traktujemy jako wymiar (wielkość) wartości dla danego rogu.

Im bliżej rogu oznaczonego jako czas (rys. 4.11) znajduje się produkt, tym szybciej jesteśmy go w stanie dostarczyć, oznacza to jednak, że koszt jego wytworzenia musi być znaczny.

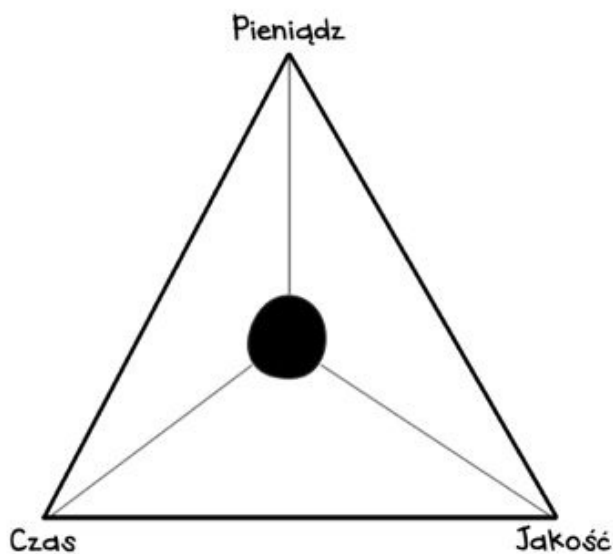
Im bliżej rogu oznaczonego jako pieniądz znajduje się produkt, tym taniej jesteśmy go w stanie dostarczyć, ale potrzebujemy na to więcej czasu (rys. 4.12).

Im bliżej rogu oznaczonego jako jakość znajduje się produkt, tym wyższej jakości (uwaga: inaczej niż w wypadku pozostałych) jesteśmy go w stanie dostarczyć (rys. 4.13). To z kolei wymaga od nas większego nakładu czasu i pieniędzy.

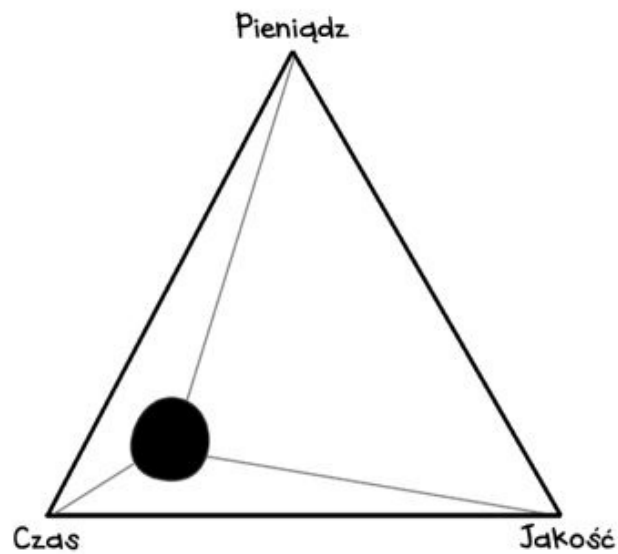
**Tab. 4.1.** Zestawienie czynności i cech testowania oprogramowania i zapewnienia jakości oprogramowania

Testowanie oprogramowania	Zapewnienie jakości oprogramowania
Zapewnienie jakości pojedynczego produktu w ramach jednego lub wielu projektów	Zapewnienie jakości produktu na poziomie procesu
Odpowiedzialność testera oprogramowania lub kierownika do spraw testów	Odpowiedzialność inżyniera lub kierownika zapewnienia jakości
Czynności operacyjne na poziomie wytwarzania oprogramowania	Czynności definiowane na poziomie organizacyjnym
Monitorowanie na poziomie produktu	Monitorowanie na poziomie procesu, standardu lub modelu
Przed rozpoczęciem projektu: szczątkowe czynności weryfikujące testowalność założonych wymagań	Przed rozpoczęciem projektu: rozbudowane czynności definiujące proces wytwarzania oprogramowania i inne procesy z nim powiązane w tym: definiowanie czynności, wzorców dokumentacji projektowej
Przed rozpoczęciem projektu: szczątkowe czynności prewencyjne mające na celu wykrycie defektów	Przed rozpoczęciem projektu: pełne czynności prewencyjne mające na celu zapewnienie wiedzy

typowych dla danej organizacji lub dla danego typu projektu/produktu	i umiejętności uczestników procesu w wytworzeniu oprogramowania
W czasie trwania projektu: prewencyjne działania mające na celu wyszukiwanie defektów, zanim zostaną przeniesione z wymagań do innych dokumentów	W czasie trwania projektu: prewencyjne sprawdzenie, w jakim stopniu projekt posuwa się do przodu zgodnie ze zdefiniowanym wcześniej procesem
W czasie trwania projektu: weryfikacja poprawności implementacji poszczególnych produktów i półproduktów (np. poprawność implementacji względem wymagań)	W czasie trwania projektu: weryfikacja zgodności poszczególnych produktów i półproduktów z procesem (np. zgodność dokumentu wymagań z dokumentem wzorcowym)
Po zakończeniu projektu: czynności zamykające testowanie i podsumowujące jakość wytworzonego oprogramowania	Po zakończeniu projektu: czynności zamykające projekt i podsumowujące jego zgodność z procesem analiza wyniku projektu (zakończony sukcesem, częściowym sukcesem lub zakończony niepowodzeniem) i usprawnienie procesów dla dalszych projektów

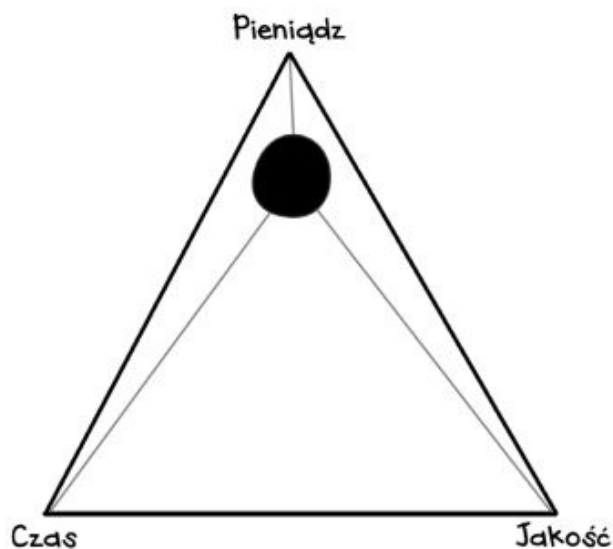


**Rys. 4.10.** Projektowy trójkąt jakości

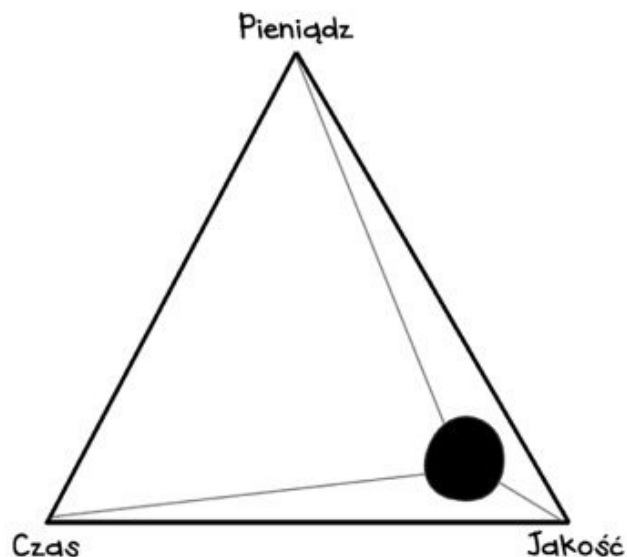


**Rys. 4.11.** Projektowy trójkąt jakości – czas





**Rys. 4.12.** Projektowy trójkąt jakości – pieniądz

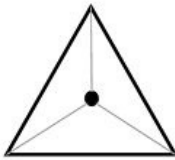
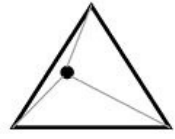


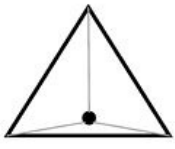
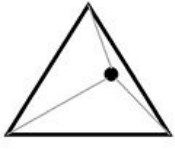
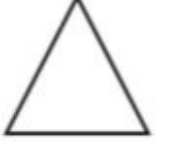
**Rys. 4.13.** Projektowy trójkąt jakości – jakość

W tab. 4.2 zawarto przykładową analizę umiejscowienia produktu (kropki) w trójkącie.

Wielowymiarowość aspektów do uwzględnienia przekłada się na mnogość poziomów w testowaniu, które omówię w dalszej części tej książki.

**Tab. 4.2.** Przykłady umiejscowienia produktu w trójkącie jakości

Wizualizacja	Czas (jak szybko?)	Pieniądze (jak tanio?)	Jakość (jak dobrze?)	Komentarz
	Szybko	Tanio	Dobrze	Produkt znajduje się w samym środku trójkąta, więc próbujemy osiągnąć równowagę między czasem, ceną i jakością. W nomenklaturze projektowej możemy to nazwać dobrze zarządzanym projektem. Zakładamy, że testowanie będzie optymalne czasowo i kosztowo
	Szybko	Tanio	Nie ma znaczenia	Przy próbie wytworzenia czegokolwiek szybko i tanio zazwyczaj tracimy głównie na jakości. Prawdopodobnie testowanie nie będzie częścią procesu wytwarzania oprogramowania lub też będzie nieformalne

	Szybko	Nie ma znaczenia	Dobrze	Aby wytworzyć produkt szybko i dobrze, będziemy potrzebowali bardzo dużo środków finansowych. Duże środki finansowe przekładają się na wysoką jakość produktu
	Nie ma znaczenia	Tanio	Dobrze	Wytworzenie produktu tanio i dobrze będzie długo trwało. Może się również zdarzyć, że obniżenie środków na projekt zakończy się redukcją wydatków na testowanie
	Brak danych	Brak danych	Brak danych	Gdy zarówno wymagania, budżet, jak i czas na wytworzenie produktu nie są nam znane, nie jesteśmy w stanie założyć, co uda się osiągnąć na poziomie produktu

#### 4.5.2. Weryfikować a walidować

Wąska definicja testów opisana w podejściu procesowym na kolejnym etapie może zostać rozdzielona na dwie części. Często mówimy więc o testowaniu jako o weryfikacji jakości i/lub walidacji wymagań.

Rozróżnienie tych dwóch pojęć opisywane jest w wielu miejscach, ale najszerzej akceptowane definicje pojawiają się w normie ISO 9000.

##### Weryfikacja i walidacja

**Weryfikacja** – proces kontroli polegający na sprawdzeniu, czy produkty danego etapu produkcji spełniają zdefiniowane warunki.

**Walidacja** – określenie poprawności produktów procesu tworzenia oprogramowania pod względem spełnienia potrzeb i wymagań użytkownika.

W uproszczeniu możemy powiedzieć, że weryfikacja to sprawdzenie, czy zbudowaliśmy produkt taki, jaki sobie założyliśmy, a walidacja – czy produkt ten jest zgodny z tym, czego chciał klient. Choć wydaje się, że te dwa pojęcia powinny być zbieżne, w wielu wypadkach okazują się różne. W rezultacie producent wytwarza wysokiej jakości produkt, ale niestety zupełnie różny od tego, czego chciał klient.

W literaturze i w szeroko pojętym internecie znajdziemy wiele

w literaturze i w szeroko pojętym internecie znajdziecie wiele źródeł, które albo odwracają te definicje, albo całkowicie je zmieniają. Po raz kolejny warto podkreślić, że jest to propozycja słownikowej definicji, którą możecie adaptować do własnych środowisk i doświadczeń.

### 4.5.3. Szkoła defektów kontra szkoła jakości

Istnieje wiele szkół testowania. Większość z nich operuje na skończonej liczbie dogmatów testowania, więc ich podejście, mimo różnych nazw, będzie bardzo podobne. Część dogmatów będzie negowana z zasady. Negacja staje się tu narzędziem dyskredytacji innej szkoły testowania. Niektóre grupy czy społeczności testerskie dość fundamentalistycznie podchodzą do prawd, które wyznają. Możemy wyróżnić szkoły testowania silnie nakierowane na jedną z wielu czynności testowania, ale też takie, które uznają wiele strategii testowania. Możemy je jednak sklasyfikować na wysokim poziomie jako dwie szkoły:

- Szkoła defektów: testujemy, aby znajdować defekty, więc naszym głównym celem jest udowodnienie, że w jakimś obszarze lub całościowo oprogramowanie nie działa.
- Szkoła jakości: testujemy, aby pokazać, że działa – defekty są jedynie produktem ubocznym.

Można wykazać, że szkoła defektów jest bardziej typowa dla odbiorców oprogramowania, a szkoła jakości jest charakterystyczna dla jego wytwórców. W zależności od naszego celu (lub celu naszej organizacji) testowanie będzie wyglądało inaczej.

Szkoła defektów jest podejściem, które łączy zarówno ISTQB (ciało certyfikujące testerów oprogramowania, o którym więcej później), jak i Michaela Boltona z jego Rapid Software Testing. Według tego ostatniego *We don't test to find out if something works. We test to find out if it doesn't work*, czyli nie testujemy, aby pokazać, że coś działa, testujemy, aby się przekonać, że coś nie działa. Podejście to ma swoje zalety i wady. Oprogramowanie ma zawsze jakieś defekty, więc jeśli odpowiednio długo będziemy je testowali, to udowodnimy w końcu, że przynajmniej częściowo nie działa. Czy jednak ciągle wątplenie i negacja się opłaca? To zależy. Firmom oferującym testowanie oprogramowania jako usługę opłaca się pokazywać słabą jakość. Klienci,

którzy z nich korzystają, chcą słyszeć, że jakość oprogramowania jest niska. Jeśli przekazuje się informację, że defektów nie ma, to klient uznaje, że praca nie została dobrze wykonana.

Szkoła jakości uznaje, że każde oprogramowanie ma jakość. Wątpliwość, która się pojawia to to, jak bardzo przymykamy oko na złą jakość, aby dostrzec zalety. Takie podejście ma więcej zalet, gdy dotyczy tzw. klienta wewnętrznego, który nie lubi złych wiadomości. Chce wiedzieć, że wszystko działa, a ewentualne zgłoszenia to tylko rzeczy, które można ulepszyć.

Miedzy tymi szkołami jest też szkoła neutralna, która nie robi założeń, a jedynie dostarcza informacji o jakości. Takie pozbawione emocji podejście można by nazwać zimnym profesjonalizmem w pracy. Z doświadczenia jednak widać, że nie jest to jednak cecha pożądana przez pracodawców czy szefów projektów. Neutralność może być postrzegana jako nijakość. A przecież chodzi nam przede wszystkim o jakość.

#### **4.5.4. Testy automatyczne**

Na szczęście dla testerów manualnych samotestujące się oprogramowanie jest ciągle pojęciem z nieokreślonej przyszłości. Gdy pojawi się oprogramowanie, które będzie mogło się testować samo, bez pomocy człowieka, nasz zawód umrze, tak jak już dziś umierają zawody wypierane przed produkcję maszynową. Już teraz automaty wykonują dwa typy testów:

- analizę statyczną, czyli proste sprawdzenie oprogramowania pod kątem reguł zdefiniowanych i ustandaryzowanych w danych obszarach;
- testowanie automatyczne, opierające się na uruchomieniu skryptów testowych (napisanych lub wygenerowanych).

Przeprowadzanie takich testów ma swoje zalety i wady, które przekraczają zakres tej publikacji. Temat jest interesujący, ale rzadko trafia do początkujących testerów. Eksperci testowania oprogramowania, tak jak nie potrafią określić ostatecznie definicji testów, tak też nie wypracowali wspólnego stanowiska na temat automatyzacji. Czy przy dzisiejszym poziomie wiedzy to jest opłacalne?

Czy automatyzacja jest w stanie zastąpić albo choć znacząco uzupełnić testy wykonywane manualnie? Gdzie kończy się poziom opłacalności automatycznego testowania oprogramowania? Te pytania w wielu przypadkach nie doczekały się jeszcze odpowiedzi, choć bez kłopotu znajdziemy ludzi, którzy twierdzą, że ją znają. Jedni bez mrugnięcia okiem powiedzą, że automatyzacja się zawsze sprawdzi. Drudzy zanegują tę prawdę, twierdząc z całą pewnością, że tylko testy manualne są realnie opłacalne i mogą przynieść korzyść. Prawda jak zawsze najprawdopodobniej leży pośrodku.

James Bach i Michael Bolton opisali, jak należy rozgraniczać testowanie i automatyzację (sprawdzenie). Oto esencja ich wniosków.

- Testowanie i używanie narzędzi to dwie podstawowe cechy człowieka.
- W każdej dziedzinie i obszarze narzędzia, których używamy, zmieniają nasze otoczenie i to, jak pracujemy i jak żyjemy.
- Narzędzia używane są również w testowaniu, co stanowi wyzwanie dla każdego testera.
- Zwiększona presja na szybkie testowanie zmuszają do powszechniejszego użycia narzędzi.
- Testowanie oprogramowania jest czynnością wykonywaną przez człowieka-testera, programowanie zaś przez programistę.
- Sprawdzenie jest działaniem, które narzędzie testowe może wykonać. Tak jak kompilowanie jest wykonywane przez narzędzie programistyczne.
- Nikt nie mówi o manualnym i automatycznym programowaniu. Jest programowanie i są działania wykonywane przez narzędzia. Tego, co robią narzędzia do programowania, nie nazywamy już programowaniem.

Proponują oni inne definicje, które możemy wrzucić do koszyka z poznanymi już wcześniej definicjami testowania.

*Testowanie – ocenianie produktu poprzez uczenie się go w drodze eksperymentu, uwzględniające: odpytywanie, studiowanie, modelowanie, obserwację oraz interferowanie.*  
*Sprawdzanie – ocenianie przez zastosowanie algorytmicznych reguł decyzyjnych do konkretnych obserwacji produktu.*

Wynika z tego, że testowanie jest pojęciem szerszym i obejmuje

W tym z tego, że testowanie jest pojęciem szerszym i obejmuje sprawdzanie będące *de facto* tylko podzbiorem testowania (rys. 4.14). Z kolei sprawdzanie jest wykonywane przez narzędzie, a testowanie może być jedynie wspierane narzędziami.

Sprawdzanie wykonywane przez ludzi jest więc próbą podążania za pewnym procesem algorytmicznym, natomiast gdy sprawdzanie wykonują maszyny, tym procesem są narzędzia. Jeśli każesz człowiekowi wykonywać zestaw instrukcji niemożliwych do ukończenia, w którymś momencie zmieni swoje działania albo zrezygnuje. Maszyna tego nie potrafi.

Na rys. 4.14 tłumaczenie zaproponowanej przez obu autorów wizualizacji testowania (*testing*), sprawdzania wykonywanego przez człowieka (*human checking*), sprawdzania wykonywanego przez maszynę (*machine checking*) oraz obszaru wspólnego, w którym człowiek i maszyna współpracują przy sprawdzaniu (*human/machine checking*). O automatyzacji napiszę więcej na kolejnych stronach.



**Rys. 4.14.** Sprawdzenie a testowanie wg Bacha i Boltona

---

## Zadanie

Czy organizacja, w której pracujesz albo chcesz pracować, odróżnia zapewnienie jakości od testowania? Czy odróżnia testowanie od automatycznego sprawdzania? Zrób analizę, która pozwoli ci zrozumieć, gdzie się myli w swojej ocenie testowania. Możliwe, że nie masz jeszcze projektu informatycznego, który możesz ocenić. Czasami wystarczy

przejrzeć ogłoszenia o pracy, aby poznać wymagania wobec testera w danej firmie albo sprawdzić, czy w poszukiwaniu „specjalisty ds. zapewniania jakości” nie umieszcza się zadań typowo testerskich.

---

## 4.6. Testowanie jest potrzebne

Dzięki wczesnemu znajdowaniu defektów w oprogramowaniu jesteśmy w stanie je usunąć, zanim wykryje je użytkownik. Dzięki testowaniu, którego wynikiem jest niezastawianie defektów, możemy powiedzieć – z prawdopodobieństwem wprost proporcjonalnym do wkładu w testowanie – że defektów w danym obszarze nie ma lub są głęboko zaszyte i ryzyko, że użytkownik je znajdzie, jest niewielkie. Istnieje natomiast ryzyko, że użytkownik doprowadzi do sytuacji nieprzewidzianej przez testera lub niemożliwej do odtworzenia w środowisku testerskim. W takim przypadku pozostaje nam wiara, że aplikacja zachowa się poprawnie, a możliwe defekty będą odpowiednio obsłużone.

### Liczba defektów w oprogramowaniu

W danej wersji oprogramowania istnieje skończona liczba defektów. Niestety nigdy nie wiemy, ile ich jest. Zakłada się również, że eliminacja pięciu defektów powoduje (średnio) jeden nowy defekt, o wadze zazwyczaj mniejszej lub równej wadze najpoważniejszego z usuniętych defektów. Część praktyków testowania twierdzi, że dziesięć poprawek defektów to jeden (pewny) defekt krytyczny. Wyliczenia są oparte raczej na doświadczeniu niż matematyce, więc należy do nich podchodzić z dystansem. Jest to jednak wiedza, która w pewnych okolicznościach może się okazać przydatna.

Testowanie jest towarzyszem całego procesu wytwarzania oprogramowania. Jego obecność zazwyczaj wiąże się z następującymi podejściami:

- **Przyzwyczajenie.** W organizacjach, w których testowanie jest obecne od początku, nikt nie zadaje sobie pytania, po co testować. Z jednej strony, jest to korzystne, bo weryfikujemy jakość oprogramowania. Z drugiej strony jednak, nie jesteśmy świadomi

kosztów i zysków z testowania, co pociąga za sobą brak optymalizacji kosztów i brak próby zwiększenia efektywności.

- **Wymuszenie.** Zdarza się, że jako wykonawcy jesteśmy zmuszani przez sponsorów lub klientów do prowadzenia testów. Ich jakość, jeśli testujemy tylko dlatego, że jesteśmy do tego przymuszeni, jest zazwyczaj niska.
- **Świadomość i znaczenie zapewnienia jakości.** Organizacje dojrzałe testersko to takie, które nie tylko testują, ale również wiedzą, dlaczego to robią. Każdy pracownik takiej organizacji jest świadomy nakładów przeznaczonych na testowanie i potrafi pokazać przydatność swojej pracy. Każdego dnia eliminuje się defekty znajdowane przez testerów i innych uczestników procesu wytwórczego i mierzy się opłacalność testowania wartością takiego defektu. Może jest to parabola, ale w świecie IT są organizacje, które przy dobrym wdrożeniu narzędzi zarządczych stosują taki model.

Bez względu na powód testowania może ono dawać poczucie bezpieczeństwa dzięki dostarczonym miarom. Wiemy, że naszym głównym celem jest wytworzenie produktu bezawaryjnego, ale znając realia, rozumiemy, że to niemożliwe. Staramy się więc zrobić wszystko, by uzyskać jak najwięcej informacji na jego temat, zarówno pozytywnych, jak i negatywnych.

### Informacja z testowania

W większości przypadków testerzy jako produkt swojej pracy dostarczają informację i ocenę jakości. Informacja taka nie jest oferowana jako produkt na wolnym rynku (są drobne wyjątki, np. firmy zarabiające na testowaniu produktów innych wytwórców lub organizacje przyznające certyfikaty produktom).

Czasami mówi się więc, że testowanie nie wnosi wartości dodanej do produktu, w przeciwieństwie do programowania, które produkt tworzy. Przykład: jeśli słabym programistom damy dowolny budżet, to bez względu na jakość ich pracy i tak na końcu powstanie „jakiś” produkt. Może nie działać, być niestabilny i niewydajny, ale jeśli mamy dobry marketing lub dominującą pozycję na rynku (patrz Microsoft i Windows Vista), to i tak go sprzedamy. Testerzy, dostarczając jedynie informację, mają mniejsze możliwości komercyjnego oferowania swojego produktu.



Konieczność testowania wynika z ludzkiej omyłności i dopóki ludzie będą omylni w trakcie wytwarzania oprogramowania albo dopóki oprogramowanie będzie tworzone przez ludzi, dopóty niezbędne będzie testowanie rozumiane jako weryfikacja oprogramowania przez człowieka.

## 4.7. Testowanie jest nieskończone

Oczywiście wszystko ma koniec (a kij ma nawet dwa końce). W testowaniu jednak przyjęliśmy jako dogmat, że testowanie nigdy się nie kończy.

Swego czasu rozmawiałem prywatnie z kolejnym Kierownikiem Projektu o wyższości testowania nad nietestowaniem. Konwersacja przebiegała mniej więcej tak:

On: *To kiedy się kończy testowanie?*

Ja: *Nigdy.*

On: *Ha! Więc „my” już wydajemy produkt, a „wy” ciągle go testujecie?! (ze złośliwym uśmiechem na twarzy)*

Ja: *„My” już raczej nie, ale „Oni” tak...*

On: *Jacy Oni?*

Ja: *Użytkownicy.*

On: *Czyli jeśli produkt jest ciągle testowany, to ciągle musi być poprawiany.*

Ja: *Tak i nazywa się to utrzymaniem.*

On: *Czy ta pętla się kończy?*

Ja: *Tak, kiedy produkt zostanie wycofany.*

On: *Wy, testerzy, zawsze umiecie się ustawić.*

„Oni”, stosując retorykę Kierownika, to pośredni uczestnicy projektu, tacy jak klienci, użytkownicy, zlecający itd. Tym, co zwraca uwagę, jest podejście części osób z obszaru zarządczego, które posługując się określeniami „my” i „wy”, dzielą członków zespołów projektowych na COŚ wytwarzających oraz innych, m.in. testujących. Udało się to częściowo wyeliminować przez podejście zwinne w tworzeniu produktów. Pozwala ono usunąć podziały i pokazać, że programista, tester, analityk, szef projektu i końcowy odbiorca stanowią zespół, któremu zależy, by produkt pojawił się na rynku.

Brak zdefiniowanego kryterium zakończenia testowania pozwala

nam powiedzieć, że testowanie się nie kończy, więc i koszt jest nieskończony. Kontynuując ten tok myślenia, dochodzimy do stwierdzenia, że cena testów jest tak wysoka, tak niewyobrażalna, że testowanie staje się bezcenne. Jak „Mona Lisa”. Po raz kolejny to poważna przesada, ale nie da się ukryć, że jesteśmy w stanie wykorzystać każdy przekazany nam budżet. Jednocześnie, jak pokazaliśmy wcześniej (i co będziemy powtarzali w całej tej publikacji), nie jesteśmy w stanie dać żadnych gwarancji. Niestety, w prawdziwym świecie budżet nigdy nie jest nieskończony. Co więcej, w praktyce jest zazwyczaj niesatysfakcjonująco niski.

### Wycena testowania

Ile kosztuje testowanie oprogramowania? To pytanie zadawane jest każdego dnia. Jak wycenić coś, co jest nieskończone w czasie? Niemożliwe. Możliwe są jedynie triki:

*Kliencie, powiedz, ile chcesz przeznaczyć na testowanie? A my powiemy, co możemy ci za to dać.*

*Kliencie, zapłać za godzinę pracy i zobacz, ile możemy ci za to dać.*

Moment zakończenia testów będzie wyznaczony przez:

- wyczerpanie budżetu przeznaczonego na testy (lub projekt),
- zakończenie czasu na testowanie,
- ryzyko pojawienia się zawodnego oprogramowania na rynku.

Dwa pierwsze elementy są najbardziej oczywiste i najmniej pożądane z punktu widzenia testerów. Ostatni będzie wynikał z naszych, testerskich szacunków i analiz jakości oprogramowania. A teraz spróbujcie nałożyć to na trójkąt projektowy i wszystko zaczyna się układać w sensowną całość.

### ***Time to market* (w wolnym tłumaczeniu „czas premiery”)**

W wypadku aplikacji zawsze zakładamy, po jakim czasie produkt będzie gotowy, by pojawić się na rynku. Jeśli oprogramowanie trafia na otwarty rynek i do masowego odbiorcy, często wiąże się to z uruchomieniem maszyny marketingowej, wykupieniem powierzchni reklamowej i rezerwacją czasu antenowego na różnego typu reklamy. Tak drobna rzecz jak defekt (nawet krytyczny) nie będzie w stanie zatrzymać tej rozpędzonej maszynerii. Ważne jest więc, aby czas

premiery definiowany był z pełną świadomością zagrożeń, a czas pozostały do wydania – optymalnie wykorzystany.

Zakładamy, po raz kolejny, że działania (testowe) mogą zminimalizować ryzyko w najbardziej krytycznych obszarach. Kryterium zakończenia testów powinno zawsze uwzględniać zmierzony poziom jakości produktu, wystarczający do publicznej prezentacji. Oczywiście musimy podkreślić, że jest to jedynie założenie. Brak w aplikacji nieusuniętych defektów krytycznych nie oznacza, że ich tam nie ma. Testowanie pokazuje, że poziom niepewności może być mniejszy, ale nigdy nie spadnie do zera. W całym procesie zarządzania ryzykiem wystarczy, że nasze testy pokryją ryzyka powiązane z celem aplikacji, pod warunkiem, że wszystkie najważniejsze ryzyka zostaną zidentyfikowane. Chcemy, aby aplikacja, od której może zależeć ludzkie życie, nie przyczyniła się do niczyjej śmierci. Chcemy, aby aplikacja internetowa mogła pełnić (w założonych) warunkach swoje funkcje. Czy chcemy za dużo? Oczywiście. Te wymagania przekładają się na olbrzymią liczbę ryzyk i nieskończoną liczbę przypadków testowych do uruchomienia.

### **Kwantyfikacja ryzyka**

Ryzyko możemy mierzyć. Prosty iloczyn prawdopodobieństwa i wpływu może określić, o jakim ryzyku mówimy.

- Prawdopodobieństwo powstania ryzyka, mierzone w procentach.
- Wpływ liczony w jednostkach monetarnych.

Tak zdefiniowane poziomy ryzyka mogą służyć do określenia, które ryzyko ma dla nas najwyższy priorytet w testowaniu.

Pocieszać się możemy jedynie tym, że nikt przed nami i długo po nas nie stworzy aplikacji idealnej i bezawaryjnej.

### **Aplikacja bezawaryjna**

Według standardowej definicji awaria jest możliwa jedynie po uruchomieniu oprogramowania. Każda aplikacja jest więc bezawaryjna do momentu jej uruchomienia.

---

Zadanie

Na przykładzie dowolnej aplikacji udowodnij nieskończoność testowania. Wskaż, gdzie znajdują się „obszary nieskończoności”.

---

## 4.8. O wyższości wczesnego testowania nad późnym

Testerzy powinni z zasady negować zapisy typu „wczesne”, „późne” itp. jak wszystkie inne zapisy nieprecyzyjne, a więc w konsekwencji również nieweryfikowalne. W książce nie odnosimy się jednak do konkretnych założeń organizacji ani projektu i musimy (niestety) utrzymywać wysoki poziom ogólności. Przy bardzo ogólnych założeniach precyzyjne określenia czasu zaangażowania nie jest możliwe. Będzie on zależał od wielu czynników.

Wczesne techniki zaangażowania dzielimy na dwie kategorie:

- czytanie dokumentacji ze zrozumieniem i zwracanie uwagi na temat nieścisłości, niepoprawności zapisów, czyli przeglądy;
- analiza dokumentu napisanego w języku zrozumiałym przez maszynę za pomocą narzędzi, czyli analiza statyczna.

Realnie rzecz ujmując, tego działania projektowego nie możemy uznać za testowanie oprogramowania. Są to raczej czynności z zakresu statycznej weryfikacji.

Jak wcześnie możemy zacząć testować? Jeszcze zanim powstanie oprogramowanie. Co więcej, możemy rozpocząć testowanie, zanim powstanie dokumentacja oprogramowania. Możemy testować już samą koncepcję i jej realizowalność. Nie zawsze uda nam się zaangażować wystarczająco wcześnie, ale musimy pamiętać starą maksymę „Lepiej późno niż wcale”. Zwłaszcza że możemy również testować każdy półprodukt lub produkt dodatkowy, który wytwarzany jest na drodze od koncepcji do końcowego produktu. Służą do tego niezliczone techniki, metody i narzędzia. Zakładamy, że defekty znalezione we wczesnych fazach zapobiegają defektom w fazach późnych. Zgodnie z badaniami naukowców ze Standish Group i ich rokrocznie publikowanego raportu, prawie 75% wszystkich problemów projektowych to wynik złej specyfikacji i nieudolnego nią zarządzania. Podobne dane możemy

znaleźć w publikacjach Instytutu Gartnera czy też Software Engineering Institute.

### **Amerykańscy naukowcy**

Stwierdzenie: „Amerykańscy naukowcy odkryli, że...” są dla wielu synonimem nic nie wartych badań i oczywistych wyników. Przykład: „Jak udowodnili amerykańscy naukowcy, otwarcie paczki chipsów wymaga tyle siły, że część osób nie może jej otworzyć, co może być uznawane za dyskryminację”. Nie zmienia to faktu, że wiele nawet nie do końca potwierdzonych badań, których wyniki możemy odnieść do swojego doświadczenia, stają się naszymi prawdami.

Dzięki niezliczonym doświadczeniom wiemy również, że testowanie specyfikacji jest w stanie wyeliminować do 70% defektów. Szybka kalkulacja pokazuje, że wczesne zaangażowanie to eliminacja ponad 50% defektów w projekcie. Więc czemu, odwołując się do własnych doświadczeń, projekty, w których pracujemy, nie angażują nas wcześniej? Może dlatego, że testowanie dla wielu to wyłącznie koszt, a nie korzyść? Możemy przeanalizować wymyślone historie, których podobieństwo do realnych zdarzeń i faktów jest przypadkowe:

- Kierownik projektu nie wierzy w testowanie oprogramowania, dlatego budżet na testerów jest ułamkową częścią środków na programistów, a wydłużenie czasu na wytwarzanie oprogramowania powoduje skrócenie czasu na testowanie.
- Klient uważa, że programiści są w stanie zadbać o jakość bez pomocy testerów, dlatego nie uwzględnia się testowania w budżecie projektu. Klient nie zgodzi się na płacenie za testowanie, poprosi raczej o takie pisanie kodu, żeby nie trzeba było go testować.
- Przecież testowanie nie musi być wykonywane przez testerów. Wielu spośród członków projektu przedkłada pracę twórczą nad odtwórczą (za jaką uważa się testy), więc testowanie zostawimy użytkownikowi końcowemu.
- Sami testerzy nie w pełni rozumieją sens czytania dokumentów we wczesnych fazach, skoro i tak ich nie pojmują, a do końca projektu zmieniają się one jeszcze wielokrotnie.
- We wczesnych fazach projektu nie widzi się awarii, więc trudno uzasadnić potrzebę testowania. Przecież nie ma oprogramowania, więc co może nie działać?

Bez względu na to, co myślicie o wczesnym zaangażowaniu, jeśli jeszcze nie wierzycie w skuteczność tych technik, to i tak warto spróbować. Chociażby po to, by udowodnić swoje przekonania lub im zaprzeczyć. Spójrzcie na inne obszary naszego życia, aby poszukać inspiracji. Ustawy, zanim zostaną uchwalone, są poddawane konsultacjom społecznym (testy beta, czyli testy z użytkownikami). Przed wybudowaniem mostu oblicza się jego siłę nośną i możliwe obciążenia, jakie pojawią się na poszczególnych elementach nośnych (analiza statyczna). Środki chemiczne przed premierą rynkową sprawdza się na zwierzętach (testy laboratoryjne). Skoro wczesne testowanie sprawdza się wszędzie dookoła, to dlaczego nie miałyby pomagać w produkcji oprogramowania?

## 4.9. Ekonomia testowania

Wiele testerek i wielu testerów, pytanych, czy testowanie się opłaca, z oczywistych względów nie zaprzecza. Powinniśmy zadać sobie jednak pytanie: ilu z nas potrafi w otwartej dyskusji obronić wartość testowania oprogramowania? Gdzie kryje się wartość dodana znajdowania defektów?

Jedną z najtrudniejszych rzeczy w testowaniu jest ekonomiczne uzasadnienie opłacalności z jednoczesnym powołaniem się na rzeczywiste dane i przykłady. Łatwo z wiarą w swoją nieomyślność, lecz bez argumentów powtarzać, że testowanie jest opłacalne. Czasami jednak stajemy twarzą w twarz z rozmówcą, który niczego nie przyjmuje na słowo (np. klientem). Kiedy trzeba udowodnić swoje racje, nierzadko brakuje nam namacalnych i jednoznacznych dowodów potwierdzających nasze tezy. Jak powiedział Christopher Hitchens (a może przed nim ktoś równie mądry): *Jeśli można twierdzić coś bez dowodu, to można to również bez dowodu odrzucić.*

Zatem, czy testowanie naprawdę się opłaca? Można śmiało powiedzieć TAK, ale za TAK muszą iść konkretne argumenty. Na początek kilka stwierdzeń, które da się łatwo udowodnić i które stanowią podstawę do dalszych rozważań.

- Testowanie oprogramowania ujawnia defekty i awarie lub może pokazać (w ograniczonym zakresie) ich brak.

*Prosty dowód:* tester oprogramowania, który uruchamia aplikację i ma wystarczającą wiedzę, jest w stanie znaleźć w aplikacji przynajmniej jeden incydent, który może się okazać defektem. Rozwijając myśl: testowanie ostatecznie nie musi ujawniać defektów, co nawet jest pożądane. Testowanie może pokazać, że w pewnych szczególnych okolicznościach i w konkretnym środowisku defektów nie ma.

- Znajdowanie defektów w specyfikacji jest tańsze niż w oprogramowaniu.

*Prosty dowód:* do testowania specyfikacji potrzeba wydrukowanej kartki, do testowania aplikacji potrzeba co najmniej komputera z systemem operacyjnym.

- Naprawianie defektów w specyfikacji jest tańsze niż awarii w oprogramowaniu.

*Prosty dowód:* po znalezieniu defektu w specyfikacji można go natychmiast poprawić, po znalezieniu awarii w aplikacji trzeba ją debugować, aby znaleźć jej przyczynę, a następnie defekt usunąć. Wczesne techniki zaangażowania, takie jak przeglądy, mogą podnosić opłacalność całych projektów informatycznych.

- Znajdowanie awarii po stronie wykonawcy systemu jest tańsze niż ich znajdowanie po stronie jego użytkownika, zwłaszcza gdy z niepoprawnym działaniem oprogramowania wiążą się kary umowne dla wytwórcy.

*Prosty dowód 1:* zapłacenie kary umownej jest kosztem niskiej jakości. Zapłacenie kary będzie stratą, a przeznaczenie tych środków na profesjonalne testowanie zwróci się z nawiązką. Jeśli zgodzimy się co do powyższego, to z prostej logiki wynika, że testowanie systemów z zakontraktowanymi karami umownymi jest opłacalne.

*Prosty dowód 2:* awarie widoczne po stronie użytkownika zniechęcają go do zakupu kolejnych produktów, nowych wersji, a także obniżają prestiż rynkowy firmy dostarczającej oprogramowanie.

Przytoczone tu argumenty, choć niepoparte liczbami, są same w sobie niezaprzeczalne. Oczywiście nasi rozmówcy mogą podważyć któryś z nich, ale w takim wypadku należy rozważyć zakończenie dysputy. Nadal niektórzy ludzie wierzą, że Słońce kręci się wokół Ziemi, a życie na naszej planecie zaczęło się 5000 lat temu. Choć głęboko wierzą w to, co głoszą, rozmowa z nimi nie ma większego sensu. Żadne,

najbardziej logiczne argumenty i dowody naukowe ich nie przekonają, że białe jest białe, a czarne – czarne.

Próbując udowadniać opłacalność testowania, należy się również odwoływać do liczb i twardych dowodów, gdyż takie podejście jest obrazowe dla naszych klientów i – co ważniejsze dla nich – priorytetowe. Pierwszym krokiem będą liczby, takie jak liczba defektów, koszt dobrej i złej jakości, czy ROI (*return of investment* – zwrot z inwestycji).

### **Zwrot z inwestycji w testowanie wg Rexa Blacka**

Zwrot z inwestycji w testowanie oprogramowania opisał, przytaczając liczby, Rex Black w swojej publikacji „What IT Managers Should Know about Testing: How to Analyze the Return on the Testing Investment”. Co więcej, dostarczył narzędzie do liczenia wartości zwrotu z inwestycji. Z jego wyliczeń jednoznacznie wynika, że testowanie oprogramowania w każdym przypadku jest zyskowne.

Warto również posłużyć się przykładami ukazującymi konkretne koszty i możliwe oszczędności:

- Z publikacji „The Economic Impacts of Inadequate Infrastructure for Software Testing”, wydanej przez National Institute of Standards & Technology, można się dowiedzieć, że koszty niskiej jakości oprogramowania wynoszą 59,5 mld dolarów rocznie, a możliwe oszczędności związane z lepszym testowaniem to 22,2 mld dolarów rocznie.
- Koszt defektów to pojęcie pojawiające się w wielu publikacjach, w tym w książce Steva McConnella „Code Complete”, wydanej przez Microsoft Press.
- Sposób obliczania kosztów naprawiania defektów i awarii w zależności od czasu ich znalezienia i usunięcia pokazano w tab. 4.3. Dane tam zawarte pokrywają się z regułą: 1 : 10 : 100, wspomnianą w wielu innych źródłach, m.in. w badaniach IBM czy w książce Williama E. Perry’ego *Effective Methods for Software Testing*. Reguła ta mówi, że koszt naprawy defektów w specyfikacji jest jednokrotnością w porównaniu do 10-krotnie większego kosztu, gdy ten sam defekt trzeba naprawić po testowaniu, i 100-krotnie większego, jeśli usuwany jest po wydaniu oprogramowania klientowi.



- Informacje o awariach. Najlepiej się powołać na najnowsze i najgłośniejsze informacje o awariach i ich kosztach. Kilka przykładów znajdziecie w dalszej części tej książki.

**Tab. 4.3.** Koszty poprawiania defektów i awarii w zależności od czasu ich znalezienia i usunięcia

Koszt naprawy defektu		Czas i miejsce wykrycia				
		Wymagania	Architektura	Implementacja	Testowanie systemowe	Po wyd
Miejsce wprowadzenia	wymagania	1x	3x	5–10x	10x	10–1
	architektura	–	1x	10x	15x	25–1
	implementacja	–	–	1x	10x	10–

Przytoczone tu przykłady powinny przekonać nawet najzatatwardzialszych krytyków testowania.

Nie należy jednocześnie zapominać, że testowanie samo w sobie jest kosztem. To nie darmowy lunch dla wszystkich i nie da się jednoznacznie powiedzieć, że przynosi same korzyści. Dla większości projektów testowanie będzie jedynie obciążeniem, a nie zyskiem. Nie zmienia to jednak faktu, że przy prostej analizie zwrotu z inwestycji testowanie zwiększa prawdopodobieństwa powodzenia inwestycji w wytworzeniu produktu software’owego.

Testerzy od początku istnienia ich dziedziny próbują przekonać siebie i świat, że testowanie samo w sobie i w każdej sytuacji przynosi zyski. Działają zgodnie z regułą, że odpowiednio długo powtarzane kłamstwo ma szansę stać się prawdą. Oczywiście nie mówimy tu w każdym przypadku o złej woli i chęci wprowadzenia kogoś w błąd. Wiąże się to raczej z wiarą w swoją nieomylność. Należy podkreślić, że wielu testerów ma pełne przekonanie co do tego dogmatu i również tutaj dyskusja nie ma wielkiego sensu. Powiedzmy sobie szczerze:

- Testowanie nie podnosi jakości oprogramowania – tak jak mieszanie łyżeczką w herbie nie powoduje, że staje się ona słodsza. Trzeba dodać cukier jako dodatkowy składnik. Brak cukru jest tu metaforą złego zarządzania procesem wytwórczym.
- Testowanie to koszt. Im wcześniej są zaangażowane testy, tym teoretycznie mniejsze koszty poprawy defektów w późniejszych

fazach. Tym jednak większe koszty samego testowania ponoszone są od początku projektu.

- Testy mogą stać się wydatkiem, jeśli ich wyniki nie przekładają się na decyzje biznesowe lub ogólnie projekt nie jest w stanie dostarczyć produktu.
- Testowanie może być niekończącym się kosztem. Czy uda nam się zapewnić zerową liczbę defektów w skończonym czasie? Czy testowanie daje 100% gwarancję poprawności oprogramowania? Nie.

Odpowiednio jednak przeprowadzone testowanie ma szansę na powodzenie i udowodnienie swojej wartości biznesowej. Jest kosztem niezbędnym do poniesienia, tak jak (prawie) każdy inny koszt projektowy:

- Czy koszt pracy nawet najlepszego kierownika projektu jest zyskiem? Nie. Czy zagwarantuje powodzenie projektu? Nie. Może ograniczyć ryzyko jego niepowodzenia, ale gwarancji nie da.
- Czy programowanie jest zyskiem? Nie. Jest nakładem pracy, więc wydatkiem. Czy najlepszy programista zagwarantuje wytworzenie produktu? Nie.
- I tak dalej.

W naszych rozważaniach zawsze należy założyć, że testowanie jest kosztem, który ma szansę się zwrócić, jeśli tylko środki wydatkowane na weryfikację i sprawdzenie zostaną rozdystrybuowane z głową.

---

## Zadanie

Znajdź namacalny dowód na opłacalność testowania. Serwisy pełne są informacji o awariach oprogramowania. Sprawdź, czy w podanych przez media przykładach rzeczywiście testowanie mogłoby pomóc?

---

## 5. Dzielenie testowania

### 5.1. Wprowadzenie

Z jakiegoś niepojętego powodu testowanie musi się dzielić na poziomy, typy, etapy i dziesiątki innych kategorii, które nie są stosowane w większości znanych mi – i zapewne również czytelnikowi – projektów. Podziały te są jednak na tyle mocno rozpowszechnione i opisane, że trudno się do nich nie odwołać. Zakładamy przy tym, że są to pewnego rodzaju „najlepsze praktyki”, które można wdrożyć, jeśli planujemy naszą organizację prowadzić w zgodzie z wybranymi wzorcami tzw. dobrego testowania.

Testy możemy podzielić na poziomy, białoskrzynkowe i czarnoskrzynkowe, funkcjonalne i niefunkcjonalne, testy potwierdzające itd. Możemy znaleźć również inne podziały, mniej lub bardziej rozpowszechnione i spopularyzowane. Przyjrzyjmy się części z nich, aby pokazać, że są sztucznymi zbiorami zawierającymi się same w sobie lub też znacząco się pokrywającymi.

Kategorie testów
Pewnym pocieszeniem dla każdego testera powinna być ogólna trudność kategoryzowania wszystkiego w przyrodzie. Ze względu na złożoność i zmienność w nauce każdego dnia pojawiają się nowe odkrycia, więc trudno jednoznacznie skatalogować chociażby organizmy żywe, a co dopiero wymyślić dobre podziały w testowaniu.

Dlaczego uważam, że każdy szanujący siebie i swój zawód tester powinien znać podziały w testowaniu, ponieważ:

- są stosowane i powinniśmy znać środowisko, w którym pracujemy;
- stanowią ciekawe pomysły na projektowanie nowych testów;

- pozwalają nam w niektórych wypadkach pójść na skróty, np. przy klasyfikowaniu defektów;
- testowanie *stricte* funkcjonalne jest tylko pierwszym szczeblem w karierze, a rozwój w testowaniu to albo role menedżerskie, albo stanowisko specjalisty w danym obszarze testowania.

Szczególnie ten ostatni punkt jest istotny i chciałbym czytelniku, abyś czytając ten rozdział, myślał o tym, który obszar testowania najbardziej ci się podoba. Czy ekscytują cię może hakerskie ataki na oprogramowanie? W takim wypadku najbliższej ci będzie do testowania bezpieczeństwa. A może mityczna szybkość aplikacji jest ci bliska? Mógłbyś więcej uwagi poświęcić wydajności. Jeśli zauroczony jesteś interfejsami, to ciekawym obszarem może być użyteczność. Nie ma co ukrywać, że tester funkcjonalny, choć jest to projektowo ważna rola, dostaje marne grosze w porównaniu z tym, ile może zarobić specjalista na charakterystykach jakościowych.

---

## Zadanie

Zanim przystąpisz do czytania tego rozdziału, zastanów się i napisz na kartce, jak widzisz podziały w testowaniu.

---

## 5.2. Czarna skrzynka i biała skrzynka

Ktoś kiedyś nazwał oprogramowanie zamkniętą skrzynką. Ta skrzynka ma wyprowadzone na zewnątrz interfejsy, które mogą stanowić podstawę do zweryfikowania jej zawartości. Niestety, ze względu na złożoność zawartości skrzynki oraz ułomność technik testowania samych interfejsów weryfikacja nigdy nie będzie pełna.

Będąc dzieckiem, zapewne bawiłeś się, czytelniku, w zgadywanie, co jest w skrzynce. Różne wersje tej zabawy polegały na pytaniach z odpowiedzią TAK/NIE bądź na dotykanie przedmiotu, nie widząc, jak wygląda. Wiesz, jak trudne jest zrozumienie zawartości skrzynki, jeśli można korzystać jedynie w ograniczonym zakresie ze swoich zmysłów. Nie mogąc czegoś zobaczyć, musimy losowo zadawać pytania lub po omacku penetrować, by odpowiedzieć na pytanie, co się znajduje w środku. W dorosłym życiu ta zabawa z dzieciństwa staje się naszym zawodem.

Osoby, które wiedzą, co jest w skrzynce, to umiejący kodować programiści i architekci systemowi. Osoby, które funkcjonują na zewnątrz skrzynki, zazwyczaj nie mają kompetencji programistycznych lub dostępu do kodu źródłowego.

#### **Pokrycie kodu przez testy zewnętrznych interfejsów?**

Badania dowodzą, że testowanie na poziomie zewnętrznych interfejsów aplikacji dają nie więcej niż 20-procentowe pokrycie kodu źródłowego. Oznacza to, że takimi testami udaje nam się uruchomić nie więcej niż 1/4 kodu oprogramowania.

W związku z takim nazewnictwem możemy wyróżnić dwa nowe typy testów: „białoskrzynkowe”, które ze względu na zakres wykonywania i odpowiedzialności możemy nazwać po prostu programistycznymi, i „czarnoskrzynkowe”, które w dużym uproszczeniu możemy nazwać systemowymi.

### **5.2.1. Testy białej skrzynki**

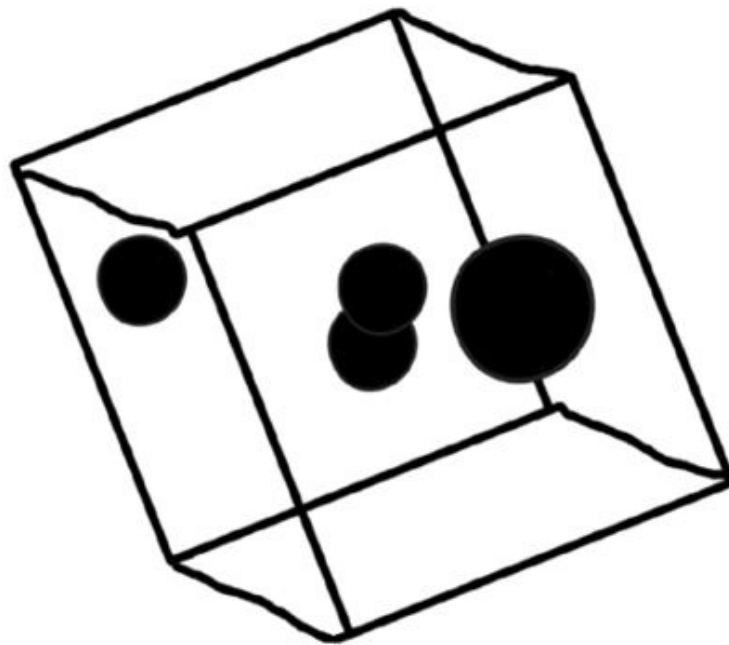
Będąc wewnątrz białej skrzynki, mamy dostęp do wiedzy wykraczającej poza to, co jest nam znane z poziomu interfejsów. Możemy widzieć takie elementy, jak struktura danych (np. tabele przechowujące informacje) czy algorytmy wewnętrzne aplikacji. Próbuję to przedstawić na rys. 5.1. Testowanie białoskrzynkowe ma niezaprzeczalną zaletę precyzyjnej miary pokrycia. Trudno szukać tego na poziomie czarnej skrzynki. Testowanie na poziomie struktury wewnętrznej oprogramowania umożliwia określenie, w jakim stopniu udało się uruchomić pojedyncze wiersze kodu, punkty decyzyjne czy też warunki w decyzjach. Dla większości znanego nam oprogramowania takie miary nie będą miały większego znaczenia, ale dla systemów, od których może zależeć ludzkie życie, takie testy mogą oznaczać znaczące zwiększenie świadomości jakości czy też zakładanej niezawodności systemu.

#### **Co programista wie o pokryciu?**

Kiedy na szkoleniach z testowania spotykam programistów, nie mogę się powstrzymać, by nie zapytać ich o wiedzę na temat pokrycia funkcji czy też pokrycia decyzji w kodzie. Po wielu rozmowach wiem, że znacząca większość koderów (ok. 90%) nie ma najmniejszego pojęcia,

czym te techniki są i do czego się ich używa. Dlaczego więc 100% testerów pragnących zdobyć certyfikat testerski musi je poznać?

Testowanie na poziomie kodu ma wiele odmian, ale raczej należy je traktować jako podzbiór technik białej skrzynki i określić mianem metod. Warto na pewno do ogólnego procesu zapewnienia jakości w skrzynce dodać również testowanie statyczne. Czysty kod źródłowy jest zwykłym dokumentem napisanym w specyficznym języku. Osoba, która może przeczytać dany tekst, może przez samo czytanie ze zrozumieniem weryfikować jego poprawność. Takie czytanie zakończone raportem z testów możemy nazwać testem statycznym lub przeglądem, które nie wymagają uruchomienia oprogramowania.



**Rys. 5.1.** Oprogramowanie jako biała skrzynka

W ramach białej skrzynki zdefiniowano wiele technik, w których przypadki testowe opierają się na analizie wewnętrznej struktury oprogramowania:

- testowanie instrukcji, czyli pojedynczych wierszy kodu;
- testowanie punktów decyzyjnych w kodzie, czyli wszystkich miejsc, gdzie w kodzie źródłowym podejmowana jest decyzja, np. prawda lub fałsz;

- testowanie rozgałęzień w kodzie, zbliżone do pokrycia decyzji, przy czym technika zorientowana jest na gałęzie kodu, a nie na pojedyncze punkty wejścia w gałęzie (punkty decyzyjne);
- testowanie warunków decyzji w kodzie, której zawężeniem jest testowanie jedynie warunków mających znaczenie – techniki te dotyczą szczegółowych zdarzeń w ramach punktów decyzyjnych;
- testowanie przepływu danych (nie mylić z analizą), w trakcie którego dane będą analizowane od momentu ich zdefiniowania, przez użycie, aż do usunięcia;
- testowanie przepływu sterowania, czyli weryfikacja złożonych kombinacji decyzji w kodzie;
- testowanie ścieżek w kodzie, które w realnym oprogramowaniu cechują się nieosiągalnym pokryciem 100-procentowym i gdzie testy powinny wykonać każdą ścieżkę, którą można przejść przez kod.

To jedynie część technik, które można spotkać po uruchomieniu kodu na poziomie programistycznym. Nie będę ich omawiać ze względu na wąską dziedzinę zastosowania, a osoby zainteresowane odsyłam do publikacji z obszaru programistycznego.

Znając kod, możemy testować zachowanie poszczególnych modułów lub komponentów również innymi metodami. Najbardziej rozpowszechnioną techniką jest tzw. wstrzykiwanie defektów, mające sprawdzić odporność jednostek na niepoprawny format danych wejściowych. Ta technika ma również zastosowanie do różnego rodzaju elementów komunikacji, nie tylko na poziomie komponentów, ale również systemów.

Pokrewną techniką jest tzw. posiew błędów, czyli celowe wprowadzanie defektów do oprogramowania. Dlaczego ktokolwiek miałby wprowadzać defekty do kodu? Celów jest kilka, a przynajmniej jeden z nich może być niespodzianką dla czytelników.

Technika służy do:

- monitorowania liczby znalezionych defektów i ich usuwania – zakładamy, że nieznanie przez zespół testerski defektu, który został celowo wprowadzony do oprogramowania, dowodzi, że testy jeszcze się nie zakończyły (może to więc być pewne kryterium zakończenia testów);

- estymacji defektów pozostałych w oprogramowaniu – jeśli określimy, jaki procent defektów celowo wprowadzonych do oprogramowania udało się znaleźć, i odejmiemy tę wartość od 100%, to możemy otrzymać estymatę procentowej liczby defektów pozostałych w jednej wersji oprogramowania;
- testowania specyfikacji testowej – jeśli zestaw przypadków testowych nie pozwala wykryć defektu celowo wprowadzonego do oprogramowania, oznacza to, że musi zostać poprawiony (lub trzeba dodać nowe przypadki testowe);
- testowania testerów – bardzo wyrachowana technika, stosowana przez kierowników do oceny jakości pracy, gdyż jeśli testerzy nie są w stanie znaleźć defektów celowo wprowadzonych do oprogramowania, to może to świadczyć o ich braku skuteczności lub tego, że oprogramowanie nie ujawnia symptomów problemu (jest nietestowalne). Miara ta może być dość kontrowersyjna i wzbudzać negatywne emocje, ale należy pamiętać, że jest to tylko jedno z wielu narzędzi oceny pracy testerów przez menedżerów. Jak to z narzędziami bywa, w nieodpowiednich rękach z elementu wsparcia mogą się zamienić w narzędzie do wywierania presji na testera.

### **Jedna wersja oprogramowania**

Bardzo ważnym założeniem, przyjmowanym dla analizy jakości oprogramowania, jest pojedyncza wersja oprogramowania. To dla pojedynczej wersji jesteśmy w stanie założyć skończoną liczbę defektów i niemożność przetestowania wszystkiego. Musimy pamiętać, że poprawki do oprogramowania są w rzeczywistości wytworzeniem jego nowej wersji.

## **5.2.2. Testy czarnej skrzynki**

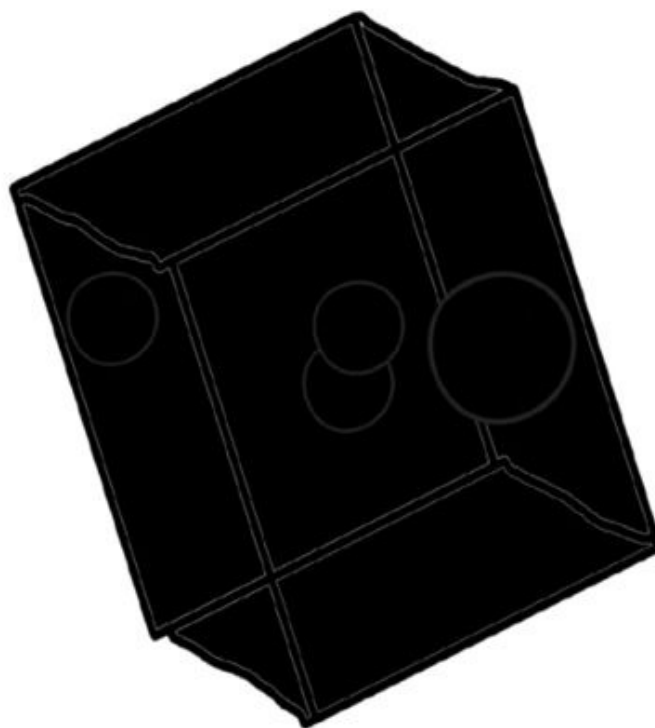
Testowanie czarnoskrzynkowe to kwintesencja pracy testera oprogramowania. Widzimy więc naszą skrzynkę w pełnej okazałości, mając blade lub żadne pojęcie o jej zawartości, za to z dostępem do interfejsów wyprowadzonych na zewnątrz (rys. 5.2). Większości z nas, zawodowych testerów, może się to kojarzyć z codzienną praktyką, czyli próbą analizy zachowania aplikacji pozbawionej dokumentacji. I nie mówimy tu tylko o dokumentacji opisującej zachowanie aplikacji, lecz



także o jakiegokolwiek specyfikacji działania. Nasze testy na pierwszym etapie mogą zostać sprowadzone do tzw. ślepego klikania, zakończonego raportami z defektów nagminnie odrzucanymi przez programistów z komentarzem: działa tak, jak zaprojektowano. Aby tego uniknąć, testowanie czarnoskrzynkowe powinno być wspierane dokumentacją (*requirements based testing*). Tylko gdzie ta specyfikacja jest?

<b>Dlaczego ta skrzynka jest czarna?</b>
Złośliwa wersja pochodzenia nazwy „czarnoskrzynkowa” wywodzi się z branżowego suchara, że testowanie manualne – klikanie – wykonywane jest przez ludzi od czarnej roboty.

W ramach tego typu testowania wyróżniamy dziesiątki formalnie zdefiniowanych technik, które łączą próbę weryfikacji jakości oprogramowania z możliwą optymalizacją efektywności poszukiwania defektów. Stąd biorą się techniki testowania jedynie pojedynczej wartości w klasach równoważności czy jedynie wartości na granicach tych klas. Wyróżniamy oczywiście także inne techniki opisane pobieżnie w dalszej części. I tu należy się wytłumaczenie, dlaczego nie będę opisywał wszystkich tych technik z uwzględnieniem ich formalnych nazw, ograniczeń i opisów skuteczności. Powodów jest kilka, a do najważniejszych zaliczyłbym dwa. Po pierwsze, to nie jest praktyka testowania, a większość z technik jest intuicyjna i nie musisz, czytelniku, ich znać z nazwy, a i tak będziesz je stosował. Po drugie, istnieje wiele publikacji, które opisują techniki, pokazując ich „praktyczne” zastosowanie.



**Rys. 5.2.** Oprogramowanie jako czarna skrzynka (opracowanie własne)

Czarna skrzynka ma wady. Skoro nie wiem, co w niej jest, to nie wiem, jak działa oprogramowanie, jak jest skonstruowane, co zostało zaimplementowane, a co nie, więc jak mogę przetestować poprawność jego działania? Skąd wiem, że coś jest na tyle złożone kodowo, że wymaga dziesiątków przypadków testowych, a inne obszary są prostymi funkcjami z wystarczającym pokryciem jednego testu? Niestety nie wiem. Powinienem tę informację otrzymać w dokumentacji. A dokumentacji, co powtarzam do znudzenia, zazwyczaj brakuje.

Niektóre publikacje testerskie wyróżniają dodatkowy typ – testy szaroskrzynkowe. Nie traktuję ich jednak jako osobnego elementu w kategorii, skoro są połączeniem testów czarnej i białej skrzynki, czyli testowaniem z częściową wiedzą o działaniu oprogramowania. Wykonywanie testów ciągle odbywa się na poziomie interfejsów. Nie ma więc innych cech, które stanowiłyby o ich wyjątkowości.

### **Jeszcze o stuprocentowym pokryciu**

Jednym z ważniejszych aspektów testowania jest uzyskanie dla danego obszaru aplikacji pokrycia 100-procentowego, czyli pełnego. To byłoby

potwierdzenie poprawności działania i dowód braku defektów. Był z wielu powodów nierzeczywisty.

Dążenie do 100% ma zagwarantować większą świadomość jakości oprogramowania. Każdy typ pokrycia ma swoje wady i zalety, a pokrycie 100-procentowe, choć istnieje, nie jest jednoznaczne z „pełnym” pokryciem.

Możemy uzyskać 100-procentowe pokrycie struktury lub architektury kodu, ale niestety nic (lub bardzo niewiele) to nie mówi o jakości dostarczanego oprogramowania na poziomie jego interfejsu. Dodatkowo pokrycie 100-procentowe jest możliwe jedynie dla najsłabszych technik, takich jak pokrycie instrukcji czy decyzji.

100% pokrycia możemy uzyskać dla modelu oprogramowania. Ze względu jednak na złożoność większości znanego nam oprogramowania niemożliwe jest wymodelowanie go w 100%, a niedoskonały model nie może być podstawą do uzyskania pełnego pokrycia.

100-procentowe pokrycie specyfikacji napisanej w języku naturalnym oznacza jeszcze mniej. Opisana narracyjnie dokumentacja wymagań to opis słowny, który zazwyczaj jest nie tylko niepełny, ale może też być interpretowany. Nie można więc na jego podstawie uzyskać mitycznych 100%. Jest on znacznie gorszy dla informatyków niż suchy język techniczny.

100-procentowe pokrycie nie jest więc możliwe, a co więcej – jest nieosiągalne.

Istotne zdanie pojawiające się w wielu publikacjach brzmi: *Pokrycie nie oznacza, że cały zestaw testów jest kompletny, a jedynie, że dana technika/metoda/podejście nie sugeruje żadnych przydatnych testów dodatkowych.*

---

## Zadanie

Znajdź w internecie dowolne oprogramowanie lub kod źródłowy i sprawdź jego podatność na testy.

---

## 5.3. Testowanie funkcjonalne i нефunkcjonalne

Funkcjonalność oprogramowania to zestaw jego funkcji. Czym w takim razie jest нефunkcjonalność? Wracając do rozważań z początku książki

i odwołując się do słownika, możemy powiedzieć, że testy niefunkcjonalne to testy inne niż funkcjonalne. Próbując zagmatwać rzeczy, możemy sformułować definicję, że jeżeli dana aplikacja komputerowa ma określony zbiór funkcji, to zanegowaniem tego zbioru jest wyróżnienie zbioru cech oprogramowania tworzących zbiór niefunkcjonalny. Zatem testy funkcjonalne to testy funkcji, a testy niefunkcjonalne to testy atrybutów, które nie są funkcjami. Stąd pojawi się konieczność zdefiniowania tychże atrybutów jako cech oprogramowania.

Ponieważ cechy jako takie trudno się definiuje, dlatego przy ich określaniu odwołamy się do przykładów.

### 5.3.1. Testy funkcjonalne

Testowanie funkcjonalne jest sprawdzeniem funkcji oprogramowania. Są dwa podstawowe źródła wiedzy o poprawności działania danego oprogramowania. Pierwsze, ciągle niezastąpione źródło to człowiek, drugim jest dokument lub precyzyjniej: specyfikacja funkcjonalna oprogramowania. Mogą one poinformować, co dane oprogramowania ma robić, czyli jakie są oczekiwane odpowiedzi systemu na dane działania użytkownika.

#### **Testowanie funkcjonalne wg certyfikacji ISTQB**

Zgodnie ze słownikiem testerskim ISTQB, funkcjonalność to zdolność oprogramowania do zapewnienia funkcji odpowiadających zdefiniowanym i przewidywanym potrzebom, gdy oprogramowanie jest używane w określonych warunkach. Testowanie funkcjonalne pojawia się zazwyczaj w ujęciu czarnoskrzynkowym. Słownik definiuje projektowanie funkcjonalnych przypadków testowych jako procedurę bez zaglądania w wewnętrzną strukturę oprogramowania, zatem jest to czarnoskrzynkowa technika projektowania przypadków testowych. Natomiast testowanie czarnoskrzynkowe zdefiniowane jest zarówno jako funkcjonalne, jak i niefunkcjonalne. Możemy więc założyć, że w rozumieniu ISTQB czarna skrzynka jest pojęciem szerszym niż funkcjonalność.

Nie należy łączyć pojęcia funkcjonalności i uzyskania pokrycia wywołań funkcji, które wymaga od nas wiedzy programistycznej.

Co do specyfikacji, to doświadczeni członkowie projektów informatycznych przestali się łudzić, że ona istnieje lub że jest aktualna. To trochę tak jak z mitycznym yeti. Niby ktoś go widział, ale gdy przychodzi do znalezienia, to nikt nie wie, gdzie szukać. Dobrze, jeśli w profesjonalnie prowadzonych projektach jest specyfikacja, ale akceptuje się nawet to, że jest niekompletna lub częściowo nieprawdziwa.

A kim jest wspomniany człowiek? Niestety, zazwyczaj o tym, jak działa oprogramowanie, opowiada nam osoba o podobnej wiedzy jak nasza własna. Warto jednak pamiętać, że najważniejszą osobą, do której warto dotrzeć, jest tzw. sponsor. To zazwyczaj ktoś ważny, bardzo zapracowany i niechętny do kontaktów z osobami z niższego poziomu projektowego lub organizacyjnego. Dotarcie do niego może więc być niezmiernie trudne. Co więcej, sponsor jest też świadomy swojej wagi dla projektu informatycznego, dlatego wie, że inni będą próbowali zadawać mu pytania i prosić o pomoc. Ucieczką i formą obrony dla sponsora staje się więc przejście do bytu bardziej wirtualnego, bez imienia i nazwiska, pod takimi nazwami jak Rada, Zgromadzenie czy Komitet. Dotarcie do sponsora staje się więc już praktycznie niemożliwe. Dlaczego chcemy do niego dotrzeć? Bo zgodnie ze starym przysłowiem „Kto płaci, ten wymaga”, sponsor jest najlepszym źródłem wiedzy o tym, jak oprogramowanie powinno się zachowywać. Co prawda, nie do końca, ale możemy powiedzieć, że osoba płacąca za oprogramowanie będzie miała pewne oczekiwania względem zestawu jego funkcji. Te wymagania sprawny specjalista od wytwarzania oprogramowania musi przekuć w oprogramowanie. Jeśli założymy, że nasze wymagania nie przyjmują formy papierowego, oficjalnego dokumentu, to oprogramowanie tworzone jest w iteracjach, a poszczególne funkcjonalności lub też doprecyzowania funkcjonalności są konsultowane ze sponsorem. Jeśli unika on odpowiedzialności i nie chce podejmować decyzji, możemy tworzyć oprogramowanie, błędząc po omacku, ze ślepą wiarą, że a nuż tego właśnie sponsor oczekiwał. Możemy próbować rekompensować brak dostępu do sponsora kontaktem z innymi osobami mającymi wiedzę o działaniu oprogramowania, np. użytkownikami lub testerami podobnych systemów. Wracamy więc do sprawdzającej się w wielu miejscach reguły, że oprogramowanie muszą testować osoby, które je znają.

Należy pamiętać, że testowanie funkcji jest również weryfikowaniem funkcji niepotrzebnych. Szukamy więc funkcjonalności, które z jakiegoś

powodu zostały dodane do oprogramowania bez wyraźnych oczekiwań płynących ze specyfikacji czy też od sponsorów. Część z nich może być typu „skoro już jest, to niech będzie”, a część może być niepotrzebna do tego stopnia, że zamazuje przejrzystość aplikacji i należy je usunąć.

Testowanie funkcjonalne jest więc zadaniem relatywnie trudnym, jednak przy odpowiedniej dozie doświadczenia połączonego ze sprytem i ogólną wiedzą o działaniu oprogramowania można je przeprowadzić profesjonalnie.

### 5.3.2. Testy niefunkcjonalne

Niefunkcjonalność to wszystko, co nie mieści się w pojęciu funkcjonalności. Szybkość, przyjazność, niezawodność działania oprogramowania nie są jego funkcjami, a jedynie cechami lub atrybutami.

#### Pozytywne myślenie czy czepianie się

Współczesne metodyki wytwarzania oprogramowania, takie jak podejście zwinne (*agile*), nie traktują testowania oprogramowania jako osobnej funkcji ze względu na wąski zakres jej działania. Wiele osób uważa, że celem testera jest „szukanie dziury w całym”. To podejście bardzo popularne i promowane, co nie zmienia faktu, że nie jest niepoprawne.

Istnieją dwa podejścia, które tester powinien stosować swojej pracy:

1. Pozytywne myślenie – testerzy powinni myśleć o sobie jako o osobach chcących potwierdzić, że aplikacja robi dokładnie to, co chciał stworzyć programista i co chciał dostać klient.
2. Czepianie się – czyli udowadnianie, że aplikacja nie działa. Podejście destruktywne nie różni się niczym od pozytywnego, ale dużo trudniej je sprzedać.

Opisaliśmy to wcześniej jako szkoły testowania.

Testowanie niefunkcjonalne jest trudniejsze od funkcjonalnego. Podstawowym założeniem jest to, że mamy określoną charakterystykę, którą będziemy próbowali zweryfikować przez różnego rodzaju badania laboratoryjne, eksperymenty i obserwacje.

Badania laboratoryjne będą polegały na przeprowadzeniu dowodu poprawności działania oprogramowania w rozumieniu spełnienia

pokładanych w nim nadziei. Dla dowolnego obiektu badań tworzymy możliwie optymalne środowisko funkcjonowania i sprawdzamy, czy po wykonaniu prac naprawczych jest on:

- szybszy lub tak szybki, jak chciał zlecający,
- odporniejszy lub tak odporny, jak zakładają standardy,
- użyteczniejszy lub tak użyteczny, jak chciałby użytkownik końcowy,
- itd.

Aby sprawdzić, czy jest szybszy i czy jest odporniejszy, musimy mieć inny system, do którego możemy się odnieść. Może to być wcześniejsza wersja wytwarzanego oprogramowania lub software konkurencji. Na punkt odniesienia w postaci innego produktu stawiamy z powodu braku specyfikacji.

Oczywiście nasze badanie zawsze zakłada, że dostępne są cele, jakie mamy osiągnąć, czyli w naszym przypadku miary ilościowe lub jakościowe spisane w specyfikacji. W razie braku zdefiniowanych miar nie istnieje punkt odniesienia, który powie nam z dużą dozą prawdopodobieństwa o poziomach akceptacji oprogramowania. Punktem odniesienia może być, jak w testowaniu funkcjonalnym, człowiek. Takie działania możemy nazwać testowaniem charakterystyk oprogramowania będącym mierzaniem wartości wyjściowych z aplikacji, jak np. w testowaniu wydajności.

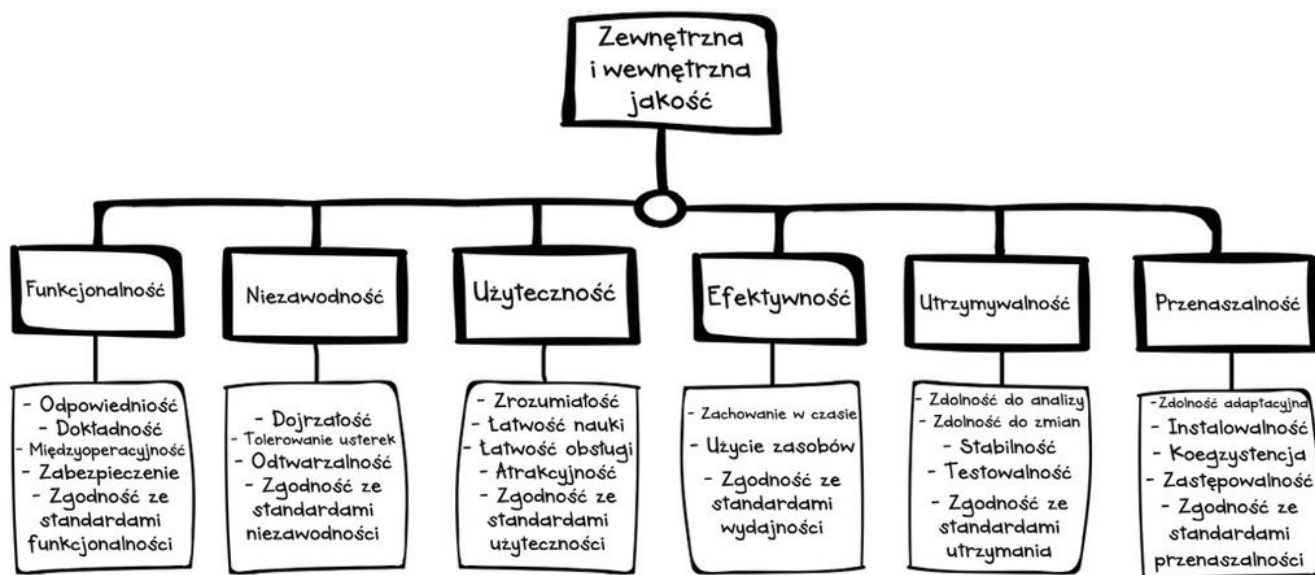
Eksperymenty będą się pojawiać tam, gdzie będziemy się opierać na własnej interpretacji wielkości miary. Możemy eksperymentować na ludziach, włączając do tego częściowo obserwację zachowań użytkowników. Możemy również eksperymentować z wyizolowanym obiektem przy założeniu, że jego działanie w świecie rzeczywistym będzie zbliżone do przyjętego na potrzeby eksperymentu. Eksperyment w tym wypadku wydaje się najgorszą z opcji, ale w praktyce bywa najczęściej stosowany. Takiego rodzaju testowanie wiąże się z niepełną specyfikacją wymagań (lub kompletnym brakiem wymagań) w obszarze niefunkcjonalnym.

Obserwacja pełna będzie już całkowicie polegać na eksperymentach z ludźmi pracującymi z daną aplikacją. Do tych celów wybiera się reprezentatywną grupę osób, którym daje się pewne zadania i analizuje się zdolność do ich wykonania. Klasycznym przykładem są testy użyteczności lub przyjazności danej aplikacji.

Dalej prezentuję najbardziej rozpowszechnione próby klasyfikacji atrybutów oprogramowania.

### 5.3.3. Charakterystyki oprogramowania wg ISO 9126/ISO 25010

W kwestii kompleksowego podejścia do kategoryzacji funkcjonalności i niefunkcjonalności nikt nie poszedł tak daleko jak ISO. International Organization for Standardization próbuje standaryzować wielość i mnogość obszarów nie tylko w IT, ale również na przykład w biznesie. Stworzyła normę ISO/IEC 9126-1, która próbuje opisać jakość oprogramowania przy użyciu charakterystyk i subcharakterystyk. Co prawda, norma ta została zastąpiona przez ISO/IEC 25010, ale w czasie pisania tej książki funkcjonują one jako dwa oficjalne międzynarodowe standardy. Wersja 25010 została oficjalnie opublikowana i jest standardem, wersja 9126-1 jest wycofywana. Mimo że zgodnie z zaleceniami powinniśmy się odwoływać raczej do normy nowszej, to starsza nadal traktowana jest jako wyznacznik i zgodnie z nią przyjrzymy się kategoryzacji charakterystyk. Warto zauważyć, że w normie tej funkcjonalność stanowi jedną z charakterystyk.





**Rys. 5.3.** Charakterystyki oprogramowania wg ISO 9126

MTBF
Dla oprogramowania przyjmuje się bardzo pesymistyczne i jednocześnie realistyczne założenie, że zawiedzie, a główną miarą niezawodności stało się MTBF ( <i>mean time between failure</i> ), co po polsku oznacza średni czas między awariami. Twórcy miary nie zakładają, że liczba awarii będzie równa zero, bo jednocześnie musielibyśmy doprowadzić do dzielenia przez zero, co nie jest ani łatwe, ani łatwo wytłumaczalne zarówno z fizycznego, jak i matematycznego punktu widzenia. Taka definicja potwierdza tezę, że awaria systemu jest tylko kwestią czasu. Miarą, która określi, jaki to czas w przypadku nowych aplikacji, jest MTTF ( <i>mean time to failure</i> – czyli czas do następnej awarii).

Podział pokazany na rys. 5.3. jest następujący:

1. Funkcjonalność (*functionality*), czyli opisywana wcześniej zdolność oprogramowania do zapewnienia zdefiniowanych funkcji.
  - Odpowiedniość (*suitability*) – w jakim zakresie dane oprogramowanie ma zaimplementowane funkcje potrzebne z punktu widzenia użytkownika. Jest to charakterystyka pokrywająca się z subiektywnie rozumianą użytecznością czy też ocenianym przez użytkownika poziomem przydatności.
  - Dokładność (*accuracy*) możemy podzielić na dwa obszary. Pierwszy dotyczy tego, czy dane oprogramowanie jest w stanie dostarczyć rezultaty, jakich oczekujemy. Drugi to zwykła precyzja otrzymanych rezultatów. Rozważając na przykład system sprzedaży w sklepie, oczekujemy rezultatu z dokładnością do 0,01, czyli do jednego grosza. Gdybyśmy otrzymali rezultat zaokrąglony do złotych, to moglibyśmy poczuć się oszukani.
  - Międzyoperacyjność (*interoperability*), czyli zdolność systemu lub modułu do współdziałania i wymiany informacji z innymi modułami lub systemami.
  - Zabezpieczenie (*security*). Dla części czytelników może być dziwne, że zabezpieczenie pojawia się jako podkategoria funkcjonalności, jednak norma dobrze wyjaśnia ten konkretny

zakres funkcji. Są to przede wszystkim elementy mające na celu weryfikację poziomu zabezpieczeń przez nieautoryzowany, acz przypadkowy dostęp lub też umyślny dostęp do funkcjonalności i/lub danych systemu. Takie testy nazywamy również testami autoryzacji, czyli sprawdzeniem, czy osoby o danym poziomie autoryzacji (w tym zerowym – brak dostępu) mają dostęp do zdefiniowanych dla nich funkcji i nie mają dostępu do innych.

- Zgodność ze standardami funkcjonalności (*functionality compliance*). Norma nie odnosi się do konkretnych standardów, dlatego należy założyć, że jest to zgodność ze standardami krajowymi, prawnymi czy też branżowymi.

2. Niezawodność (*reliability*). W określonych warunkach oprogramowanie musi dostarczać określone funkcje. Ważne jest założenie, że niezawodność analizuje się często jako liczbę poprawnie wykonanych operacji w określonym czasie. Tak więc łatwo możemy przeprowadzić wywód analityczny: oprogramowanie ma defekty → oprogramowanie w czasie użytkowania ujawni defekty → defekty mogą prowadzić do awarii → jeśli oprogramowanie będzie użytkowane odpowiednio długo, to, zgodnie z prawami natury, awaria na pewno nastąpi. Praktyk wie, że oprogramowanie zawiedzie prędzej czy później, pytanie tylko – kiedy?

- Dojrzałość (*maturity*). Dojrzałość w IT kojarzy nam się zazwyczaj z analizą zdolności organizacji do efektywnej pracy. W odniesieniu do niezawodności jest rzadszym występowaniem awarii w coraz to dłuższym okresie użytkowania. W odniesieniu do człowieka moglibyśmy mówić o dojrzałości jako zdolności do nauki na własnych błędach. Jeśli więc wiemy, że dany czynnik powoduje awarię, to jego wyeliminowanie pozwoli zapobiec awarii lub też obniżyć jej wagę. Możemy zatem powiedzieć, że dojrzałość oprogramowania rośnie z każdą usuniętą awarią.
- Tolerowanie usterek (*fault tolerance*) polega na poprawnym działaniu oprogramowania mimo ich występowania. Wracamy tu do tzw. obsługi błędu. Jeśli dany moduł zwróci wyjątek, czyli nastąpi nieoczekiwane zachowanie, które może być awarią, to takie zdarzenie powinno uruchomić działania obsługujące. Jeśli

usterka nie jest poważna, to system powinien dalej funkcjonować.

- Odtwarzalność (*recoverability*). Poważny defekt w aplikacji może doprowadzić do awaryjnego wyłączenia systemu. Odtwarzalność jest w tym przypadku zdolnością aplikacji do poprawnego działania wraz z próbą przywrócenia danych. Przykładem może być tutaj tryb awaryjny w kolejnych wersjach Windows.
- Zgodność ze standardami niezawodności (*reliability compliance*) nieokreślonymi w normie.

3. Użyteczność (*usability*) to charakterystyka najbliższa potrzebom użytkowników końcowych aplikacji. Dotyczy głównie aplikacji z graficznym interfejsem użytkownika. Chcąc ją określić, analizujemy nie tylko użycie aplikacji, ale również stopień, w jakim wspiera użytkownika w jego działaniach. Specjaliści od użyteczności analizują program oraz kontekst jego użycia i środowisko. Na przykład zwiększony poziom hałasu musi wpływać na redukcję sygnałów dźwiękowych docierających do użytkownika, a nasłonecznienie miejsca, w którym znajduje się ekran aplikacji, zmusza do zwiększenia kontrastu. Użyteczności blisko więc do wzornictwa przemysłowego, ogólnego procesu optymalizacji wytwarzania czy nawet sztuki.

- Zrozumiałość (*understandability*) – podczas projektowania, budowania oraz testowania oprogramowania zadajemy sobie pytanie, czy użytkownik zrozumie, co się za nim kryje? Zrozumienie funkcjonalności jest kluczem do użycia narzędzia w określony dla niego sposób, ale również do uzyskania optymalnych rezultatów w najkrótszym możliwym czasie.
- Łatwość nauki (*learnability*). Prostych rzeczy uczymy się szybko, trudnych długo. Jeśli uczymy się czegoś długo – a w czasach internetu szybko się zniechęcamy – to produkt zostanie odrzucony i rozpoczną się poszukiwania łatwiejszego rozwiązania. Aby nie stracić klientów, musimy spowodować, że szybko osiągną cel lub biegłość w posługiwaniu się oprogramowaniem. Stosując elementy, które ludzie już znają (*Mnie się podobają melodie, które już raz słyszałem*) i wiedzą, jak działają, dajemy czytelny sygnał, że powielamy najlepsze (czytaj: „najszerzej stosowane”) wzorce.

## Kto nam powie, co jest użyteczne?

Czy odpowiedzi na pytanie o użyteczność lub używalność powinniśmy szukać osobiście, czy zapytać zainteresowanych, czyli osoby, które oprogramowania skorzystają, a może szukać innego rozwiązania?

Czy rodzic jest w stanie obiektywnie spojrzeć na swoje dziecko i powiedzieć: „W życiu nie widziałem brzydszego bachora!”. Absolutnie nie. Osoby niebędące rodzicami muszą mi uwierzyć na słowo. Nasze dziecko jest najpiękniejsze, najmądrzejsze i najszybciej się rozwija. Jeśli nawet dostrzegamy jakieś wady, to nie będziemy o nich głośno mówić, bo przecież wszystkie inne dzieci mają większe wady i na ich tle wady naszego dziecka błędną.

Czy jesteśmy w stanie testować własne oprogramowanie? Oczywiście tak, ale testy będą subiektywne i łatwo przy nich o pomyłkę. Testujący autor zazwyczaj definiuje sobie oczekiwany rezultat zachowania aplikacji. Test więc zazwyczaj zakończy się wynikiem pozytywnym („Działa tak, jak zaprojektowałem”).

Lepiej, żeby ktoś inny powiedział, co jest poprawnym zachowaniem naszego oprogramowania. Metoda konsultatywna zakłada, że istnieje ekspert, specjalista w danej dziedzinie, który wie więcej od nas. Możemy więc powiedzieć, co jest, a co nie jest użyteczne. Oczywiście przydatność eksperta zależy od jego doświadczenia i wiedzy.

Ostatni i najważniejsi w określaniu rzeczy jako użyteczne są użytkownicy. Kto, jeśli nie oni, wie, jak pewne rzeczy mają działać lub nie działać? Ile osób musimy zapytać? Pewnie nie mniej niż jedną, ale również nie wszystkich z jednej grupy (użytkowników dzieli się na grupy pod względem wieku, wykształcenia, doświadczenia itd.).

- Łatwość obsługi (*operability*). Wchodząc do sklepu po dowolny produkt, wybierzemy łatwy w użyciu czy trudny? Pewnie powinniśmy odpowiedzieć: najtańszy. Taką decyzję podejmuje większość klientów, ale z dwóch produktów o porównywalnej cenie wybierzemy łatwiejszy w użyciu. Programy, tak jak produkty ze sklepowej półki, mają instrukcje obsługi, ale są one: przeważnie napisane niezrozumiałym językiem, długie, nieprzymocowane do produktu na stałe, a ponadto w większości służą do zabezpieczenia producenta przed prawnymi konsekwencjami niepoprawnego stosowania produktu. Z rzeczy, których trudno się używa, korzystamy krótko lub wcale. Łatwe w użyciu zostają z nami na długo. Jeśli

chcemy sprzedawać produkt, musi on być intuicyjnie używalny.

- Atrakcyjność (*attractiveness*). Mówi się, że ludziom atrakcyjnym jest łatwiej. Szybciej znajdują partnera życiowego, dobrą pracę, ludzie bardziej im ufają. Z oprogramowaniem jest podobnie. Jeśli użycie oprogramowania sprawia nam przyjemność porównywalną z delektowaniem się złożonością maszyn Leonarda da Vinci lub też upajaniem obrazami Moneta, to znaczy, że blisko mu do dzieła sztuki. Wiele produktów sprzedaje się dzięki dobremu wyglądowi, a nie funkcjom, które oferuje. Produkty Apple'a są po prostu ładne, dlatego tylu ludzi się na nie decyduje. Podobnie będzie z atrakcyjnym interfejsem – łatwo „sprzeda” aplikację.
  - Zgodność ze standardami użyteczności (*usability compliance*) w normie nieokreślonymi. Mogą być to np. ISO 9241-11: „Guidance on Usability” czy ISO 13407: „Human-centred design processes for interactive systems”.
4. Efektywność (*efficiency*) jest zdolnością oprogramowania do wydajnego działania w zależności od liczby użytych zasobów i w określonych warunkach.
- Zachowanie w czasie (*time behaviour*) jest podstawową cechą oprogramowania wskazującą jego zdolność do udzielenia odpowiedzi lub przetwarzania w odpowiednim czasie. Podstawowym problemem większości organizacji jest umiejętne doprecyzowanie słowa „odpowiednie” i przypisanie do niego wartości liczbowych. Obszar związany z czasem odpowiedzi aplikacji jest jednym z bardziej krytycznych i jednocześnie najbardziej niedodefiniowanym przez zamawiającego oprogramowanie spośród wszystkich znanych mi atrybutów jakości.
  - Użycie zasobów (*resource utilisation*). W trakcie działania oprogramowanie korzysta z mocy obliczeniowej procesora, pamięci operacyjnej i dyskowej i wielu innych tzw. zasobów sprzętowych maszyny. Cechą optymalnej aplikacji jest użycie tylko takiej ilości zasobów, jaka jest rzeczywiście niezbędna, i zwalnianie ich, gdy już potrzebne nie są.
  - Zgodność ze standardami wydajności (*efficiency compliance*) nieokreślonymi w normie.

5. Pielęgnowalność lub raczej utrzymywalność (*maintainability*) jest to łatwość kontrolowania/monitorowania/modyfikowania oprogramowania w dłuższym okresie po jego dostarczeniu do środowiska produkcyjnego. Modyfikacje w fazie utrzymania oprogramowania związane są między innymi z naprawą defektów, dostosowaniem aplikacji do nowych wymagań lub też dostosowaniem do zmian zachodzących w otoczeniu (środowisku) aplikacji.
- Zdolność do analizy (*analyzability*), czyli zdiagnozowania wytwarzanego produktu pod kątem braków awarii lub ich przyczyn. Diagnoza może dotyczyć również rozpoznania obszarów aplikacji wymagających zmodyfikowania. Wykonanie tego badania jest podstawowym zadaniem osób odpowiedzialnych za utrzymanie aplikacji i jej dalsze rozwijanie.
  - Zdolność do zmian (*changeability*). Czy aplikację możemy łatwo zmienić? Łatwość zmiany może wynikać z jej odpowiedniej budowy, ale również z ilości informacji zawartych w samym kodzie źródłowym na poziomie nazw zmiennych czy też komentarzy do kodu.
  - Stabilność (*stability*), czyli zdolność oprogramowania do obsługi niespodziewanych zachowań pojawiających się przy użyciu lub też po jego modyfikacji.
  - Testowalność (*testability*) lub zdolność do bycia testowanym jest właściwością oprogramowania umożliwiającą walidowanie go po zmianach. Głównie ten aspekt pojawi się w kontekście ujawniania awarii. Jeśli wyobrazimy sobie oprogramowanie, które nie dostarcza nam informacji o swoim zachowaniu, trudno będzie określić, czy zachowuje się ono poprawnie. Gdyby koncept czarnej skrzynki maksymalnie rozciągnąć i powiedzieć, że oprogramowanie ma tylko wejście (*input*), ale nie ujawnia informacji na wyjściu (*output*), to mówilibyśmy o oprogramowaniu doskonale nietestowalnym.
  - Zgodność ze standardami utrzymania (*maintainability compliance*) czyli zdolność oprogramowania do podlegania standardom, konwencjom lub regulacjom prawnym. W normie brak wskazania norm.
6. Przenaszalność (*portability*), czyli łatwość, z jaką oprogramowanie może być przeniesione z jednego środowiska sprzętowego lub

programowego do innego. Środowisko może również oznaczać przeniesienie aplikacji z jednej organizacji do drugiej. Główny cel testów to uzyskanie dowodu, że poprawne działanie w nowym otoczeniu jest możliwe. Oczywiście w wielu przypadkach będzie to wymagało programowych modyfikacji elementów oraz dodatkowych testów osadzenia aplikacji w nowym środowisku.

- Zdolność adaptacyjna (*adaptability*). Czy aplikacja będzie zdolna dostosować się do nowego środowiska bez znaczących modyfikacji? Na przykład, czy będzie w stanie się przeskalować i dostosować do rozmiaru wyświetlacza w nowym środowisku?
- Instalowalność (*installability*), czyli zdolność oprogramowania do bycia zainstalowanym w konkretnym środowisku. Zazwyczaj dostawca oprogramowania wymienia środowiska obsługiwane i nieobsługiwane. Brak możliwości zainstalowania oprogramowania we wskazanym środowisku oznacza defekt w instalatorze lub brak testów dla danego obszaru.
- Koegzystencja (*co-existence*) jest zdolnością oprogramowania do działania z innym niezależnym oprogramowaniem we wspólnym środowisku, dzieląc wspólne zasoby. Rozwiązania sprzętowe niezmiernie rzadko przeznaczone są jedynie do jednego oprogramowania, zazwyczaj na jednej platformie mamy bardzo wiele różnego rodzaju aplikacji. Koegzystencją możemy nazwać współdzielenie przez nie zasobów platformy bez wchodzenia w konflikty.
- Zastępowalność (*replaceability*), czyli zdolność oprogramowania do bycia użytym w miejscu innego oprogramowania. Jeśli aplikacje mają takie same przeznaczenie w danym środowisku, to oznacza, że przy zachowaniu standardów odpowiednich dla tego obszaru powinna być możliwa wymiana aplikacji na inną. W najprostszym ujęciu wgranie nowej wersji oprogramowania powinno obsłużyć powiązania zrealizowane przez wcześniejszą wersję, która z kolei powinna zostać usunięta. Atrybut ten dotyczy również usunięcia ze środowiska aplikacji jednego producenta i zastąpienia jej inną bez konieczności zmiany całego środowiska, w którym pracujemy.
- Zgodność ze standardami przenaszalności (*portability compliance*) – w normie nie są one wskazane.

## Mnogość kategorii charakterystyk

Norma ISO 9126-1 wprowadza olbrzymią liczbę kategorii charakterystyk i subcharakterystyk. Takie działania wydają się błędne, jeśli nie przeanalizuje się podstawy problemu, który norma próbuje rozwiązać. Wiedząc, że weryfikacji muszą podlegać charakterystyki, staramy się testy odpowiednio podzielić, wskazując jednocześnie odpowiedzialność za nie. Jeśli pojawiają się w naszej specyfikacji testowej przypadki, których nie jesteśmy w stanie umieścić w żadnej z kategorii, zaczynamy daną formę kategoryzacji odrzucać. ISO zaproponowało więc wiele kategorii, aby każdy test znalazł odpowiednią grupę.

Następny kłopot pojawia się, gdy test kwalifikuje się do więcej niż jednej kategorii. Norma jest skończonym opisem i nie daje w pełni gotowego rozwiązania na każdą okoliczność. Odrzucenie standardów musi skłonić nas do stworzenia własnej kategoryzacji, właściwej dla danej organizacji czy nawet projektu. Jest to jednak praca kosztowna i w wielu przypadkach niepotrzebna.

### 5.3.4. Charakterystyki oprogramowania wg TheTest Eye

„Software Quality Characteristics” są autorstwa trzech panów związanych z testerskim blogiem [thetesteye.com](http://thetesteye.com): Rikarda Edgrena, Henrika Emilssona i Martina Janssona. Przetłumaczyłem je, ponieważ uznałem, że są ciekawą alternatywą dla charakterystyk sformalizowanych, znanych ze standardów. Postanowiłem je zaprezentować w tej książce w całości, co powinno dać czytelnikowi możliwość zestawienia ich z ciężkimi kategoriami ISO. Ich opis jest na tyle czytelny, że nie wymaga dodatkowego komentarza. W publikacji tej wyróżnia się osobno wewnętrzne charakterystyki jakości oprogramowania. Rozróżnienie po części wynika z białej i czarnej skrzynki opisanej wcześniej, po części związane jest z perspektywą różnych udziałowców projektu informatycznego. Panowie proponują następujące charakterystyki:

***Zdolność.*** *Czy produkt ma wartościowe funkcje?*

- *Kompletność: wszystkie ważne funkcje potrzebne użytkownikom końcowym są dostępne.*



- *Dokładność: dowolny wynik lub kalkulacja w produkcie są poprawne oraz prezentowane z użyciem dokładnych wartości.*
- *Efektywność: produkt wykonuje swoje czynności wydajnie (nie robiąc rzeczy, których nie powinien robić).*
- *Współpraca: różne funkcje współdziałają ze sobą w możliwie najlepszy sposób.*
- *Wielozadaniowość: produkt może wykonywać wiele zadań prowadzonych równolegle.*
- *Agnostycyzm danych: produkt obsługuje wszystkie możliwe formaty danych i radzi sobie z zakłóceniami.*
- *Rozwijalność: produkt umożliwia klientom lub osobom trzecim dodawanie nowych funkcji lub zmianę swoich zachowań.*

**Niezawodność.** Czy możesz zaufać produktowi w wielu trudnych sytuacjach?

- *Stabilność: produkt nie może powodować awarii, nieobsługiwanych wyjątków ani błędów skryptów.*
- *Odporność: produkt obsługuje przewidziane i nieprzewidziane błędy we właściwy sposób.*
- *Obsługa stresowych sytuacji: jak system radzi sobie w przypadku przekraczania różnych limitów?*
- *Odtwarzalność: istnieje możliwość odzyskania danych i dalszego korzystania z produktu po wystąpieniu błędu krytycznego.*
- *Spójność danych: wszystkie typy danych pozostają niezmiennie w całym produkcie.*
- *Bezpieczeństwo: produkt nie spowoduje żadnych strat w ludziach ani w mieniu.*
- *Odtworzenie po katastrofalnej awarii: co będzie, jeśli stanie się coś bardzo, bardzo złego?*
- *Zaufanie: czy zachowanie produktu jest spójne, przewidywalne i godne zaufania?*

**Użyteczność.** Czy produkt jest łatwy w użyciu?

- *Zachęcenie: produkt zachęca do odkrywania jego możliwości.*
- *Intuicyjność: czy łatwo jest zrozumieć i wyjaśnić, do czego służy ten produkt?*

- *Minimalizm: nie ma zbędnych elementów w zawartości i wyglądzie produktu.*
- *Uczenie się: można szybko i łatwo nauczyć się obsługi produktu.*
- *Zapamiętywanie: jeśli raz nauczyłeś się, jak coś robić, nie zapomnisz tego.*
- *Odkrywanie: informacje i potencjał produktu mogą być odkryte przez eksplorację interfejsu użytkownika.*
- *Operowanie: doświadczony użytkownik może wykonywać standardowe czynności bardzo szybko.*
- *Interaktywność: produkt ma łatwe do zrozumienia stany oraz możliwe interakcje wewnątrz aplikacji (przez GUI i API).*
- *Kontrola: użytkownik powinien mieć poczucie kontroli nad działaniem programu.*
- *Klarowność: czy wszystko jest przekazane w sposób szczegółowy, zrozumiały i niepodlegający wątpliwości?*
- *Błędy: komunikaty o błędach powinny być pomocne; trudno popełnić błąd, a popełniony łatwo naprawić.*
- *Spójność: zachowanie oraz wygląd są takie same w całym produkcie.*
- *Dopasowanie: domyślne ustawienia i zachowania mogą być elastycznie zmieniane.*
- *Dostępność: produkt może być użyty przez możliwie największą grupę użytkowników i jest zgodny ze standardami dostępności.*
- *Dokumentacja: dostępna jest Pomoc, która rzeczywiście pomaga i opisuje pełen zakres funkcjonalności.*

### **Charyzma.** Czy produkt ma „to coś”?

- *Unikatowość: produkt wyróżnia się na tle innych i ma coś, czego nie mają inne produkty.*
- *Satysfakcja: jakie są twoje wrażenia po skorzystaniu z tego produktu?*
- *Profesjonalizm: czy produkt ma właściwy poziom profesjonalizmu i sprawia wrażenie odpowiedniego do realizacji celu?*
- *Atrakcyjność: czy wszystkie aspekty produktu oddziałują na wzrok i inne zmysły?*
- *Ciekawość: czy użytkownicy będą na tyle zainteresowani, by sprawdzić, co mogą osiągnąć, korzystając z produktu?*
- *Zaciekawienie: czy użytkownik „połknął haczyk”, czy korzystał z produktu z przyjemnością, będąc w pełni zaangażowany?*

- *Nowinki*: czy produkt używa najnowszych i najlepszych technologii oraz pomysłów?
- *Oczekiwania*: produkt wyprzedza oczekiwania oraz zaspokaja potrzeby, których użytkownicy nawet nie byli świadomi.
- *Perspektywa*: czy produkt i informacje w nim zawarte są poprawnie skonstruowane pod względem języka i stylu?
- *Kierunkowość*: czy aplikacja robi dobre (pierwsze) wrażenie?
- *Historia*: czy istnieją interesujące historie o produkcie, jego powstaniu, konstruowaniu i użytkowaniu?

**Bezpieczeństwo.** Czy produkt jest zabezpieczony przed niepożądanym użyciem?

- *Autentykacja*: produkt identyfikuje użytkownika.
- *Autoryzacja*: obsługa tego, co użytkownik może zobaczyć oraz zrobić w produkcie.
- *Prywatność*: zdolność do nieujawniania chronionych danych nieautoryzowanym użytkownikom.
- *Luki zabezpieczeń*: w produkcie nie ma luk możliwych do wykorzystania przy użyciu socjotechnik.
- *Tajność*: produkt w żadnym wypadku nie ujawnia informacji o technologiach leżących u podstawy systemu.
- *Nienaruszalność*: zdolność stawiania oporu w razie prób penetracji.
- *Brak wirusów*: produkt nie ma wirusów i nie jest identyfikowany jako wirus.
- *Odporność na piractwo*: brak możliwości nielegalnego kopiowania oraz dystrybucji oprogramowania lub kodu.
- *Zgodność*: przestrzeganie obowiązujących standardów.

**Wydajność.** Czy produkt jest wystarczająco szybki?

- *Zdolność*: różne ograniczenia w produkcie dla różnych okoliczności (np. niska przepustowość).
- *Utylizacja zasobów*: właściwe zarządzanie pamięcią, przestrzenią dyskową i innymi zasobami.
- *Reagowanie*: szybkość wykonania (lub postrzegania wykonania) zadania.
- *Dostępność*: system jest dostępny do użycia wtedy, kiedy powinien.
- *Przepustowość*: zdolność produktu do przetwarzania wielu rzeczy.

- *Wytrzymałość*: produkt może wytrzymać obciążenie przez długi czas.
- *Informacja zwrotna*: czy informacja zwrotna wysyłana przez system w odpowiedzi na działanie użytkownika jest odpowiednia?
- *Skalowalność*: jak dobrze produkt skaluje się w przypadku wzrastającego i malejącego obciążenia?

**IT-arność.** Czy produkt jest łatwy do zainstalowania, utrzymania i wspierania jego dalszego rozwoju?

- *Wymagania systemowe*: zdolność do uruchomienia produktu na wspieranych konfiguracjach i obsługa różnych środowisk lub brakujących komponentów.
- *Instalowalność*: produkt może zostać zainstalowany na zdefiniowanej platformie oraz w odpowiedniej przestrzeni.
- *Aktualizacje*: łatwość zaktualizowania do nowszej wersji bez utraty konfiguracji i ustawień.
- *Dezinstalacja*: czy wszystkie pliki (z wyjątkiem plików użytkownika lub plików systemowych) i inne zasoby zostają usunięte podczas dezinstalacji?
- *Konfiguracja*: czy instalacja może być konfigurowana na różne sposoby i dopasowana do różnych lokalizacji, co pomaga w dalszym użytkowaniu produktu przez użytkownika?
- *Dostarczenie*: dział IT może dostarczyć produkt użytkownikom różnego typu oraz do różnych środowisk.
- *Utrzymywanie*: czy produkt i jego składowe mogą być łatwo utrzymane i rozwijane przez klienta?
- *Testowalność*: jak dostarczony produkt może być efektywnie testowany przez klienta?

**Kompatybilność.** Jak dobrze produkt współdziała z innym oprogramowaniem i środowiskami?

- *Kompatybilność sprzętowa*: produkt może zostać użyty w odpowiednich konfiguracjach sprzętowych.
- *Kompatybilność z systemem operacyjnym*: produkt może zostać uruchomiony z wyspecyfikowanymi wersjami systemu operacyjnego i zachowuje się w typowy sposób.

- *Kompatybilność aplikacyjna*: produkt i jego dane współpracują z innymi aplikacjami, których klient może używać.
- *Kompatybilność konfiguracyjna*: zdolność oprogramowania do rozpoznania konfiguracji środowiska.
- *Kompatybilność wsteczna*: czy produkt robi wszystko to, co jego wcześniejsza wersja?
- *Kompatybilność przyszła*: czy produkt będzie zdolny do użycia artefaktów oraz interfejsów przyszłych wersji?
- *Zrównoważony rozwój*: wpływ na środowisko np. wydajność energetyczna, wyłączanie, tryb oszczędzania energii, telepraca.
- *Zgodność ze standardami*: produkt jest zgodny z odpowiednimi standardami, regulacjami, prawem lub etyką.

### **Wewnętrzne charakterystyki jakości oprogramowania**

Zaprezentowane tu charakterystyki nie są bezpośrednio zauważalne dla końcowego użytkownika, jednak mogą być równie ważne dla powodzenia produktu.

**Wsparcie.** Czy użytkowanie i związane z nim problemy klienta mogą być rozwiązywane przez grupę wsparcia?

- *Identyfikacja*: czy można łatwo zidentyfikować części produktu, ich wersje lub konkretne defekty?
- *Diagnostyka*: czy znalezienie szczegółów problemów dotyczących klienta jest możliwe?
- *Rozwiązywalność problemów*: czy można łatwo wskazać defekty (np. w logach) i uzyskać pomoc?
- *Debugowanie*: czy w razie potrzeby można zaobserwować wewnętrzne stany oprogramowania?
- *Wszechstronność*: możliwość użycia produktu na więcej sposobów niż wstępnie zaprojektowano.

**Testowalność.** Czy można łatwo sprawdzić i przetestować produkt?

- *Śledzenie*: produkt loguje zdarzenia na właściwym poziomie i w użytecznym formacie.
- *Kontrolowalność*: zdolność do niezależnego definiowania stanów, obiektów oraz zmiennych.

- *Obserwowalność*: zdolność do obserwowania elementów, które powinny zostać poddane testom.
- *Monitorowanie*: czy produkt podpowiada, co się z nim dzieje?
- *Izolacja*: możliwość testowania pojedynczych części produktu.
- *Stabilność*: zmiany oprogramowania są kontrolowane i niezbyt częste.
- *Automatyzacja*: czy w oprogramowaniu są publiczne lub ukryte interfejsy programistyczne, z których można skorzystać?
- *Informacja*: umożliwienie testerom nauczenia się tego, czego trzeba.
- *Audyt*: czy produkt i to, co on tworzy, może być walidowane?

**Utrzymanie.** Czy produkt może być utrzymywany i rozszerzany niskim kosztem?

- *Elastyczność*: możliwość modyfikacji produktu na żądanie klienta.
- *Rozszerzalność*: czy w przyszłości będzie łatwo dodawać nowe funkcje?
- *Prostota*: kod nie jest bardziej skomplikowany niż to konieczne; nie utrudnia projektowania, uruchomienia i ewaluacji.
- *Czytelność*: kod jest właściwie opisany, łatwy do odczytania i zrozumienia.
- *Przejrzystość*: czy można łatwo zrozumieć strukturę bazową?
- *Modułowość*: kod jest podzielony na dające się kontrolować elementy.
- *Refaktorowalność*: czy testy jednostkowe są według Ciebie wystarczająco dobre?
- *Analiza*: możliwość znalezienia przyczyn defektów lub innego interesującego nas kodu.

**Przenośność.** Czy przenoszenie oprogramowania do innych środowisk i języków jest możliwe?

- *Ponowne użycie*: czy części produktu mogą być ponownie użyte w innym miejscu?
- *Adaptacja*: czy daje się łatwo zmieniać produkt, by mógł działać w innym środowisku?
- *Kompatybilność*: czy produkt współgra ze standardowymi interfejsami lub oficjalnymi standardami?
- *Umiędzynarodowienie*: czy można łatwo przetłumaczyć produkt?

- *Lokalizacja: czy części produktu można dopasować do potrzeb docelowych kultur lub państw?*
- *Odporność interfejsu użytkownika: czy produkt będzie wyglądał tak samo dobrze po przetłumaczeniu?*

Wprawny tester może tych charakterystyk użyć jako źródła pomysłów do weryfikacji poprawności działania oprogramowania lub też jako listy kontrolnej. W przypadku listy możemy zapisy umieścić w tabeli i odpowiedzieć sobie na pytanie, czy oprogramowanie spełnia dane kryterium atrybutów jakości. Dokument można znaleźć na stronach The Test Eye.

### 5.3.5. Charakterystyki oprogramowania wg Jamesa Bacha

Proponowana klasyfikacja Bacha nie do końca opisuje charakterystyki oprogramowania, ale może być łatwo przekuta na testowanie atrybutów. Pan James mówi tutaj o heurystykach testowalności oprogramowania, czyli pewnego rodzaju liście kontrolnej pozwalającej nam określić, czy jesteśmy gotowi do testowania oprogramowania. Jest to również informacja dla wytwórców oprogramowania, czy dostarczyli nam wystarczających podstaw do testów.

Przytoczony tu tekst jest tłumaczeniem pracy Jamesa Bacha (Satisfice, Inc.).

***Kontrolowalność*** (controllability) – *im lepiej kontrolujemy oprogramowanie, tym więcej możemy zautomatyzować i zoptymalizować.*

- *Istnieje możliwy do opisanie skryptami interfejs lub środowisko testowe umożliwiające sterowanie testowanym oprogramowaniem.*
- *Stany oprogramowania, stany sprzętu i zmienne mogą być kontrolowane przez testera.*
- *Moduły, obiekty, warstwy funkcjonalne oprogramowania mogą być testowane oddzielnie.*

***Obserwowalność*** (observability) – *to, co widzimy, może być przetestowane.*

- Wcześniejsze stany systemu i zmiennych są widoczne lub możliwe do znalezienia (np. logi transakcji).
- Zróżnicowany wynik jest generowany dla każdej informacji wejściowej.
- Stany systemu i zmienne są widoczne lub możliwe do wyszukania podczas uruchamiania testów.
- Wszystkie czynniki wpływające na wynik są widoczne.
- Niepoprawny wynik jest łatwy do zidentyfikowania.
- Wewnętrzne defekty są automatycznie wychwytywane i raportowane przez samotestujący mechanizm.

**Dostępność** (availability) – *aby testować, musimy mieć gotowy obiekt testów.*

- System ma niewiele defektów.
- Defekty nie blokują uruchamiania testów.
- Produkt jest rozwijany w cyklach funkcjonalnych (co pozwala na jednoczesne tworzenie oprogramowania i jego testowanie).
- Istnieje dostęp do kodu źródłowego.

**Prostota** (simplicity) – im prostsze oprogramowanie, tym mniej testów.

- Projekt oprogramowania jest wewnętrznie spójny.
- Aplikacja jest prosta funkcjonalnie (np. minimalna liczba funkcji, aby spełnić wymagania).
- Aplikacja jest prosta strukturalnie (np. moduły są spójne i luźno sprzężone).
- Kod jest prosty (na tyle, że zewnętrzny obserwator może go efektywnie zrewidować).

**Stabilność** (stability) – im mniej zmian, tym mniej problemów w testowaniu.

- *Zmiany w oprogramowaniu nie są częste.*
- *Zmiany w oprogramowaniu są kontrolowane i komunikowane.*
- *Zmiany w oprogramowaniu nie powodują konieczności zmiany automatów testowych.*



**Informacja** (information) – im więcej informacji mamy, tym mądrzej testujemy.

- *Projekt oprogramowania jest podobny do innych produktów, które już znamy.*
- *Technologia użyta do wytworzenia produktu jest możliwa do zrozumienia (w skończonym czasie – przyp. tłum.).*
- *Zależności między wewnętrznymi, zewnętrznymi i współdzielonymi komponentami są możliwe do zrozumienia.*
- *Cel działania oprogramowania jest zrozumiały.*
- *Użytkownicy oprogramowania mogą być łatwo zrozumiani.*
- *Środowisko, w którym aplikacja będzie używana, jest zrozumiałe.*
- *Dokumentacja techniczna jest dokładna i dostępna.*

Przez proste linkowanie założeń autora do charakterystyk znanych z innych źródeł rzeczywiście możemy dostrzec podobieństwo. Łatwo dopatrzymy się w tych cechach oprogramowania użyteczności w prostocie czy utrzymywalności w obserwowalności.

---

## Zadanie

Dla dowolnego oprogramowania wyróżnij charakterystyki, które mają dla niego znaczenie. Wykorzystaj klasyfikację, która jest ci najbliższa.

---

## 5.4. Testy potwierdzające

Osobną kategorię, choć jednocześnie podkategorię innych testów, stanowią testy potwierdzające. Zaliczają się do niej retesty, tzn. sprawdzenie, czy z oprogramowania wyeliminowano wcześniej znalezione defekty, oraz testy regresywne, które mają potwierdzić, że oprogramowanie po zmianach ciągle działa. Co ważne, są to testy, których wykonanie wiąże się bezpośrednio ze zdarzeniami wokół procesu twórczego. Powodem wykonania tych testów jest zazwyczaj tzw. zmiana. Jeśli zastanawiacie się, dlaczego testy, które realnie nie stanowią osobnej kategorii, stały się podstawą do jej stworzenia, to mogę podać tylko jedno uzasadnienie: ponieważ testy te są absolutnie podstawowe dla powodzenia projektu informatycznego.

### 5.4.1. Retesty

Retest jest ściśle związany z weryfikacją poprawności implementacji poprawki defektu. Biorąc definicję na chłopski rozum, jest to ponowne wykonanie już raz wykonanego testu. Nie jest to precyzyjne określenie, ale pewne uproszczenie. Dlaczego? Dlatego, że retest nie musi być konsekwencją testu. Zagmatwane? Przeanalizujmy drogę do retestu.

- Krok „przedprzedostatni”: znaleziony defekt.
- Krok przedostatni: poprawka w oprogramowaniu.
- Krok ostatni: retest.

Czy defekt jest znajdowany jedynie w wyniku testów? Oczywiście nie. Może się objawiać awarią oprogramowania dopiero w fazie jego użytkowania. Skoro defekt nie został znaleziony podczas testów, w takim razie nie możemy mówić o przypadku testowym, który defekt znalazł. Pewna szkoła testowania mówi jednak, że ujawniony defekt powinien zostać opisany przez przypadek testowy. Przypadek ten powinien być regularnie uruchamiany, co zagwarantuje (prawdopodobnie), że defekt nie wróci, czyli nie pojawi się po stronie klienta. Współcześnie retestem nazywamy weryfikację, czy defekt został naprawiony, niekoniecznie ponownym wykonaniem testu, co odbiega od klasycznej definicji ze *Słownika wyrażений związanych z testowaniem*.

Przypadki testowe i *stricte* testowanie odnoszą się do oprogramowania, więc ponowne wykonanie testu wiąże się z uruchomieniem oprogramowania, czyli jest testowaniem dynamicznym (opisanym dalej).

### 5.4.2. Testowanie regresywne

Uruchomianie testów z ujawnieniem lub też niewykrywaniem defektów będzie testowaniem regresywnym. Głównym celem i założeniem tzw. regresji jest upewnienie się, że oprogramowanie ciągle działa poprawnie. Oczywiście podstawą do tego, by uznać, że oprogramowanie może przestać działać poprawnie, będzie jego bezpośrednia modyfikacja lub pośrednia zmiana w jego otoczeniu. Ogólnie przyjmujemy, że uruchomienia testów regresji wymaga „zmiany”. Może ona dotyczyć:

- samego kodu oprogramowania – najbardziej oczywista ze zmian;

- platformy sprzętowej – niekoniecznie musi to być duża zmiana, wystarczy nawet drobnostka w rodzaju podłączenia nowego (nawet lepszego) układu pamięci;
- oprogramowania w otoczeniu aplikacji – aktualizacja wersji systemu operacyjnego lub zastosowanie nowej wersji bazy danych jest już ważną zmianą, mogącą wpływać na stabilność czy poprawność działania naszego systemu.

Najbardziej oczywiste testy dotyczą zmodyfikowanych obszarów. Musimy jednak pamiętać, że testowane powinny być również obszary powiązane z obszarem zmodyfikowanym. Jeśli nasza wiedza na temat wewnętrznej struktury powiązań jest niewielka, jak w testach czarnej skrzynki, to nie uda nam się osiągnąć kompletności testów. Ale w przypadku testów białej skrzynki, gdy znamy powiązania między klasami, komponentami czy funkcjami, oczywiste staje się wyizolowanie obszarów powiązanych. Niestety testy regresyjne na poziomie interfejsów zewnętrznych oprogramowania stają się formą przypadkowego strzelania w obszary powiązane lub też testowaniem losowo wybranych obszarów. Gdy nasze testowanie jest czarnoskrzynkowe, możemy się posilkować wiedzą innych członków zespołu projektowego, w tym aby identyfikować powiązania między obszarami aplikacji. Dobrze skonstruowane organizacje programistyczne z wysoką świadomością jakości kodu opisują wydawane wersje oprogramowania notami wydania (*release notes*). W notcie takiej otrzymujemy podstawowe informacje, które każdy tester może przekuć w testy:

1. Jakie nowe funkcje zostały dodane?
2. Jakie defekty zostały naprawione?
3. Jakie obszary mogą być powiązane ze zmianami i jaki to może mieć wpływ na stabilność i poprawność działania oprogramowania?

Zestawy przypadków testów regresyjnych to przede wszystkim te, które pokrywają istotne obszary funkcjonalne systemu. Ich wykonywanie jest formą skanowania czy też precyzyjnego monitorowania jakości oprogramowania. Mogą one, w zależności od szkoły testerskiej lub podejścia, być rozszerzane o:

- przypadki testowe, które ujawniły już wcześniej defekt (mówimy wówczas o permanentnym reteście);
- przypadki testowe, które weryfikują typowe błędy programistyczne;
- przypadki testowe, które wiążą się ze zidentyfikowanymi ryzykami.

### Dlaczego regresja?

Regresja to cofanie się (nie mylić z błędnym, acz popularnym „cofaniem się do tyłu”). To również uwstecznianie się. Testy regresji mają regresowi zapobiec, ale jednocześnie nie powodują progresji, czyli ruchu do przodu.

### Zadanie

Wyszukaj w internecie rzeczywistą „notę wydania” lub *release notes* i przeanalizuj ją pod kątem zakresu testów do przeprowadzenia.

## 5.5. Testowanie statyczne i dynamiczne

Granica między testami statycznymi i dynamicznymi, choć klarownie wyznaczona samymi definicjami, nie jest oczywista dla adeptów testowania. Nie ma się jednak co łudzić, że przekroczenie tej granicy ma jakieś realne znaczenie. Przecież określenie ram pracy nie będzie się opierało na teoretycznych i wydumanych definicjach. Zakres prac będzie definiowany potrzebą lub celem, który chcemy osiągnąć. Na dobry początek w tab. 5.4 podaję porównanie głównych cech testowania statycznego i dynamicznego.

**Tab. 5.4.** Zestawienie cech testowania statycznego i dynamicznego

Testowanie statyczne	Testowanie dynamiczne
Nie wymaga uruchomienia oprogramowania	Wymaga uruchomienia oprogramowania
Wspiera przede wszystkim weryfikację	Wspiera przede wszystkim walidację
Wyszukuje raczej defekty	Wyszukuje raczej awarie
Może być wspierane listami kontrolnymi i procesami	Często wymaga zaprojektowania przypadków testowych
Wykonywane przed kompilacją	Wykonywane po kompilacji

### 5.5.1. Testowanie statyczne

Pierwszym skojarzeniem z elementem statycznym wydaje się bezruch. Statyczny oznacza nieporuszający się. Blisko stąd już do testowania statycznego, które jest weryfikacją niepowiązaną z ruchem (uruchomieniem) działającego oprogramowania. Testowanie statyczne wiąże się więc bezpośrednio z najwcześniejszym możliwym zaangażowaniem testera, kiedy system nie może lub też nie musi działać. Testowanie statyczne dzielimy na dwie podstawowe podkategorie:

1. Przeglądy, czyli czytanie ze zrozumieniem i rewidowanie wszelkiego rodzaju dokumentacji projektowej, specyfikacji wymagań czy kodu źródłowego. Przez wiele osób testowanie statyczne nie jest traktowane jako testy i trudno z tą opinią polemizować. Dużo zależy od tego, co definiujemy jako testowanie. Przyjmujemy jednak, że przeglądanie statyczne przez testerów, mających szeroką wiedzę na temat funkcjonowania oprogramowania, jest wartościowe. Czynności te, choć bliższe ogólnemu procesowi zapewnienia jakości w projekcie, są ważne ze względu na opisany w poprzednich rozdziałach moment wczesnego zaangażowania. Testowanie dokumentacji, jak wskazuje definicja, to kontrola jakości, w tym dokładności, prawidłowości i kompletności wszelkiego rodzaju specyfikacji, które powstają w procesie wytwarzania oprogramowania. Przeglądy dzielą się na dalsze podkategorie, ale nie będą one tutaj opisane. Warto jednak zauważyć, że praca statyczna testera koncentruje się wokół dokumentacji pisanej w języku naturalnym, z kolei praca statyczna programisty to głównie dbanie o dokument zapisany językiem kodowym.
2. Analiza statyczna to proces automatycznego sprawdzenia dokumentacji, w tym dokumentów kodowych (kodu źródłowego). Analizie podlegają wszystkie dokumenty zrozumiałe dla maszyny lub też takie, w których można wskazać powtarzalne reguły. Człowiek ma naturalną skłonność do automatyzowania czynności

nudnych i żmudnych. Nie ma w tym nic złego. Musimy tylko uwzględnić, że maszyny funkcjonujące automatycznie nie są w stanie działać tak doskonale, jak robi to człowiek posługujący się mózgiem. Subtelna różnica polega na obsłudze sytuacji wyjątkowych. Aplikacja może się nauczyć odtwarzać powtarzalne czynności, więc łatwo zaprogramować robota spawającego samochód na taśmie produkcyjnej. Człowiek z kolei potrafi się odnaleźć na nowych, w jego opinii, nieprzemierzonych jeszcze ścieżkach. Jeśli podczas produkcji piwa na linii napełniania butelek nagle nastąpi korek, reakcje maszyny i człowieka będą zupełnie różne. Człowiek zdiagnozuje problem, zatrzyma taśmę, przerwie napełnianie. Maszyna będzie próbowała ślepo kontynuować, butelki zaczną się spiętrzać i pękać aż do momentu, kiedy tego spektaklu – ściskającego za serce każdego piwosza – nie przerwie pracownik obsługujący taśmę. Jest to zdarzenie realne, które ludzie o mocnych nerwach każdego dnia mogą zobaczyć podczas wizyty w zautomatyzowanych browarach.

#### **Dlaczego rozróżnia się terminy „testowanie” i „analiza”?**

Podział bierze się z historycznych uprzedzeń programistów do testowania oprogramowania jako czynności, którą może wykonać odpowiednio wytresowana małpa. Tak więc programiści nie testują, oni analizują.

Programy nie radzą sobie z sytuacjami nieoczekiwanymi lub nieprzewidzianymi przez człowieka podczas pisania ich kodu. Stąd też na przykład niewielka przydatność automatycznie generowanego programu, w tym kodu skryptów automatycznych testów.

Analiza statyczna dotyczy przede wszystkim kodu, a przeprowadza się ją za pomocą narzędzi sprawdzających logikę kodu, jego „czystość” i przejrzystość (np. liczba komentarzy na wiersze kodu) czy po prostu poprawność składni. W ramach analizy statycznej stosujemy następujące techniki:

- analizę przepływu sterowania – szukamy w kodzie elementów, które mogą spowodować, że aplikacja nie będzie w stanie pomyślnie zakończyć swojego działania, np. z powodu nieskończonych pętli czy nieosiągalnych kroków procesu;

- analizę przepływu danych – bada w kodzie procesy definiowania, użycia i usuwania zmiennych;
- analizę spójności i naruszeń standardów, czyli zgodności danego dokumentu z jego innymi częściami lub innymi dokumentami (ręczne testowanie spójności będzie formą testowania statycznego).

Typowymi narzędziami stosowanymi do analizy są analizatory statyczne, które umożliwiają również kalkulowanie metryk, np. złożoności kodowej. Zazwyczaj mówimy o mechanizmach, które działają jak kompilatory czy interpretatory lub po prostu nimi są. Z narzędziami tymi wiążą się poważne problemy natury projektowej. Zazwyczaj analizatory zwracają olbrzymią ilość ostrzeżeń. Nie wszystkie ich przyczyny źródłowe muszą być wyeliminowane, by aplikacja działała. Niektóre naruszenia standardów po prostu nie mają znaczenia. Przykładem może być naruszenie standardów kodowania, kiedy chcemy, aby nasze oprogramowanie działało na aplikacjach „historycznych”, jak Internet Explorer 6. Podstawowym problemem jest określenie, w jakim zakresie naruszenia opisane ostrzeżeniem wpłyną w przyszłości na działania aplikacji i które z nich muszą koniecznie zostać wyeliminowane.

### **Kompilacja kontra interpretacja**

W przypadku aplikacji kompilowanych kod jest z wyprzedzeniem tłumaczony z języka kodowego na maszynowy i zapisywany w postaci pliku uruchamialnego. W przypadku interpretacji kodu źródłowego jest on przetwarzany na bieżąco, bez kompilacji.

Problemem testowania statycznego jest określenie, co jest uruchomieniem kodu, a co nim nie jest. Jest to znacznie łatwiejsze w odniesieniu do kodu kompilowanego niż w odniesieniu do kodu niekompilowanego. Próbując znaleźć zakres, możemy powiedzieć, że:

- statyczne testowanie kodu kompilowanego kończy się w momencie jego kompilacji i uruchomienia programu,
- statyczne testowanie kodu niekompilowanego kończy się w miejscu, w którym jest on już zinterpretowany, np. przez przeglądarkę.

W przypadku kodu niekompilowanego możemy dużo łatwiej odwołać się do kodu źródłowego niż w przypadku kompilacji.

Testowanie statyczne przeważnie poprzedza testy dynamiczne, a usunięcie defektów znalezionych podczas testowania statycznego jest zazwyczaj dużo tańsze.

### 5.5.2. Testowanie dynamiczne

Testowanie dynamiczne wymaga wykonania kodu przez uruchomienie skompilowanego programu lub przez interpretację kodu za pomocą odpowiedniego narzędzia.

Dla wielu testowanie dynamiczne jest kwintesencją testowania. Uruchomienie oprogramowania, w sposób niekoniecznie sformalizowany czy przygotowany, zapewne będzie najczęstszym użyciem w ramach definicji testowania.

#### **Testowanie białoskrzynkowe jest dynamiczne czy statyczne?**

Dla wielu praktyków testowania oprogramowania odpowiedź wydaje się oczywista, ale dla innych w pewnym stopniu kontrowersyjna. Niestety po raz kolejny spór toczy się na poziomie definicji i zakresu. Generalnie uznaje się, że testowanie komponentów za pomocą zaślepek (*stubs*) jest formą uruchomienia oprogramowania. Zaśleпки pozwalają symulować zewnętrzne komponenty czy też aplikacje, a to z kolei tworzy dość realne środowisko działania oprogramowania. Co prawda, nie jest to kompilacja, lecz interpretacja, ale nie zmienia to faktu, że poszczególne wiersze kodu są uruchamiane. Podobnie sprawa ma się z debugowaniem i użyciem debagerów. Debugowanie jest formą „odpalenia” kodu i kwalifikuje się jako test dynamiczny.

Warto na koniec przypomnieć ważną różnicę między testami statycznymi i dynamicznymi. Testy statyczne, ze względu na swój poziom wczesnego zaangażowania, szukają defektów, z kolei w testowaniu dynamicznym naszym głównym celem jest znalezienie awarii, czyli konsekwencji pojawienia się defektów.

## 5.6. Zestawienie testów

Rozdział rozpocząłem od wskazania niekonsekwencji w dzieleniu testowania. Wyróżniane typy testów są konstrukcjami sztucznymi i nie



tworzą jednoznacznych i odseparowanych od siebie elementów. Wprost przeciwnie. Nachodzą na siebie i zakwalifikowanie jednego z testów tylko do jednego zbioru jest praktycznie niemożliwe. Zestawienie pokazuje, jak (w uproszczeniu) pewne testy można próbować klasyfikować.

Przykłady pokrycia typów testów z perspektywy testów funkcjonalnych:

- Wykonując testy funkcjonalne, wykonujemy w pewnym zakresie testy нефункционалне, np. testując formularz, sprawdzamy przyjazność komunikatów o błędach (testy użyteczności).
- Retesty wykonywane są dla poprawek defektów zgłoszonych w ramach testów funkcjonalnych.
- Testy regresywne wykonuje się po zmianach w funkcjonalności, weryfikowanych przez testy funkcjonalne. Część testów funkcjonalnych stanowi więc zbiór testów regresywnych.
- Testowanie funkcjonalne wymaga dynamicznego uruchomienia oprogramowania. Niektóre potencjalne problemy funkcjonalne będą widoczne w specyfikacji (np. dwa różne opisy działania tej samej funkcji). Część defektów funkcjonalnych będzie widoczna podczas analizy kodu (np. niepoprawnie skonstruowana pętla).
- Testy dynamiczne to zbiór testów zawierający również testy funkcjonalne.
- Testy funkcjonalne wykonywane są zarówno na poziomie czarnej skrzynki (np. testy interfejsu użytkownika), jak i białej skrzynki (np. testy poprawności działania funkcji w kodzie).

Przykłady pokrycia typów testów z perspektywy testów нефункционалных:

- Przeprowadzając testy нефункционалне, wykonujemy w ograniczonym zakresie testy funkcjonalne, np. testując wydajność oprogramowania, symulujemy użycie funkcji systemu.
- Wykonywane są również retesty poprawek defektów zgłoszonych w ramach testów нефункционалных.
- Testy regresywne wykonuje się po modyfikacjach oprogramowania związanych z znalezionymi defektami нефункционалными (np. defektami niezawodności).

- Testowanie niefunkcjonalne wymagają dynamicznego uruchomienia oprogramowania. Część problemów niefunkcjonalnych może zostać ujawniona, zanim uruchomimy oprogramowanie (np. nieoptymalne zapytania bazodanowe).
- Testy dynamiczne to zbiór zawierający testy niefunkcjonalne.
- Testy niefunkcjonalne wykonywane są często na poziomie czarnej skrzynki (np. testowanie przeciążenia). Całe kategorie testów niefunkcjonalnych należą do testów białej skrzynki (np. utrzymywalność oprogramowania).

W przypadku znalezienia każdego typu defektu zaleca się sprawdzenie, czy został usunięty i czy awaria już nie występuje. W związku z tym każdy typ testów podlega retestom. Nie jest to może popularny w testowaniu statycznym zabieg, ale przy analizie testowalności wymagań często zdarza się, że po zgłoszeniu defektu wymagany i po przeformułowaniu testu sprawdzamy, czy rzeczywiście naniesiona poprawka koryguje zapisy.

Testy regresywne są również w dużej mierze kojarzone z każdym dynamicznym uruchomieniem oprogramowania i mówimy zazwyczaj o sprawdzeniu działania oprogramowania po wprowadzeniu zmian w nim lub w środowisku. Każdy dynamicznie uruchomiony typ testów może być weryfikowany w ramach testów regresywnych.

**Tab. 5.1.** Relacje między typami testów

	Testy funkcjonalne	Testy niefunkcjonalne	Retesty	Testy regresywne	Testy statyczne	Testy dyna
Testy funkcjonalne		Pokrywają częściowo	Wymagają	Wymagają	–	Mogą realiz jako
Testy niefunkcjonalne	Pokrywają częściowo		Wymagają	Wymagają	–	Mogą realiz jako
Retesty	Są wykonywane dla	Są wykonywane dla		–	–	Są wyko dla
Test regresywny	Są wykonywane dla	Są wykonywane dla	–		–	Są wyko dla

<b>Testy statyczne</b>	–	–	–	–		–
<b>Testy dynamiczne</b>	Są wykonywane dla	Są wykonywane dla	Wymagają	Wymagają	–	
<b>Testy czarnoskrzynkowe</b>	Są wykonywane dla	Są wykonywane dla	Wymagają	Wymagają	–	Są wyko dla
<b>Testy białoskrzynkowe</b>	Są wykonywane dla	Są wykonywane dla	Wymagają	Wymagają	–	Są wyko dla

Zbiór testów statycznych ze względu na swoją naturę jest zbiorem niezależnym od zbioru testów dynamicznych. Podobnie jest z testami białej oraz czarnej skrzynki.

Całościowo relacje między typami testów widać w tab. 5.1, w której elementy w wierszach wchodzą w zdefiniowaną relację z elementami w kolumnach.

---

## Zadanie

Na dowolnej aplikacji z dostępną specyfikacją lub kodem źródłowym przeprowadź testy statyczne i dynamiczne.

---

## 6. Zawód: tester

### 6.1. Wprowadzenie

„Praktyka testowania się nie rozwija”, „Testowanie jest nudne, wykonywane ręcznie i powtarzalne”, „Testerzy nie są aż tak kompetentni jak programiści”, „Testerów dużo łatwiej wymieniać niż programistów, „Tester to nie jest wartościowy zawód, to niepotrzebny pośrednik między użytkownikami aplikacji i programistą”. Takie i ostrzejsze sformułowania można usłyszeć na konferencjach zarówno testerskich, jak i programistycznych. Nie możemy uznać, że jest to tylko odpowiedź na pytanie, jak programiści postrzegają testowanie. Coraz częściej w podobny sposób testerów manualnych postrzegają testerzy automatyczni.

Jednak w mojej opinii testowanie to ciągle zawód, który ma przyszłość i przyszłość ta jawi się w jasnych barwach. Oto pięć pierwszych powodów, dlaczego warto zostać testerem lub też lepszym testerem.

1. Dla testerów jest praca.
2. Testowanie nie jest trudne. Przynajmniej na początku kariery nie tak trudne jak programowanie, a może przynieść porównywalne dochody.
3. Testowanie można mieć we krwi, ale można je w sobie wykształcić. Zdolność do testowania jest raczej zestawem cech i specyficznym nastawieniem niż wrodzoną umiejętnością. Zdolność tę można łatwo wykształcić. W życiu wystarczy być dokładnym i cierpliwym i już połowa drogi do zawodu została pokonana. Przyda się również umiejętność posługiwania się technologią, angielski i zdolność do komunikowania się ze światem zewnętrznym – druga połowa drogi zaliczona. W dodatku podobno każdy może być testerem, bo testowanie wpisane jest w nasze codzienne życie. Ten, kto urodzi się

z zestawem właściwych cech, może zostać Van Gogiem testowania, ten, kto zawodu się nauczy, będzie może malarzem pokojowym testów.

4. Testowanie niesie radość. Kiedy znajdujesz defekt, cieszysz się, że uda się go wyeliminować i klient go nie zobaczy. Kiedy nie znajdujesz defektów, cieszysz się, że wszystko działa poprawnie.
5. Testowanie jest po prostu czystym dobrem dla innych. Testerzy są nie tylko linią obrony kontrolującą jakość aplikacji, ale zajmują poważne miejsce w eliminowaniu przeszkód i usprawnianiu nieintuicyjnych funkcjonalności, które mogą wprowadzać użytkownika w błąd.

### **Zawód: tester**

Tester oprogramowania jest oficjalnym zawodem uznanym przez Ministra Pracy i Polityki Społecznej. Co więcej, mamy dwa typy testerów oprogramowania (numer na początku to kod zawodu):

„251903 Tester oprogramowania komputerowego,

251904 Tester systemów teleinformatycznych”.

Zawód testera jest podkategorią następujących klas zawodów:

„2 SPECJALIŚCI

25 Specjaliści do spraw technologii informacyjno-komunikacyjnych

251 Analitycy systemowi i programiści

2519 Analitycy systemowi i specjaliści do spraw rozwoju aplikacji komputerowych gdzie indziej niesklasyfikowani”.

## **6.2. Edukacja testerska**

Każda osoba stająca dziś przed wyborem drogi zawodowej powinna odpowiedzieć sobie na kilka pytań:

- Czy matura zagwarantuje mi pracę?
- Czy studia na danym kierunku zagwarantują mi pracę?
- Czy dyplom zagwarantuje mi pracę?
- Czy nie lepiej zamiast edukacji szkolnej wybrać pracę na ciekawym stanowisku?

Postaram się na te pytania odpowiedzieć, choć wiem, że każdy musi to zrobić samodzielnie. Kontekst i struktura rynku pracy zmieniają się tak dynamicznie, że moje uzasadnienie może być nieaktualne w dniu, w którym czytasz te słowa.

### 6.2.1. Edukacja szkolna

Szkolna edukacja testerska nie istnieje. Polskie szkoły nie uczą niczego praktycznego i przydatnego, dlatego więc miałyby robić wyjątek dla testowania? Każdy, kto się łudzi, że aby nauczyć się testować, trzeba pójść do szkoły testowania, musi poczuć rozczarowanie, że Wyższa Szkoła Testowania nie istnieje. Co więcej, w dowolnej szkole trudno będzie znaleźć kierunek studiów określony jako testowanie oprogramowania. Łatwiej za to znaleźć przedmiot: testowanie oprogramowania. Zazwyczaj jest to jeden semestr i ok. 30 godzin.

Nikt nie wątpi, że edukacja sama w sobie jest ważna dla rozwoju jednostki. Warto się jednak zastanowić, czy edukacja (w tym edukacja z testowania oprogramowania), którą oferuje nam zarówno państwo polskie, jaki i instytucje prywatne, jest rzeczywiście wartościowa.

Problemy z rekrutacją, które ma dziś wiele firm w Polsce, pokazują, jak trudno obecnie o dobrego pracownika. Mimo narzekań wielu osób na problemy z zatrudnieniem, wiele firm informatycznych ma realny problem w znalezieniu wartościowych pracowników. Czy powodem jest za niska pensja? A może mało ciekawe projekty? Na czym polega problem? Otóż problem leży po stronie edukacji.

**Problem 1.** Polski system kształcenia nie jest w stanie odpowiedzieć na zapotrzebowanie rynku. Mamy więc wielu specjalistów od funduszy europejskich, wykwalifikowanych nauczycieli, ekonomistów, którzy nie mogą znaleźć pracy, ponieważ rynek aktualnie jest nimi przesycony.

**Problem 2.** Edukacja (zwłaszcza tzw. wyższa) się skomercjalizowała. Nie ma więc znaczenia, czy student po danym kierunku znajdzie pracę, czy też nie. Liczy się, aby płacił za naukę lub aby państwo płaciło za dany kierunek. Uczelnie oczywiście będą otwierały kierunki, na które jest zapotrzebowanie, ale opóźnienie względem rynku często sięga kilku lat. Łatwiej jest również promować istniejące kierunki, z opisanym programem, kadrą nauczycielską, niż inwestować w nowe, w których wszystko trzeba konstruować od nowa.

**Problem 3.** Dyplom większości polskich uczelni nic nie znaczy. Kiedy stoisz po drugiej stronie (reprezentujesz pracodawcę) i szukasz pracowników, uczelniany papier przestaje mieć znaczenie. Tak wiele osób ma dziś wyższe wykształcenie, że o wyborze pracownika decyduje nie dyplom, a osobowość i doświadczenie zawodowe.

**Problem 4.** Uczelnie nie przygotowują do pracy zawodowej. Zatrudniając osobę po studiach, pierwsze miesiące jako pracodawca poświęcasz na wyeliminowanie złych nawyków nabytych na studiach i na pokazanie, jak tzw. przemysł czy biznes w rzeczywistości wygląda.

**Problem 5.** Uczelnie to zamknięta forteca. Wiele mówi się o ich otwartości na pomysły i pomoc ze strony przedsiębiorców. Na podstawie własnego doświadczenia stwierdzam, że kiedy mówi się „sprawdzam”, po drugiej stronie ciągle jest beton jak w głębokim PRL-u. Wewnętrzne struktury uczelni są tak skostniałe, że aby je eksplorować, trzeba doświadczonych archeologów.

Czy warto w takim razie inwestować w studia? Powiem TAK, jeśli chcesz i możesz decyzję o pracy podjąć za jakiś czas. Studia dzienne polecam wszystkim, którzy chcieliby przedłużyć sobie dzieciństwo, poczuć wolność ciągłego imprezowania i niewielerobienia. Jeśli jednak myślisz o rozwoju, wybierz samodzielną edukację lub studia zaoczne. Takie, które nie przeszkadzają ci w karierze i nie przesuną zdobywania doświadczenia na „za cztery lata”, ale takie, które będą wspierały twoją praktykę.

Ludzie w wieku nastu lub dziesięciu lat ciągle jeszcze poszukują i zastanawiają się, kim chcieliby zostać. Trudno określić jednoznacznie, co się będzie chciało robić od dziś do końca życia. Sam jestem przykładem człowieka, którego politechniczne wykształcenie radiokomunikacyjne w niewielkim zakresie przygotowało do pracy w testowaniu oprogramowania. Na swojej drodze spotkałem wielu ludzi, którzy ciągle poszukując, wreszcie znaleźli się tam, gdzie chcieli być. Niekoniecznie poprzez edukację szkolną. Jeśli nie wiesz jeszcze, czy zawód testera (lub jakiegokolwiek inny) jest dla ciebie, to zamiast grać na czas, idąc na studia, wypróbuj się w pracy. Mama powie: „Idź na studia, co ci szkodzi”; „Liczy się papier”. Nie wierz w to. Papier już dawno przestał mieć znaczenie. Uczelnie są jedynie fabrykami do zarabiania pieniędzy i generowania coraz ciekawszych nazw zawodów w papierach rzęsy bezrobotnych.

Często spotykam się z opinią, że tester to zawód dla osób z wykształceniem technicznym. Jeśli rzeczywiście dla osób

technicznych, to „jak bardzo techniczny” powinien być tester? Wykształcenie mocno techniczne kojarzy nam się z politechnikami. Czy tester musi ukończyć politechnikę? Nie. Tak jak ludzie bez wykształcenia polonistycznego mogą przecież pisać książki (żywy przykład w waszych rękach). Tester nie jest też zawodem korporacyjnym ze strukturą kastową, jak w przypadku prawników. Nie jest to zawód dziedziczny, jak widzimy to w rodzinach lekarzy. To, czy ktoś nadaje się do tego zawodu, czy też nie, nie jest warunkowane ani pochodzeniem, ani wykształceniem.

Na poparcie moich słów przytoczę dostępne w internecie źródło Wirtualny informator maturzysty, który testera opisuje następująco:

1. *Praca Testera oprogramowania polega na przeprowadzaniu testów funkcjonalności sprzętu IT w konkretnej firmie – bzdura. Definicja za wąska, nieobejmująca testowania jako weryfikacji jakości oprogramowania również na poziomie niefunkcjonalnym oprogramowania (użyteczności czy też bezpieczeństwa itp.).*
2. *Testowanie oprogramowania jest ściśle związane z procesem wytwarzania oprogramowania komputerowego – bardzo celna uwaga.*
3. *Testowanie oprogramowania komputerowego polega nie tylko na weryfikacji prawidłowego działania systemów, ale także, a właściwie przede wszystkim, na odkrywaniu powodów nieprawidłowego ich funkcjonowania – mała bzdura. Testerzy raczej szukają problemów, ale odkrywanie powodów to już nie nasz obszar działania.*
4. *Do zadań osoby zatrudnionej na takim stanowisku należą:*
  - *tworzenie scenariuszy i przypadków testowych,*
  - *przeprowadzanie testów z zakresu white/black box,*
  - *opracowywanie dokumentacji testowej,*
  - *raportowanie błędów,*
  - *współpraca z programistami – dosyć poprawnie, aczkolwiek pojęcia wymieszane losowo.*
5. *Wobec kandydatów ubiegających się o stanowisko Testera oprogramowania najczęściej stawia się następujące wymagania:*
  - *wykształcenie wyższe elektroniczne lub informatyczne – pożądane,*
  - *znajomość procesu tworzenia oprogramowania – pożądane,*
  - *znajomość podstaw obiektowych języków programowania (Java, C++) – wiedza z zakresu programowania może się okazać*



- przydatna,
- *posiadania certyfikatu z zakresu testowania (ISEB, ISTQB)* – pożądane tylko u wybranych pracodawców,
  - *umiejętności automatyzowania procesów testowania, (...)* – nie wszędzie,
  - *znajomość zagadnień HRM, (...)* – bzdura,
  - *znajomość baz danych Oracle oraz MSSQL, (...)* – pożądane,
  - *umiejętność analitycznego myślenia. (...)* – pożądane jak wszędzie.
6. *Osoba pracująca na stanowisku Tester oprogramowania otrzymuje średnio pensję mieszczącą się w przedziale 3000 – 4000 zł netto, przy czym wynagrodzenie takiego pracownika jest najczęściej prowizyjne (ok. 20% stanowi premia przyznawana za wyniki pracy)* – bzdura do kwadratu, bo zależy gdzie i jaką pracę się wykonuje.

W mojej opinii takie publikacje zamykają zawód testera oprogramowania zamiast go otwierać. Definiując wygórowane żądania wobec maturzystów, dobitnie przekonują ich do pójścia jednak na marketing i reklamę. W testowaniu nie ma się czego bać. Powtórzę: wiedza techniczna może być przydatna, ale jej brak nie dyskwalifikuje uczestników procesu testowego.

### 6.2.2. Edukacja internetowa

Szacuje się, że koszt (dla rodziców) edukacji jednego potomka to ok. 200 tys. złotych. Z kolei koszt studiowania materiałów w internecie, koszt udziału w otwartych projektach testerskich, w spotkaniach dla testerów, nawet koszt egzaminu ISTQB są nieporównywalnie mniejsze. Różnica jest jedna, za to zasadnicza – ta druga droga może pomóc ci znaleźć pracę. Ta pierwsza może cię zostawić z bezwartościowym papierem.

Dobrodziejstwem, choć niektórzy mówią przekleństwem, naszych czasów jest globalna sieć. Oprócz takich oczywistości, jak pirackie oprogramowanie, filmy i filmiki, pornografia, dostarcza nam również wielu platform do nauki. Co ciekawe, wiele z nich bezpłatnie albo za relatywnie niewielką cenę. Możliwa jest również nauka testowania i kodowania bez opłat. Większość materiałów będzie oczywiście w języku angielskim, jako drugim po HTML-u najbardziej uniwersalnym języku sieci. Jednym z przykładów nauki testowania za darmo

w internecie jest kurs dostępny na stronach [UDACITY.com](https://udacity.com). Czy będzie istniał, kiedy czytasz tę książkę, nie wiem, ale jeśli nie ma go pod tym adresem, to z dużym prawdopodobieństwem jest gdzieś – mniej lub bardziej legalnie – „zarchiwizowany”. Wiem również, że jeśli nie istnieje, na jego miejscu na pewno powstało 10 innych kursów. Aktualnie dostępne kursy zawierają wiele przydatnych w edukacji elementów, takich jak filmy, teoria, ćwiczenia i zadania.

Jeśli twoje testowanie ma być bardziej techniczne, to skorzystaj z możliwości nauki na Codecademy. Dostępne są tam kursy w obszarach JavaScript, Python i ogólnie Web. Warto wypróbować ich Get Started, aby się przekonać, czym jest Codecademy.

Kursy to jedno, ale warto również śledzić blogi testerskie. Niestety najciekawsze z nich są w języku angielskim, więc jeśli jeszcze go nie znasz, to najwyższy czas ten błąd naprawić. Szczególnie polecamy zbiorowy blog skandynawskich testerów The Test Eye oraz bardzo systematycznie uzupełniany blog EuroStar, gdzie oprócz wpisów pojawiają się również webinaria i e-booki. Warto również śledzić gwiazdy testowania i ich aktywność w sieci: Exploring Uncertainty Iaina McCowatta, Satifice Jamesa Bacha czy DevelopSense Michaela Boltona. Wartościowych miejsc jest więcej, ale nie sposób je wszystkie tutaj opisać. Korzystaj więc z wyszukiwarki, by znaleźć dalsze ciekawe miejsca.

A jeśli już zdecydujesz się poświęcić czas na naukę testowania, to dziel się tym, czego się nauczysz. Pisz. Najlepiej stwórz blog. Dlaczego o tym mówię? Ponieważ tak zaczynałem swoją karierę. Kiedy pisałem jeden z pierwszych blogów testerskich w Polsce [testerzy.pl](https://testerzy.pl), miałem świadomość niszowości, ale ta marka otworzyła mi w przyszłości wiele drzwi i stała się przepustką do własnej działalności. Mimo upływu lat jest to ciągle jeden z niewielu żywych testerskich blogów. Jeśli tylko utrzymasz pewien poziom jakości wpisów i regularności, możesz szybko zyskać czytelników na rynku, na którym ciągle mało kto dzieli się swoją wiedzą. Takie doświadczenie świetnie wygląda w CV i wiele o tobie mówi. Przede wszystkim, że testowanie jest twoją pasją i poświęcasz jej więcej niż przeciętny tester. Jeśli nie masz czasu na własny blog, publikuj w innych źródłach, byle tylko artykuły były podpisane twoim nazwiskiem. Takimi publikacjami również można błysnąć przed pracodawcą.

### 6.2.3. Edukacja przez praktykę

Stawiając w tym miejscu po raz kolejny pytanie, jak zostać testerem, możemy dać tylko jedną odpowiedź: trzeba poczuć ten zawód. Trzeba spróbować swoich sił w testach, zarówno formalnych, jak i mniej sformalizowanych. Można testować online w wielu miejscach i sprawdzić, czy nam się to podoba. Trzeba pójść na kilka rozmów kwalifikacyjnych i sprawdzić, czego się wymaga od testera. Wtedy zrozumiesz, czy się w tym zawodzie odnajdziesz, czy też nie.

Gdybym otrzymał złotówkę za każdym razem, kiedy jestem pytany, od czego zacząć swoją karierę w testowaniu, miałbym tysiące. Stukrotnie przeanalizowana i powtórzona odpowiedź, spisana na kartach tej książki, pozwoli mi, mam nadzieję, odpowiadać na to pytanie rzadziej. Jeśli nie masz jeszcze podstaw teoretycznych testowania, to je uzyskaj. Czytanie tego zdania jest w mojej opinii dużym krokiem we właściwym kierunku.

Jeśli masz podstawy teoretyczne, to uzupełnij je praktyką, a naprawdę dobrą praktyką jest testowanie dla kogoś, kogo interesują twoje zgłoszenia. Życzę ci z całego serca, aby był to pracodawca, który nie mogąc na rynku znaleźć doświadczonych testerów, inwestuje w młodych adeptów. Nie zawsze jednak tak jest, więc może warto rozejrzeć się za stażem, w ostateczności również bezpłatnym. Można żartować, że pracuje się za wpis do CV, ale jeśli rzeczywiście trafisz do projektu informatycznego, to możesz śmiało mówić o inwestycji w siebie. A może okaże się, że testowanie nie jest dla ciebie i zdecydujesz się pójść inną ścieżką? Jeśli na rynku nie ma powyższych ofert, pozostaje ci potestować w modelu *pay-per-bug* wraz z innymi testerami. W internecie jest wiele stron kontaktujących osoby chcące testować z twórcami oprogramowania. Część z nich proponuje wynagrodzenie dopiero po znalezieniu defektu. Może nie zarobisz kokosów, ale skontaktujesz się z zamawiającym oprogramowanie i z innymi testerami. Już samo czytanie zgłoszeń innych osób jest wartością. Możesz się dowiedzieć, jak raportować defekty i gdzie szukać ciekawych problemów. Ostatnią deską ratunku, jeśli zawiodą wszystkie opcje opisane wcześniej, jest testowanie projektów o otwartym kodzie (*open source*). Gdzieś zawsze znajdzie się grupa pasjonatów, którzy zupełnie za darmo, dla siebie i dla osób sobie podobnych tworzą aplikację. Programiści zaangażowani w taki projekt będą potrzebowali pomocy testerów. I ty takim testerem możesz się stać, a jednocześnie przysłużyć

się społeczności. Starannie jednak wybieraj projekty. Unikaj martwych lub elitarnych, bo twoje zgłoszenia mogą się nigdy nie doczekać obsłużenia. Nikt na nie nie odpowie, nikt się nimi nie zainteresuje. Pracuj w obszarze, na którym się znasz albo w którym chcesz się rozwijać. Testowanie aplikacji z obszaru, który kompletnie cię nudzi albo którego naturalnie nie rozumiesz, może się zakończyć bolesnym rozczarowaniem. Sprawdź również otoczenia projektu. Czy jest grupa użytkowników tego oprogramowania? Czy są już jakieś potwierdzone wdrożenia? Te informacje pomogą ci zagwarantować, że praca, którą wykonasz, komuś się przysłuży. Warto również wybierać projekty znane. Lepiej brzmi „Pracowałem przy Androidzie” niż „Pracowałem przy AplikacjaDoZamawianiaObiadów”.

Pamiętaj, że każde z powyższych doświadczeń możesz wpisać do CV. Będą one na pewno lepiej wypełniać jego przestrzeń niż hobby czy informacje o szkole podstawowej i średniej, które ukończyłeś.

#### **6.2.4. Podsumowanie**

Z testowaniem jest tak samo jak z każdym innym zainteresowaniem. Warto spróbować w nim sił, jeśli tylko jest dla ciebie ciekawe. Jeśli marzysz o podróżach, możesz oczywiście pójść na turystykę, ale czy nie lepiej wsiąść do pociągu i pojechać w świat? W testowaniu oprogramowania twoim pociągiem może być np. testowanie wersji beta.

Zawód testera jest takim samym zawodem jak każdy inny. Wymaga przygotowania. Najlepiej wybrać studia pokrywające się z zainteresowaniami testerskimi. Jeśli twoim konikiem jest testowanie aplikacji, najlepiej zacząć od inżynierii oprogramowania. Jeśli bliżej ci do testowania sprzętu, raczej wybierz elektronikę.

### **6.3. Certyfikacja testerska**

Certyfikacja również jest pewną formą edukacji. Zazwyczaj dynamiczną i skrótową, ale przynajmniej taką, która pozostawi po sobie trwały ślad w CV. Pochwalenie się dobrze ugruntowanym i rozpoznawalnym certyfikatem naprawdę potrafi otworzyć wiele drzwi i pozwala skorzystać z wielu skrótów podczas rekrutacji. W Polsce szczególnie. Jak śpiewał Kazik Staszewski, jesteśmy krajem „koncesyjno-etatystycznym”,

a twoją koncesją na pracę w testowaniu staje się papier potwierdzający zdanie egzaminu.

Moglibyśmy udawać, że istnieje wiele systemów certyfikacji, ale prawda jest taka, że monopol na tym rynku ma certyfikat ISTQB. To on jest wymagany albo pożądaný w ogłoszeniach o pracę i to o nim się wspomina w przetargach dla sektora publicznego. Pamiętajmy, że certyfikaty zdobywa się bardziej dla innych niż dla siebie. Po co nam papierowe potwierdzenie naszej wiedzy, skoro sami mamy jej świadomość. To inni muszą przyjąć do wiadomości, że ją posiadamy. Możemy im oczywiście to udowodnić podczas rozmowy kwalifikacyjnej, ale możemy im również machnąć papierem przed oczami i o dziwo ta druga opcja działa lepiej.

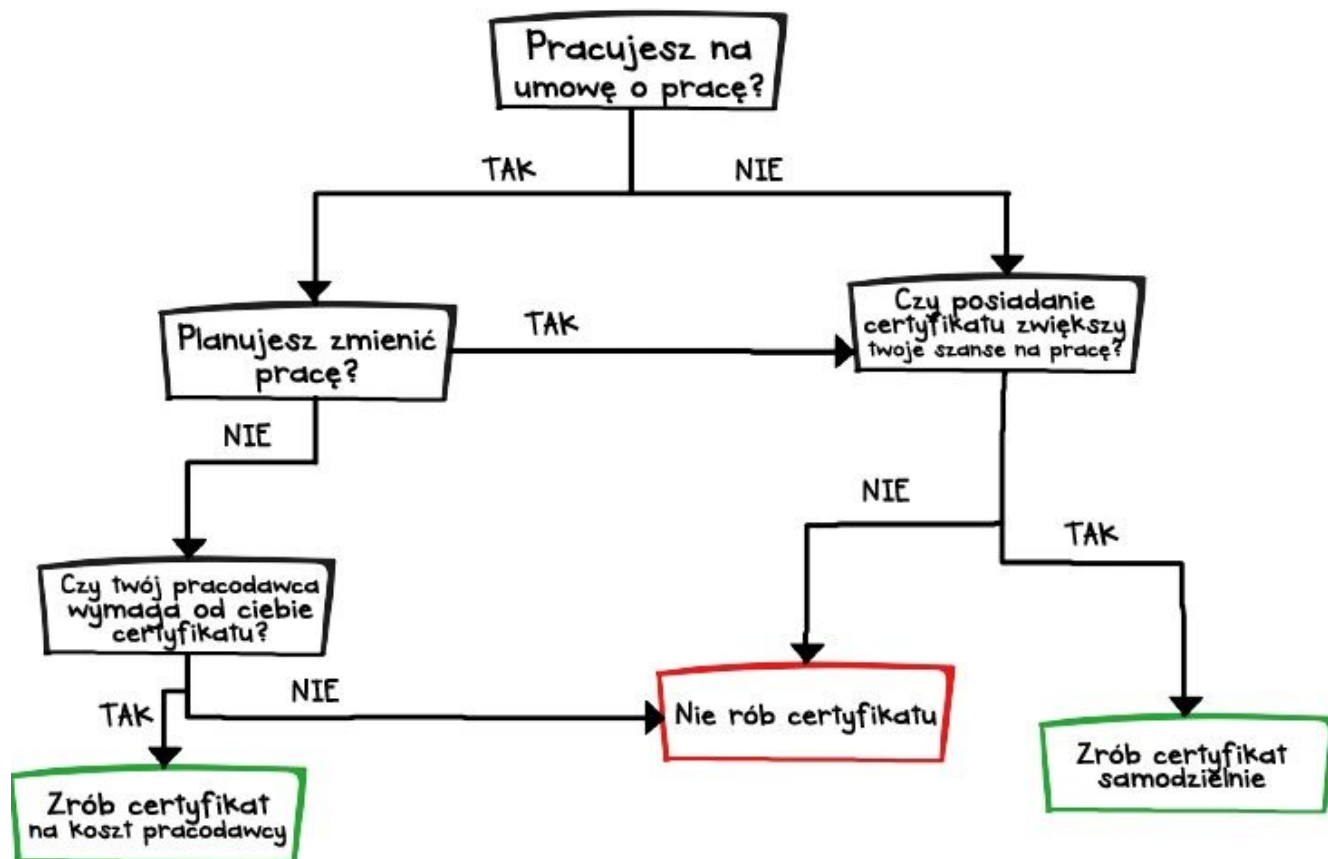
Wielu ludzi, krytykując certyfikaty, nie dostrzega właściwej perspektywy. Aby je zrozumieć, należy przeanalizować tryb ich powstania:

- Istnieje potrzeba certyfikacji w danym obszarze lub istnieje potrzeba zarobienia pieniędzy poprzez certyfikaty.
- Organizacja ze zdefiniowaną potrzebą definiuje certyfikat.
- Certyfikat jest udostępniany wraz z możliwością akredytowania trenerów, materiałów, szkoleń oraz wraz z sprzedawalnym produktem, jakim jest egzamin.
- Certyfikat zdobywają osoby, które potrzebują certyfikatu.
- Certyfikat staje się popularny i wiele organizacji decyduje się na certyfikację.
- Certyfikat staje się standardem, a następnie wymogiem.
- Osoby, które na początku nie chciały się starać o certyfikat, i tak go zdobywają ze względu na presję rynku.
- Ścieżka certyfikacji jest rozwijana przez tworzenie nowych poziomów.

Widać tu logiczny ciąg oparty na potrzebie. W Polsce istnieje potrzeba skrótów w przetargach czy też rekrutacji, stąd tak duża popularność certyfikacji. Zdobywamy więc certyfikaty dla pracodawców, zlecających prace, klientów naszych usług itd.

Kiedyś popełniłem błąd pokazany na rys. 6.1. Może nie jest on w całkiem poważny, ale w mojej opinii obrazuje potrzebę i brak potrzeby certyfikacji.

Struktura certyfikacji ISTQB jest dość złożona, ale początkujący tester nie musi się w nią zagłębiać. Ciągłe jedynym ważnym i wymaganym certyfikatem jest poziom podstawowy. Jeśli z jakiegoś powodu zdecydujecie się na inny poziom certyfikacji, to spokojnie możecie ścieżki certyfikacji eksplorować później. Głównym kluczem jest ISTQB PP i jeśli czujesz, że potrzebujesz skrótu w rozwoju testerskim, to krótszej drogi już chyba nie ma.



Rys. 6.1. Graf wspierający decyzję o certyfikacji

Ważne w certyfikacji ISTQB jest to, że nie wymaga uczestniczenia w kursie przygotowującym do egzaminu. Możesz go zdawać po samodzielnym przygotowaniu. Pomogą w tym ogólnodostępne materiały, w tym sylabus, słownik oraz przykładowe pytania. Czy bez żadnego doświadczenia i wiedzy w testowaniu można podchodzić do egzaminu? Teoretycznie rekomendowane są trzy miesiące doświadczenia w testach, ale w praktyce to jedynie deklaracja, której i tak nikt nie sprawdza. W moich kursach biorą udział osoby bez

żadnego doświadczenia i są w stanie po szkoleniu zdać egzamin. Zakładam, że średnio rozgarnięty użytkownik komputera zdeterminowany do nauki i pozyskania wiedzy w zawodzie może łatwo stać się posiadaczem certyfikatu. Jeśli jednak zdecydujesz się na kurs, warto pomyśleć o nim jako o inwestycji w siebie. Przyjdzie ci zapłacić ok. 3000 zł netto (uwzględniając koszty egzaminu) i warto od razu pomyśleć o tym, jak uzyskać zwrot tak wydanych pieniędzy. Warto mieć już na oku jakąś pracę albo stanowisko, na jakie możecie aplikować. Szkoda by było wydać pieniądze, a certyfikat schować do szuflady. Warto również dobrze wybrać kurs i egzaminującego. Teoretycznie wszystko jest takie samo, ale można, delikatnie mówiąc, źle trafić. I w naszej branży zdarzają się czarne owce. Ktoś weźmie pieniądze, ale kursu nie poprowadzi, albo egzamin poprowadzi, a certyfikat „zagubi się” w drodze do was. Zdarzają się również trenerzy, których jakość pracy stoi na nie najwyższym poziomie, i kursy, po których szanse na zdanie egzaminu mogą się tylko obniżyć.

W jakim języku zdawać? Warto wybrać egzamin w języku ojczystym, ale organizowany przez firmę zagraniczną. Zagwarantuje ona certyfikat w języku angielskim, co z kolei przełoży się na jego większą rozpoznawalność, gdy zdecydujecie się go użyć w zagranicznej firmie działającej w Polsce lub też wyemigrujecie za granicę. Szkolenia certyfikowane z testowania opierają się na jednym sylabusie i mają bardzo zbliżoną zawartość:

- zasady testowania (terminologia, dlaczego testowanie jest konieczne, podstawy procesu testowania, psychologia testowania, testowanie regresyjne, wyniki testów);
- test w różnych fazach procesu wytwarzania oprogramowania (modele procesu wytwarzania oprogramowania, ekonomika testowania, planowanie testowania, testowanie komponentów, testowanie integracyjne, testowanie systemu, testowanie własności, testowanie akceptacyjne, testowanie pielęgnacyjne);
- dynamiczne techniki testowania (techniki czarnej skrzynki, techniki białej skrzynki, zgadywanie błędów);
- statyczne techniki testowania (udział przeglądów w procesie testowania, rodzaje przeglądów i inspekcji, analiza statyczna);
- zarządzanie testowaniem (organizacja, zarządzanie konfiguracją, kontrolowanie testowania, śledzenie błędów, standardy w dziedzinie testowania);

- narzędzia wspomagające testowanie (rodzaje narzędzi, wybór i wdrożenie narzędzi wspomagających testowanie).

Nie jest to może wiedza powszechnie uznawana ani respektowana, ale pewna, wartościowa testerska perspektywa.

## 6.4. Testowanie oprogramowania ma swoich wrogów

Ciągle zbyt wiele osób uważa, że testerzy są niepotrzebni. Aby dobrze zrozumieć rację drugiej strony, należy dobrze się przyjrzeć wysuwany przez nią argumentom. W większości są to łatwo dające się obalić teorie, ale czasami można w nich znaleźć ziarno prawdy.

**Testowanie jest kosztowne.** Pomyślcie tylko, ile pieniędzy trzeba zainwestować w stworzenie od początku grupy testowej. Pomijając samych ludzi, mówimy tu o stanowiskach pracy, powierzchni biurowej i o narzędziach, w które trzeba ich wyposażyć. Oczywiście to tylko wstępne koszty. Są jeszcze pensje, benefity, premie i nagrody. Spróbujcie też oszacować koszty stworzenia sprawnego systemu wymiany danych między testerem i programistą. Może się okazać, że to znacznie więcej niż potrzeba na stworzenie stanowiska pracy programisty (patrz punkt „Programiści wiedzą więcej”). Pieniądze stają się najważniejszym przeciwnikiem idei testowania. A teraz połącz to z kolejnym faktem.

**Testerzy niczego nie produkują.** W przeciwieństwie do programistów, testerzy niczego namacalnego nie produkują, a ich praca nie przynosi żadnych wymiernych korzyści finansowych. Jedynym wynikiem pracy testera jest raport-donos, więc naturalnie jest on kojarzony z rozrostem biurokracji i w konsekwencji staje się znienawidzony przez programistów (patrz punkt „Testerzy się czepiają”).

**Programiści wiedzą więcej.** Programista nie tylko może znaleźć defekt, ale może również samodzielnie go naprawić. W mniej korzystnym przypadku wie po prostu, do kogo się zwrócić, gdy defekt odnajdzie. Kto lepiej sprawdzi kod źródłowy, jak nie sam programista? Kto może przeanalizować ewentualne wycieki pamięci, niekończące się pętle itd.? Przecież nie testerzy. Oni są jak ślepcy. Chodzą po omacku, po



ciemnym pokoju i roszczą sobie prawo do dawania wskazówek ludziom, którzy znają kod jak własną kieszeń.

**Użytkownik jest testerem.** Wysunięto swego czasu ideę testowania przez użytkownika, zwaną testowaniem alfa i beta. Kto lepiej niż użytkownik powie nam, co nie działa z jego punktu widzenia? Taki raport możemy uzyskać minimalnym kosztem. Użytkownicy-testerzy mogą wykonywać testy i w ramach nagrody dostawać za darmo nasz produkt pod koniec projektu. Mogą go mieć przed wszystkimi, co jest dodatkowym motywatorem. Nie zapomnijmy również o ich (pod)świadomości, że im więcej defektów znajdą, tym lepszy sprzęt lub oprogramowanie otrzymają w przyszłości. Kiedy już produkt wejdzie na rynek, dzięki erze internetu użytkownik łatwo może się podzielić z producentem uwagami o nim. Takie spostrzeżenia to podstawa do stworzenia jeszcze lepszego towaru w przyszłości. Testerzy, choć nie chcą się do tego przyznać, szybko wpadają w rutynę i zapominają, że to użytkownik końcowy jest najważniejszy, a nie oni sami. Jeśli przeanalizuje się ilość procesów, dokumentacji generowanej przez testerów, łatwo dochodzimy do wniosku, że nie da się w takim środowisku pracować dłużej i nie popaść w schematy prowadzące do omijania najbardziej krytycznych anomalii produktu.

**Testerzy się czepiają.** Pomyślmy o obciążeniu psychicznym i nienawiści, jaka powstaje między testerami i programistami. Nie da się ukryć, że jest to naturalna obrona przed ludźmi, którzy niczego nie wytwarzają. Ich jedynym celem jest wypominanie niedociągnięć (patrz punkt „Testerzy niczego nie produkują”). Trzeba zauważyć, że w wielu firmach stosuje się metodę mierzenia jakości testera liczbą znalezionych przez niego defektów. Wskaźnik ten przekłada się bezpośrednio na jego pensję. Jakież to ogromne pole do nadużyć! Z kolei słabość programisty mierzy się liczbą defektów znalezionych w wytworzonym przez niego oprogramowaniu. Stąd już tylko krok do napięć, wrogości i podgrzewania atmosfery w środowisku pracy. Efekt: szybko może spaść efektywność pracy i wzrosnąć opóźnienie dostarczenia produktu na rynek.

**Testowanie nigdy się nie kończy!** Z marketingowego punktu widzenia nigdy nie uda się nam niczego wypuścić na rynek, jeśli chcemy, żeby było to bezawaryjne. Z punktu widzenia testera nie ma lepszej pracy. Zawsze jesteś potrzebny!

Oczywiście, przytoczone tu stwierdzenia są przekoloryzowane i przesadzone, ale w swojej pracy możecie się z takim podejściem

spotkać. W książce tej rozprawiamy się z atakami na testowanie. Jeśli dobrze się z nią zapoznasz, w merytorycznej dyskusji obalisz mity, jakimi obrosło testowanie.

## 6.5. Cechy miękkie testera

Jakie cechy sprzyjają pracy w tym zawodzie? Są to m.in.:

- profesjonalny pesymizm w podejściu do testowanego obiektu,
- dokładność w wykonywaniu pracy (ale nie może to być perfekcjonizm),
- umiejętność komunikowania się ze wszystkimi osobami uczestniczącymi w projekcie informatycznym.

Mówi się, że testerem oprogramowania można być mimo braku tych cech, co widziałem już na wielu przykładach. Niestety, rzadko kiedy takie osoby są w stanie profesjonalnie wykonywać zawód. Zauważyłem, że do testowania trafia wiele osób przypadkowych, nieposiadających opisanych wcześniej cech, i szybko stających się zakałą zawodu. Nie tylko uwłaczają roli testera, ale jeszcze doprowadzają do tego, że inni uczestnicy projektu sceptycznie podchodzą do konieczności angażowania „takich” testerów.

Absolutnie najważniejszą cechą testerów oprogramowania jest poczucie humoru. Jeśli go nie masz, to poważnie zastanów się nad zmianą zawodu. Oto prosty test. Jeśli nie rozumiesz lub nie śmiesz cię zamieszczone tu żarty, to znaczy, że jeszcze nie znasz testowania wystarczająco dobrze.

*Pytanie: Ilu testerów potrzeba by wymienić żarówkę?*

*Odpowiedź: Żadnego. My znajdujemy problemy, od naprawiania są inni.*

*Pytanie: Skąd wiesz, że umawiasz się z testerem?*

*Odpowiedź: Twoje listy miłosne wracają do ciebie z zaznaczonymi błędami gramatycznymi.*

**Naklejki na zderzaki testerów oprogramowania wg [StickyMinds.com](http://StickyMinds.com)**

Amerykanie lubią manifestować swoje przekonania lub informować o sobie przez naklejki na zderzaki. Cytowane tu naklejki musieli nakleić

testerzy z poczuciem humoru.

*To są defekty. Jeśli ci się nie podobają... mam inne.*

*Zawsze jest jeden defekt więcej.*

*Czy twoja matka wie, jak piszesz kod?*

*Ufaj, ale weryfikuj.*

*Testowanie to zorganizowany sceptycyzm.*

*Jedyne pewniki w życiu: śmierć, podatki i bugi w kodzie!*

*„U mnie działa”.*

*Testerzy oprogramowania nie psują go, dostają je już zepsute.*

*Mylić się jest rzeczą ludzką, znalezienie pomyłki wymaga testera.*

*Definicja aktualizacji: weź stare defekty i dodaj kilka nowych.*

*Testować czy nie testować – tu nie ma pytania*

*Każdy kod jest winny, dopóki nie udowodni niewinności.*

*Znajdź bugi lub umrzyj, próbując!*

*Nie tworzymy defektów, znajdujemy twoje.*

*Masz buga? Znajdę go!*

*Testowanie oprogramowania: bug zatrzymuje się tu.*

*Wierzmy w Boga, a wszystko inne testujemy.*

## 6.6. Trudne aspekty pracy testera

Z testowaniem oprogramowania wiąże się wiele przyjemnych elementów, ale może ono być również ciężkim kawałkiem chleba. Właśnie trudne aspekty podnoszone są przez przeciwników osobnej roli testera lub przez osoby, które chcą deprecjonować rolę testów w organizacji. We współczesnych firmach takich trolli antytesterskich jest coraz mniej, ale nie znaczy to, że na nich nie traficie. Część z nich zostało jedynie zagłuszonych przez poprawność polityczną i czeka na swoją okazję, aby wypunktować testerów za brak skuteczności lub jakościowo słabą pracę. W mojej opinii najgorsze, co może spotkać testera w organizacji, to:

- Próba testowania wszystkiego i zawsze, bez względu na zakres zmian w wersji czy bez względu na ograniczenia czasowe.
- Próba ślepego klikania oprogramowania, które samo z siebie nie pozwala nam zrozumieć logiki swojego działania albo po prostu mamy na to za niskie kompetencje.

- Testowanie w zmieniającym się środowisku, gdzie najgorszym przypadkiem jest testowanie czegoś, co właśnie jest modyfikowane, np. przez programistów.
- Wielokrotne testowanie tego samego bez widocznego celu.
- Praca w sytuacji stresowej, gdy próbuje się nam wmówić, że wszystko zależy od nas, a w dodatku niepowodzenie będzie naszą winą.
- Praca w atmosferze obwiniania testera za niską jakość.

Uda się uniknąć przez dłuższy czas złych zadań testerskich, jeśli trafimy na dobrego lidera. Większość problemów testerskich rozwiązuje się po prostu dobrym zarządzaniem. Dobry kierownik testów będzie chronił swoich ludzi przed wykonywaniem „małpiej” (*monkey testing*) czy monotonnej pracy. Przykładowo minimalizacja nakładu pracy na regresję jest wynikiem dobrej współpracy testera z programistą. Współpraca ta jest jednak zazwyczaj wbudowana w organizację jako proces lub jako szkielet wzajemnego zaufania.

Wrogiem testera oprogramowania może też być nuda lub nudne zadania. Może to wynikać ze specyfiki pracy testerskiej oraz słabego zarządzania zadaniami. Konsekwencje nudy bywają krytyczne zarówno dla testerów, jak i dla organizacji. Może ona prowadzić na przykład do wypalenia zawodowego lub naturalnego zmęczenia powtarzalnymi zadaniami. Problem nudy jest mocno powiązany z testowaniem regresji czy też z manualnym testowaniem. Każda organizacja dbająca o motywowanie swoich pracowników powinna dostarczać narzędzia, które są w stanie nudne i powtarzalne czynności automatyzować albo pomóc w ich zredukowaniu. Możliwym rozwiązaniem jest też rotacja zadań wśród większej grupy pracowników.

Nuda jednak może być również wynikiem złego dopasowania zadania do naszych kompetencji. Istnieje ryzyko pojawienia się znudzenia u osób o przeciętnych kompetencjach i dostających łatwe zadania. Rozwiązaniem jest więc dostarczanie zadań dopasowanych do poziomu wiedzy, doświadczenia i kompetencji.

Spotkałem się z przypadkami wymuszania na testerach ręcznego uruchomienia każdego przypadku testowego na każdej nowej wersji oprogramowania. Jest to naturalna droga do zabicia kreatywności i doprowadzenia do zmęczenia pracą testera. Wykonywanie tych samych przypadków testowych bez odpowiedniego uzasadnienia ich uruchomienia może przypominać pracę Charliego Chaplina w filmie

„Dzisiejsze czasy”. Ciągłe przykręcanie jednej nakrętki na taśmie, przez którą przewijają się ich tysiące. Jeden popełniony błąd i cała ta misterna (lub szatańska) maszyneria przestaje poprawnie działać. Czy aby na pewno wykonanie każdego przypadku testowego jest złe? Uważam, że tak, ponieważ sygnalizuje poważne problemy organizacyjne. Przykładowo jesteśmy zmuszani do uruchomienia wszystkich przypadków, ponieważ kierownictwo nie wie, co znajduje się w poszczególnych wersjach oprogramowania. Wykonanie przypadków staje się więc dla niego potwierdzeniem, czy coś w nim jest, czy też nie. Aby tego uniknąć, należy już na poziomie planowania wersji definiować, co ma się znaleźć w oprogramowaniu, a później w notach wydania (*release notes*) opisywać, co rzeczywiście zostało do oprogramowania wprowadzone. Czasami jesteśmy zmuszani do uruchomienia wszystkich przypadków, ponieważ kierownictwo nie potrafi odpowiedzieć na pytanie, jakie są powiązania wewnątrz oprogramowania. Aby zapewnić, że drobna poprawka nie uszkodzi (nie wpłynie na działanie) jakiegoś istotnego obszaru aplikacji, testujemy wszystko. Tutaj kluczem do rozwiązania jest analiza wpływu realizowana przez wszystkich członków zespołu, od programisty przez architekta, aż po klienta, dzięki której oceniamy, jaki wpływ na poprawność działania będzie miała dana poprawka. Natomiast testerzy muszą opracować zestaw testów regresyjnych (różny od zestawu wszystkich przypadków testowych) uruchamianych na każdej nowej wersji. Zestaw ten warto zautomatyzować. W niektórych przypadkach uruchomienie wszystkich testów związane jest z brakiem zaufania do programistów. Z tym niestety niewiele da się zrobić.

Zmuszanie do wykonywania powtarzalnej czy nudnej pracy jest bardzo demotywujące. Każdy przełożony powinien być świadomy zgubnego wpływu takich nakazów na chęć pracy w danej firmie. Z nudą nie można walczyć na poziomie zwykłego testera. To zadanie menedżerskie i pojawienie się objawów zmęczenia i znużenia zadaniami jest zazwyczaj czytelnym potwierdzeniem nieudolnego zarządzania.

## 6.7. Kto może testować produkt?

Czasami w organizacji nie istnieje rola testera, a obowiązek testowania przerzuca się na innych uczestników procesu wytwarzania. Siła robocza do testowania może pochodzić z wewnątrz projektu lub możemy sięgać po osoby, które docelowo mają być użytkownikami systemu. Podkreślam, że testowanie ma wykonywać człowiek, a nie maszyna. Testowanie ciągle jest jeszcze domeną pracy ludzkich rąk. Hucznie ogłaszane wynalazki i samotestujące się oprogramowanie to tylko marketingowe chwytły i akademickie mrzonki, za którymi nie stoją konkretne rozwiązania. Nie wykluczam, że pewnego dnia powstanie oprogramowanie, które będzie bezbłędne lub rozwiąże problem w samym sobie. Dotychczas jednak nie wymyślono nic lepszego od człowieka, który uruchamiając oprogramowanie, próbuje udowodnić, że (nie) działa.

Oprogramowanie jest i jeszcze jakiś czas będzie testowane przez różne osoby. Trzy najważniejsze z nich prezentuje tab. 6.1.

**Tab. 6.1.** Typy testerów

Kto	Opis	Mocne strony	Słabe strony
Twórca	Osoba wytwarzająca oprogramowanie próbuje samodzielnie weryfikować jego poprawność. Takie podejście jest rekomendowane przez metodyki miękkie i traktowane jako część testów przez metodyki klasyczne	Wczesna próba sprawdzenia poprawności działania, umiejętność testowania elementów kodowych przed i po ich wytworzeniu (w tym technika TDD)	Problem braku dystansu do wytwarzanego przez siebie oprogramowania, dokonywanie tych samych błędnych założeń, brak zewnętrznego punktu kontrolnego, jeśli jest to jedyny poziom testowania
Jakaś osoba	Osoba bez przygotowania programistycznego lub testerskiego, delegowana do weryfikacji poprawności implementacji oprogramowania. Zazwyczaj użytkownik lub osoba o niewielkiej wiedzy z zakresu IT, pracująca z aplikacją na poziomie interfejsu	Brak szerokiej perspektywy pracownika IT; w przypadku testów prowadzonych przez użytkownika możliwość weryfikowania założeń wytwórców oprogramowania z rzeczywistym funkcjonowaniem oprogramowania	Brak wiedzy programistycznej i testerskiej, brak wyćwiczonych technik weryfikacji jakości oprogramowania; słabej jakości raporty defektów, np. „Nie działa funkcja X”
Tester			

	Osoba przeszkolona do pracy testerskiej polegającej na testowaniu oprogramowania. Specjalista w zawodzie	Umiejętność poszukiwania i odwoływania się do źródeł oczekiwanego rezultatu; umiejętność efektywnego i skutecznego testowania w zależności od założeń czasowych i budżetowych	W wielu przypadkach brak wiedzy programistycznej; po długiej pracy z jednym interfejsem niemożność dostrzegania nawet oczywistych defektów
--	--	---	--

### Co jaki czas tester powinien zmieniać projekty?

Zakłada się, że mniej więcej po sześciu miesiącach pracy z jednym interfejsem w ramach jednego projektu tester zaczyna akceptować awarie, które na początku projektu wydawały mu się warte raportowania. Ten efekt przyzwyczajenia się do niskiej jakości powinien być jednocześnie bodźcem do zmiany projektu.

Dlaczego uznaje się, że najgorszym testerem będzie programista? Powodów może być wiele. Głównie wynikają one ze specyficznego podejścia. Wielokrotnie spotykałem się ze zdaniem programisty, że testowanie ma charakter odtwórczy, natomiast programowanie jest tworzeniem. Skoro to praca łatwa i powtarzalna, to nie można jej traktować w kategoriach czegoś ciekawego czy nawet wartościowego. Nie można wykonywać profesjonalnie pracy, której się nie szanuje albo której sensu się nie widzi. Innym problemem z testowaniem programistycznym jest wąska specjalizacja i zazwyczaj niezwracanie uwagi na całość aplikacji. Krótkowzroczność przekłada się z kolei na marne pokrycie funkcji oprogramowania. I na koniec: ktoś, kto stworzył dane oprogramowanie, może podczas testowania go popełnić dokładnie te same błędy (wynikające z tych samych założeń), co przy jego wytwarzaniu.

Musimy również pamiętać, że odpowiedzialność za jakość nie może spoczywać wyłącznie na testerze. To zespół, czyli również programista, ma pilnować wysokiej jakości półproduktów powstających w procesie wytwórczym i dostarczenia końcowego produktu zgodnego z wytycznymi i założeniami. ISTQB promuje pogląd, że jakość testowania zależy w duży stopniu od (nie)zależności między testerem a programistą – im zależność głębsza, tym gorsze wykonanie testów, a staje się ono ekstremalnie złe, gdy zatrze się granica między testerem a programistą. Takie podejście nie jest pozbawione logiki, ale nie można

go traktować jako dogmat, zwłaszcza gdy uwzględnimy drugi biegun tego podejścia. Jest nim założenie, że prawdziwie profesjonalne testowanie pojawia się po pełnym oddzieleniu testerów i programistów. Lata doświadczeń projektowych pokazują, że ekstremalne rozwiązania niezmiernie rzadko są najlepsze. Zazwyczaj trzeba znaleźć punkt pośredni między testowaniem w pełni zależnym i w pełni niezależnym.

## 6.8. Umiejętności twarde testera

Tester oprogramowania musi mieć kompetencje techniczne lub wolę ich uzyskania. Zakres tych kompetencji może być bardzo ograniczony, ale nie wolno z nich zrezygnować.

Czy tester musi umieć kodować? Nie. Tester nie musi umieć pisać kodu, ale nie może się go bać. Wiele osób racjonalizuje swój brak umiejętności czytania kodu sformułowaniami typu: „Skoro nie wiem, jak to jest napisane, to jestem lepszym testerem”. Nie, nie jesteś. Powiedzmy sobie, że pisanie i czytanie kodu to nic złego i na pewno ci nie przeszkodzi być testerem doskonałym. Kod źródłowy pojawi się w wielu miejscach naszej pracy, poczynając od testów jednostkowych, przez automaty usprawniające nasze zadania, na pełnej automatyzacji kończąc. Tester potrafiący pisać kod ma większą szansę na rozwój w dowolnie wybranym obszarze testów automatycznych. Może się skoncentrować na testach niskopoziomowych, jest w stanie pisać automatyczne skrypty testujące graficzny interfejs użytkownika, może się zaangażować w testowanie wydajności, niezawodności czy utrzymania. Dodatkowo tester o kompetencjach programistycznych łatwiej włączy się w działania zespołów zwinnych, gdzie kompetencje muszą się przenikać. Musi być niczym Kobieta Pracująca z „Czterdziestolatka” – nie może się bać żadnej pracy.

Można również założyć, że na kodowaniu nie kończą się umiejętności twarde testera oprogramowania. Pracując z maszyną, musi ją rozumieć. Nie musi wiedzieć, jak przebiegają wszystkie procesy, ale musi rozumieć (bez sarkazmu i ironii), czym są i jak zostały skonstruowane: procesor, pamięć dyskowa i operacyjna, struktura plików i folderów, internet, protokół internetowy (i co dzieje się po wpisaniu adresu w przeglądarkę internetową), źródło strony itp. Powinien mieć także orientację w wielu technicznych aspektach



funkcjonowania sprzętu, sieci czy konstrukcji komunikacji. Podczas całego życia zawodowego musisz znać odpowiedzi na dziesiątki lub setki pytań typu:

- Jaka jest różnica między językiem interpretowanym i kompilowanym?
- Jakie są najważniejsze rozszerzenia dla plików i jakie jest ich przypisanie do aplikacji?
- Jak działa przeglądarka, jakie są jej funkcje podstawowe i jak możemy je rozbudować?

Wiele osób uzna te pytania za oczywistą oczywistość, niewartą uwagi. Z własnego jednak doświadczenia wiem, że jeśli ktoś zmieni branżę na przykład z historii sztuki lub księgowości na testowanie, nie będzie mu łatwo na nie odpowiedzieć. Tu pojawiają się dodatkowe punkty za techniczne wykształcenie. Tu również odcedzimy ziarno od plew, czyli testerów, którzy wiedzą, i tych, którzy nie wiedzą, ale się dowiedzą, od tych, którzy nie wiedzą i mają to w nosie. Cechą każdego testera, która pozwoli mu się utrzymać w zawodzie dłużej niż innym, będzie ciągły rozwój. Jeśli czegoś nie wie lub nie umie, musi znaleźć źródło informacji lub zyskać potrzebną umiejętność.

Dla osób, które mają wiedzę przypisaną do wąskiego zakresu aplikacji, np. wiedzę z ekonomii, potencjał rozwoju w testowaniu zamyka się zazwyczaj w tym obszarze. Jednocześnie tacy testerzy lepiej rozumieją biznesową stronę działania aplikacji. Umiejętność kodowania rzadko idzie w parze z rozumieniem biznesu, częściowo ze względu na zupełnie inny zestaw kompetencji i cech miękkich. Role te wymagają wzajemnego zrozumienia. Reasumując, ISTQB zaleca, aby każdy tester znał przynajmniej jeden język skryptowy, ale wymaga zestawu cech trudnych do połączenia z wiedzą techniczną. Na rynku testerskim jest miejsce dla osób o wiedzy zarówno technicznej, jak i nietechnicznej.

Do kompetencji testera będzie również należeć umiejętność dostosowania swojej pracy do metody prowadzenia projektu, ale o tym szczegółowo traktują publikacje dotyczące danej metody.

---

## Zadanie

Założenia: komputer, dostęp do internetu, 10 minut na przygotowanie się z wybranego tematu, najlepiej takiego, którego nie znasz, 3 minuty na omówienie zagadnienia.

Przykładowe aspekty do opracowania: jak działa i do czego służy aplikacja Xenu? Na czym polega problem komiwojażera? Opisz historię Cypru w XX wieku. Jaki jest status badań nad wirusem HIV?

---

## **6.9. Posługiwanie się narzędziami i automatyzacja**

Automatyzacja nie jest tematem prostym i nie wystarczy powiedzieć, że chce się ją robić, aby w magiczny sposób stało się to możliwe. Trzeba mieć dużo wiedzy, doświadczeń i kompetencji. Wiele firm rozpoczyna od kupna narzędzi do automatyzacji, nie mając pojęcia, jak ich używać i ile to realnie kosztuje. Największą pułapką są tu narzędzia open source, bo się ich nie kupuje, tylko ściąga bez płacenia. Problem polega na tym, że licencja nie jest w tym przypadku najwyższym kosztem wdrożenia. Po źle przygotowanej automatyzacji zostaje się z automatami stanowiącymi jedynie obciążenie dla uruchamiających je zespołów, a większość obowiązków wraca do testerów manualnych.

Nieudane wdrożenie testów automatycznych w organizacji doprowadza do sytuacji patowej. Z uwagi na nakłady kierownictwo nie chce się przyznać do błędu, więc ślepo brnie dalej, utrzymując automaty i tworząc nowe, a odpowiedzialność zrzuca na testerów manualnych, którzy równolegle muszą jeszcze testowaną aplikację „przeklikać”.

Automatyzacja jest osobnym projektem informatycznym, dla którego trzeba zdefiniować wymagania, napisać kod i to wszystko przetestować. Na końcu pojawi się koszt najczęściej zapominamy i nieuwzględniany – utrzymanie. Automatyczna aplikacja testująca musi nadążać za zmieniającymi się okolicznościami, czyli za aplikacją testowaną. Porównajmy to do innego typowego przypadku w ramach IT. Załóżmy, że zadanie polega na sprawdzaniu, czy aplikacja mobilna będzie działała w nowej wersji systemu operacyjnego. Testy takie są ważne, aby aplikacja wspierała klientów, którzy wybiorą sprzęt z najnowszym OS-em. Testy są istotne i zajmą dużo czasu i uwagi. A teraz pomyśl, że ten system operacyjny zmienia się każdego dnia, a aplikacja musi nadążyć za zmianą. Nie wystarczy obsługiwać co dziesiątą wersję, trzeba każdą. Nasza aplikacja z każdą ominiętą wersją stanie się coraz mniej stabilna, a po kilku wersjach trzeba będzie ją pisać

na nowo. Pamiętajmy również, że sponsorzy, zamawiając produkt, rzadko zamawiają dla niego automatyzację. Dla nich automat to tylko półprodukt do wytworzenia końcowego produktu software'owego, czyli realnie dodatkowy koszt.

W swojej pracy prędzej czy później zostaniesz albo zmuszony, albo zachęcony do automatyzowania. Dowiesz się, ile zarabiają eksperci od automatyzacji, i zapragniesz być jednym z nich albo twój menedżer uzna, że jedynym rozwiązaniem chaosu w organizacji będzie jego pogłębienie przez wdrożenie chaosu automatycznie generowanego. Wszyscy twoi koledzy już automatyzują, a tylko ty jesteś klikaczem? Wpędza cię to w depresję i dostrzegasz bezcelowość swojego życia. W tych okolicznościach musisz powiedzieć automatyzacji: „Przyjdź do mnie”. Mam do ciebie tylko jedną prośbę: nie płyn z prądem i skorzystaj z tych kilku rad.

**Cel.** Określ i zdefiniuj sobie cel, który chcesz osiągnąć. Bez tego się nie dowiesz, czy idziesz w dobrym kierunku. Twoim celem może być automatyzacja danego obszaru funkcjonalnego lub prosta próba pokazania, że testowanie manualne będzie mniej skuteczne.

**Wybierz narzędzie.** Ściągnięcie losowego narzędzia nie jest dobrym rozwiązaniem. Zanim pobierzesz i zainstalujesz aplikację na swoim komputerze, warto się o niej dowiedzieć czegoś więcej. Musisz znać jej wady i zalety, aby określić, czy twoje oczekiwania zostaną spełnione. Skorzystaj z internetu (ale tylko z zaufanych źródeł), on powie ci wiele na temat tego, co warto, a czego nie warto wybierać. Testowanie bez dokumentacji jest trudne. Kiedy zabierasz się za testowanie nowego oprogramowania, przede wszystkim sprawdzasz, czy nie ma do niego jakiejś specyfikacji. Podobnie jest z narzędziami: uczenie się ich jest łatwe, jeśli w sieci są odpowiednie podręczniki, tutoriale czy aktywne fora. Dzięki temu masz więcej szans na znalezienie odpowiedzi na pytania pojawiające się w trakcie używania narzędzia. Jeśli w sieci znajdujesz niedużo informacji o danej aplikacji, jest to pierwszy poważny powód, by jej nie używać.

**Pomyśl o zwrocie z inwestycji.** Czy zainwestowane w naukę narzędzia czas i środki się zwrócą? Czy znajomość tego narzędzia pomoże ci rozwinąć się w organizacji lub dostać nową albo lepszą pracę? Po co marnować czas na działania z góry skazane na niepowodzenie? Określ sobie zwrot inwestycji w naukę narzędzia i regularnie obserwuj, czy udało ci się go osiągnąć.

**Testuj aplikację do testowania.** Zaraz po instalacji warto zacząć testować aplikację. Jeśli masz zdefiniowany cel, to masz już i wymagania, jakie stawiasz aplikacji. Sprawdź, czy je spełnia. Uczenie się przez testowanie jest jedną z najbardziej efektywnych metod. Natomiast jeśli znajdziesz braki funkcjonalne i awarie, możesz szybko porzucić to rozwiązanie, zanim się okaże, że w tym wypadku nakłady nie mogą się zwrócić.

**Twórz dokumentację.** Rób notatki, spisuj defekty i dziel się wiedzą. Jeśli opiszesz online swoje problemy, to ktoś może ci pokazać, jak się z nimi uporać, jeśli poinformujesz o defektach, ktoś może zdecydować o ich usunięciu z aplikacji. Dorzucisz swoje trzy grosze do rozwoju narzędzia i może otrzymasz lepsze.

Najtrudniejsza będzie zawsze próba automatyzowania testów interfejsu graficznego aplikacji. Podlega on najbardziej dynamicznym zmianom i nawet drobne – z perspektywy sponsora czy architekta – modyfikacje mogą zniweczyć projekt testów automatycznych. Jeśli jednak chcesz brnąć ślepo w automatyzację GUI, korzystając ze skryptów generowanych na podstawie zachowania użytkowników, to uodpornij się na logiczne argumenty. To ci się przyda, kiedy będziesz zaliczać kolejne porażki w testach automatycznych i zaleje cię fala krytyki ze strony innych członków zespołu.

Nie wszystkie narzędzia w środowisku pracy to narzędzia do automatyzacji testów. Tester musi używać wielu narzędzi wszędzie tam, gdzie posługiwanie się nimi będzie ograniczać wysiłek i pracę manualną. Najlepszym lekarstwem na znużenie jest narzędzie, które nudną pracę wykona szybciej, dokładniej i efektywniej niż my. W tym przypadku rola testera ograniczy się do wynalezienia takiego narzędzia, jego adaptacji/wdrożenia, kontrolowania uruchomienia i na końcu podsumowania wyników pracy.

Istnieje też pula narzędzi, które stanowią bazę i które będziesz stosować bez względu na miejsce i czas pracy. Najpopularniejsze narzędzia testerów to:

- Program pocztowy jest centralnym miejscem na komunikację. Będziesz otrzymywać informacje i polecenia, a zwrotnie będziesz zadawać pytania i wysyłać raporty. Duża część komunikacji w narzędziu pocztowym nigdy nie powinna przez nie przechodzić. Wszystkie informacje przekazywane w relacji 1 do wielu lub 1 do 1 trzeba przeanalizować pod kątem realnych odbiorców. Jeśli wyślesz

raport z testów do kierownika, to będzie on dostępny dla dwóch osób w organizacji, a po utracie lub wymianie komputera, zwolnieniu pracowników czy awarii oprogramowania może zostać bezpowrotnie utracony. Organizacja nie będzie w stanie z niego skorzystać. Możliwie najwięcej informacji trzeba przesuwac do ogólnodostępnych narzędzi, wiki (otwartych systemów zarządzania wiedzą), dysków sieciowych itd. Program pocztowy to również problem zbyt wielu wiadomości, zalewu komunikatów i powiadomień. Pierwszą rzeczą, której musisz się nauczyć po opanowaniu testowania, jest zarządzanie tym potworem, który bezlitośnie będzie pożerał twój czas i odciągał twoją uwagę od testowania.

- Przeglądarka/internet – największy wróg i jednocześnie największy przyjaciel testera. Nawet jeśli nie testujesz aplikacji webowej, to i tak przeglądarką musisz się posługiwać. Dzięki globalnej sieci nie trzeba już tak dużo pamiętać, preparować danych (takich jak numery kont) czy przygotowywać specjalnych plików do testów (np. dużych JPG-ów o danej rozdzielczości). Wszystkie te rzeczy są dostępne w dwóch prostych krokach: Google.pl → Wyszukaj. Internet to dostępne online walidatory ciągów znaków i kodu, nieskończone źródło dowolnych danych testowych dostępnych na różnych stronach i w generatorach. Internet to kolejna hydra, która odciąga cię od pracy. Znasz to uczucie: wchodzisz do sieci tylko na chwilę poczytać newsy i po godzinie uświadamiasz sobie, że znasz już wydarzenia w Azji, Afryce i Ameryce Południowej, filmy z kotami, a twoi znajomi dzięki zaktualizowanemu statusowi na profilu społecznościowym wiedzą, że jesteś w pracy i pijesz kawę z nowego kubka (niektórzy to nawet lubią), natomiast w tym czasie twoja praca nie przesunęła się nawet o jeden defekt do przodu. Nie będziesz dobrym testerem, póki nie opanujesz do perfekcji umiejętności posługiwania się internetem i jednocześnie nie uodpornisz się na syreni śpiew śmieciotreści w rodzaju zabawnych obrazków i wiadomości z życia gwiazd.
- Edytor tekstu (zazwyczaj Microsoft Word) jest narzędziem doskonałym do wszystkiego, co wiąże się z tekstem. Może być użyty do przeglądania dokumentów i ich tworzenia. Tester używa go przede wszystkim do czytania specyfikacji produktu i tworzenia raportu z testów. Jest kilka funkcji, które przydają się w większych organizacjach, a jedna z podstawowych to funkcja recenzji.

- Arkusz kalkulacyjny (zazwyczaj Microsoft Excel), jak sama nazwa wskazuje, w firmie służy jako baza danych. Tabelaryczny widok dostępny na całą stronę z możliwością sortowania kusi testerów oprogramowania do umieszczania tam przypadków testowych, a opcja generowania wykresów z danych przyciąga kierowników. Jest to narzędzie, które zniechęciło mnie do pracy w korporacji i dzięki któremu postanowiłem szukać własnej drogi. Ale jest to narzędzie doskonałe do wszystkiego łącznie z analizami danych, projektowania interfejsów czy generowania danych. Tester oprogramowania powinien uzyskać ekspercki poziom wiedzy o arkuszu kalkulacyjnym i będzie to umiejętność, która przyda się przez całą karierę.
- Narzędzie do raportowania o defektach jest profesjonalnym rozwiązaniem zastępującym raportowanie pocztą elektroniczną czy też za pomocą dokumentów. Jest to pewien typ narzędzia, który ma następujące podstawowe funkcje: autoryzowanie użytkowników w systemie i nadawanie im odpowiedniego poziomu uprawnień, bazę danych zgłoszeń oraz wbudowany proces zarządzania zgłoszeniami. Jest konieczny w większości projektów informatycznych, zwłaszcza realizowanych przez większe zespoły, gdzie obieg informacji i dostęp do nich stanowi istotny czynnik. Dobra wiadomość dla wszystkich adeptów testowania: poznanie jednej aplikacji tego typu otwiera bramy do używania wszystkich innych, ponieważ ich konstrukcja jest bardzo zbliżona. Jeśli znasz Mantis lub Bugzillę (bardzo rozpowszechnione narzędzia o otwartym kodzie), to właściwie znasz wszystkie inne narzędzia, w tym również komercyjne, np. Jira czy HP ALM.

Użycie narzędzi to dla współczesnego testera warunek absolutnie konieczny. Posiadanie tej wiedzy to przepustka do kolejnego etapu rozwoju, a w związku z tym do innych zadań i lepszych zarobków.

## 6.10. Współpraca tester-programista

Pojedynczy tester nie może być odpowiedzialny za jakość oprogramowania bez względu na to, czy jest członkiem zespołu testerskiego, czy jedynym testerem w organizacji i projekcie. Za jakość

zawsze odpowiada zespół ludzi. Tester może jedynie zadbać o zmierzenie jakości produktu. Bez względu na model wytwarzania oprogramowania programiści muszą się zaangażować w testowanie, korzystając z technik analizy statycznej lub dbać o wysokiej jakości kod, stosując zasadę redukcji ostrzeżeń podczas kompilacji (*zero warning policy*). Przy tym tester, wcześniej angażując się w projekt, może wspierać programistów w czynnościach odwołujących się do jakości kodu, takich jak: analiza statyczna, wytwarzanie oprogramowania sterowane testami, testowanie jednostkowe czy programowanie w parach. Testerzy i programiści mogą współpracować przy testowaniu automatycznym.

Podstawowe znaczenie w relacjach między tymi dwiema rolami ma współpraca. Testerzy i programiści tworzą zespół ludzi o różnym zakresie odpowiedzialności i czynności, ale wspólnym celu. Grupa realizująca produkt i grupa poddająca go krytyce musi znaleźć wspólny język, który nie będzie antagonizował, a pomagał osiągać zamierzone cele. Organizacje, które dostrzegają konieczność zbudowania dobrych relacji między członkami zespołu, osiągną sukces.

## 6.11. Rozwój testera w organizacji

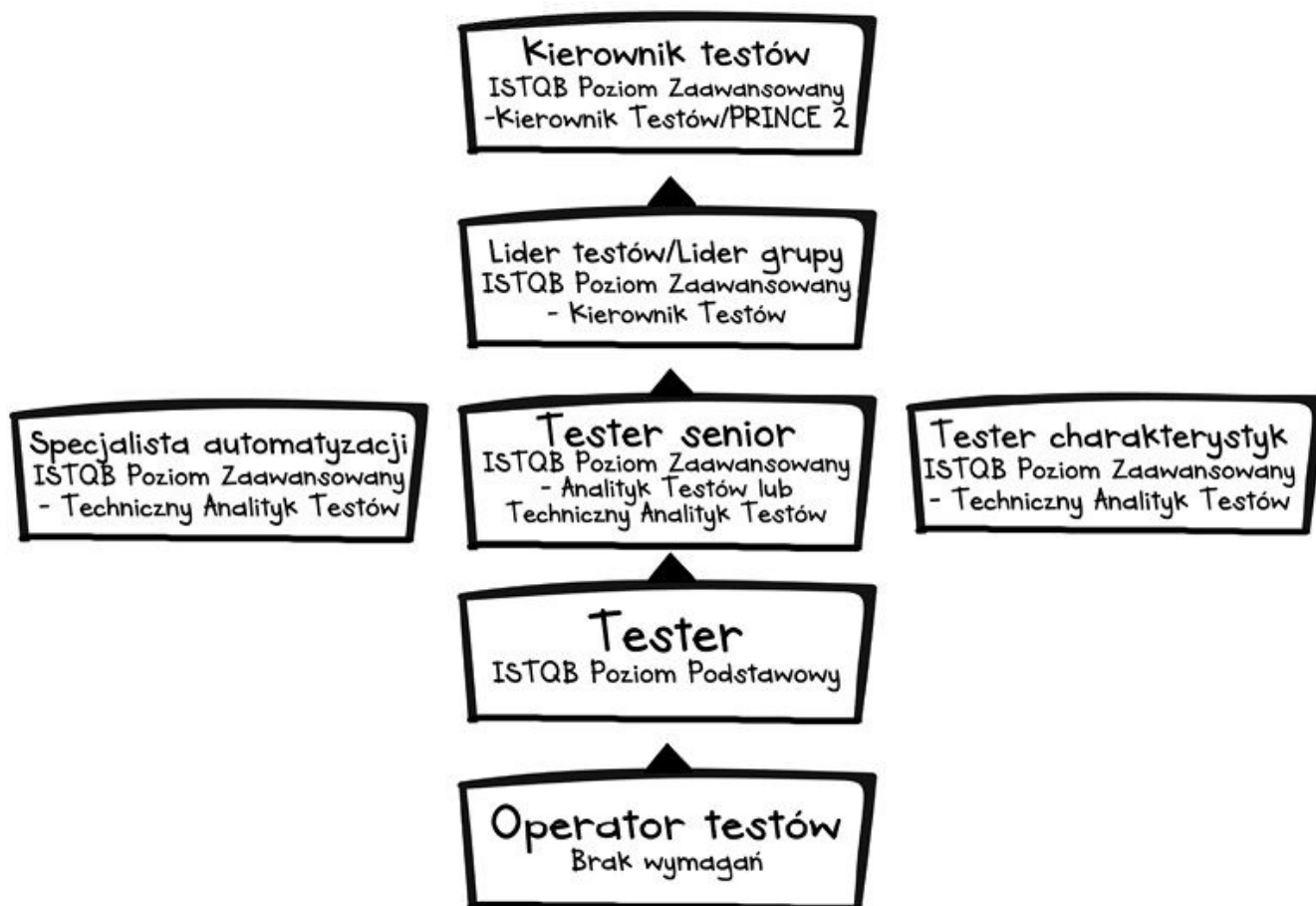
Testowanie może być zawodem na całe życie. Młody, niedoświadczony i zazwyczaj niewykształcony tester zacznie od stanowiska technika testów lub tzw. operatora. Człowiek po skończonych studiach może być inżynierem testów lub specjalistą ds. testów – tu pojawia się precyzyjniejsze określenie typu: ds. testów akceptacyjnych, integracyjnych, oprogramowania itp. Metodyka RUP (*rational unified process*) poszła jeszcze dalej, odróżniając testerów od analityków i projektantów testów. Oczywiście w ciągu lat można odgrywać różne role – zmieniając się z czasem i nabywaniem doświadczenia – od testera oprogramowania, przez seniora testów, po lidera i kierownika testów. Nie są to jednak jedyne możliwości rozwoju. Kiedy zaczynałem swoją pracę jako tester, nie miałem pewności, co będę mógł osiągnąć przez lata kariery. Z perspektywy czasu widzę, jak się ona zmieniała: programista (niecały rok), tester (dwa lata), koordynator testów (2 lata), kierownik grupy testerów (3 lata), konsultant i trener testowania (ponad 6 lat). W mojej karierze nastąpiły dość poważne skróty związane

z brakiem w tym czasie pewnych kompetencji na rynku i ścieżka, którą wybrałem, nie była w pełni optymalna. Wiem, że mogłaby wyglądać zupełnie inaczej, gdyby ktoś pokazał mi wtedy możliwości.

Na rys. 6.2. zaprezentowano przykładowy schemat rozwoju pracowników w testowaniu oprogramowania z mapowaniem na certyfikaty ISTQB. Jest to jedno z możliwych rozwiązań dla osób zainteresowanych testowaniem.

Operatorem testów jest każda osoba wykonująca przypadki testowe napisane przez bardziej doświadczonych testerów. To funkcja wymagająca niezbyt wielu umiejętności, więc może ją pełnić średnio rozgarnięty absolwent szkoły średniej bez matury. Bycie testerem to już zobowiązanie. Jest to osoba pisząca i wykonująca przypadki testowe. Kolejny etap rozwoju to specjalista od automatyzacji, osoba głównie tworząca automatyczne skrypty testowe. Na tym samym poziomie znajduje się tester senior, czyli doświadczony tester oprogramowania, który zajmuje się trudnymi obszarami oraz tworzy specyfikację testów, a także tester charakterystyk, zajmujący się niefunkcjonalnymi charakterystykami oprogramowania. Na kolejnym szczeblu jest lider testów małej grupy testerskiej lub też lider obszaru testowego, a potem kierownik zespołu testerów.



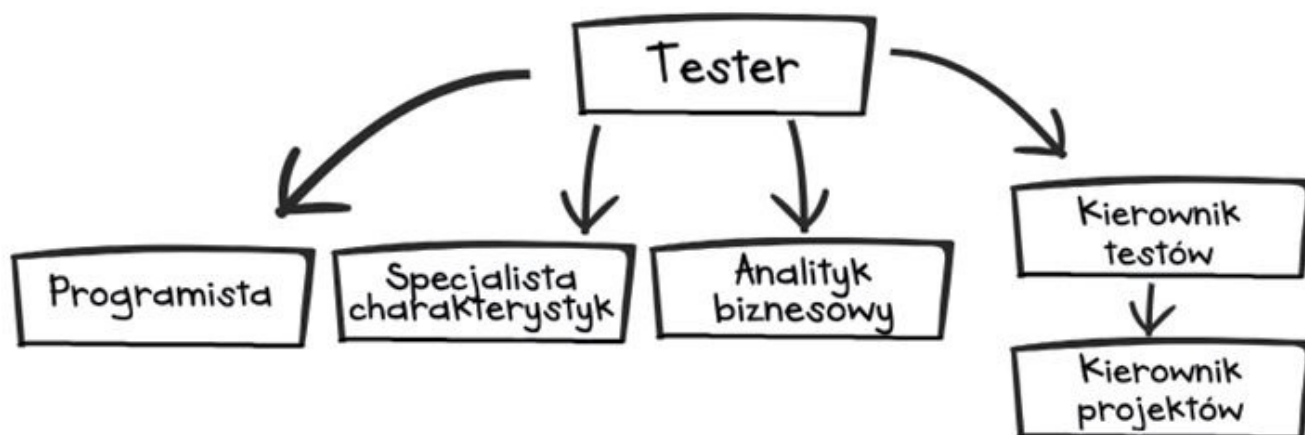


**Rys. 6.2.** Schemat potencjalnego rozwoju testera z mapowaniem na certyfikaty

Opierając się na własnych doświadczeniach, narysowałem trochę inną strukturę (rys. 6.3), która moim zdaniem bardziej przystaje do rzeczywistości i oddaje stosunki na rynku polskim.

Rozpoczęcie od roli testera otwiera kilka dróg. Oczywiście można przejść do programowania i wiele osób tak robi. „Zatrudnię się jako tester, nauczę się co nieco, a potem przejdę na programistę i zrobię coś ciekawego i pożytecznego”. Bardzo nie lubię takiego podejścia i traktuję takich ludzi jak pasażerów na gapę w testowaniu. Marnują czas swój i innych, bo skoro nie zagrzeją miejsca w testowaniu, to nie warto w nich inwestować. Zazwyczaj kończą jako słabi testerzy albo słabi programiści. Są oczywiście wyjątki, czyli ci, którzy od początku interesowali się testowaniem, ale w którymś momencie uznali, że kodowanie jest jednak ciekawsze. Inna opcja to specjalista od testów

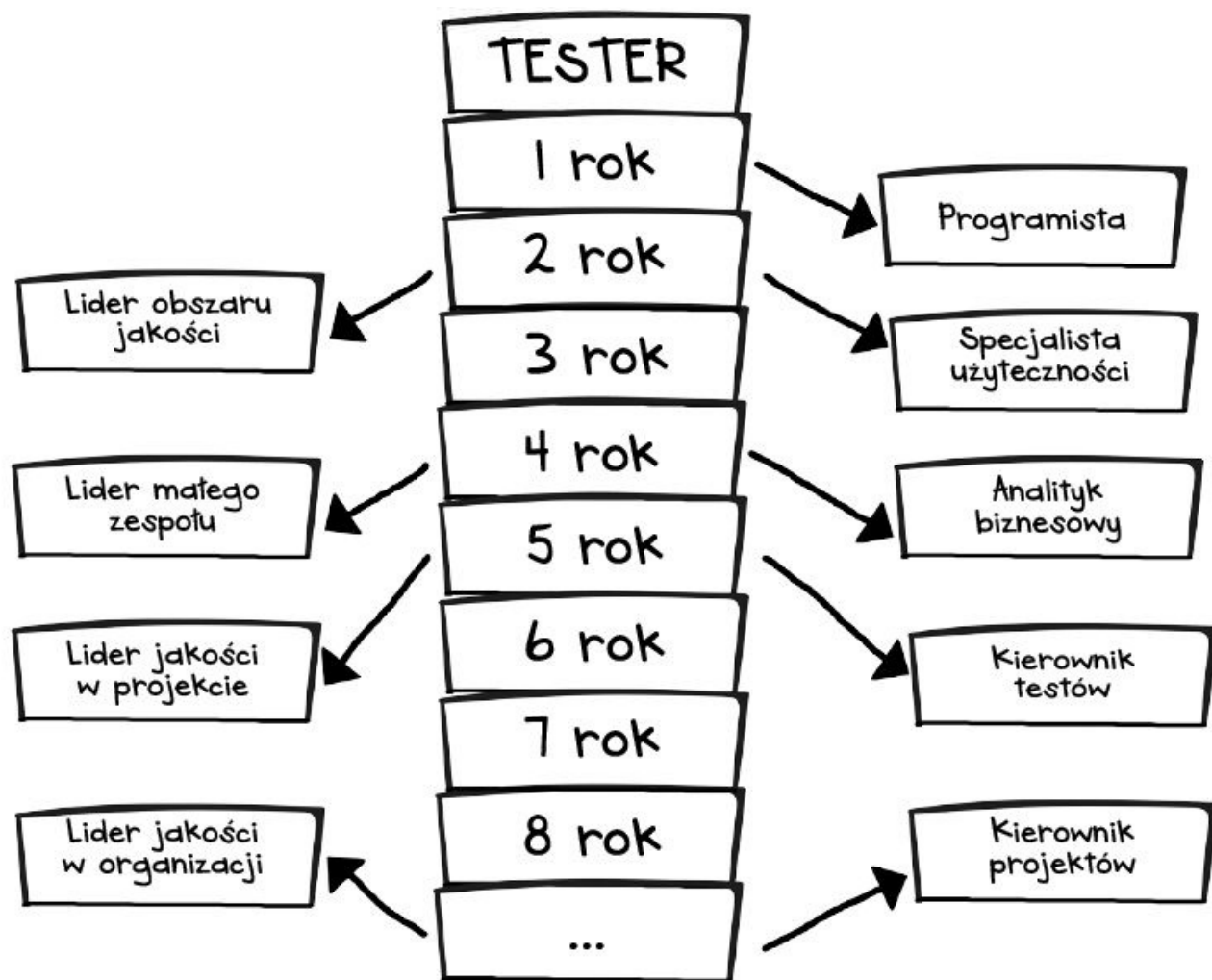
niefunkcjonalnych, czyli tester charakterystyk oprogramowania. To droga dość zróżnicowana (bo znajdziemy tam wszystkie charakterystyki opisywane w ISO 9126, od użyteczności po wydajność), zazwyczaj nie tylko ciekawa, ale również bardzo intratna finansowo. Czasami można również odejść od głównego nurtu w inne obszary, np. analizy biznesowej. Analiza jest bardzo naturalną ścieżką rozwoju i praktycznie mocno się pokrywa z testowaniem w zakresie kompetencji. Na koniec zostaje ścieżka menedżerska. Oczywiście nie dla każdego. Jest to praca bardzo trudna, bo wiążąca się z dużo większą odpowiedzialnością: nie tylko za siebie, ale i za innych. W rolach menedżerskich nigdy nie czułem się do końca komfortowo, choć bardzo to lubię. Miałem wrażenie ciągłego chodzenia po kruchej tafli zamrożonego jeziora i wiedziałem, że nawet najmniejszy błąd może mnie pogrążyć. Nie polecam tej ścieżki ludziom, którzy cenią sobie spokój i odpoczynek po 8-godzinnej pracy.



**Rys. 6.3.** Rozwój testera na różnych ścieżkach

Kolejna wizualizacja (rys. 6.4) to sugerowane przeze mnie perspektywy czasowe. Można śmiało powiedzieć, że dla wielu osób będą one wyśrubowane i zbyt ambitne, ale bardzo dużo zależy od wewnętrznej i zewnętrznej motywacji do realizowania się w danym obszarze i od cech osobowości. Niektórzy uważają, że liderem nie można zostać po dwóch latach pracy. Ja się z tym nie zgadzam. Uważam, że dwa lata doświadczenia to za mało na zarządzanie ludźmi, ale wystarczająco na wzięcie odpowiedzialności za dany obszar funkcjonalności.

Każda organizacja powinna mieć własny schemat rozwoju pracowników. Z jednej strony pracodawca może efektywniej zarządzać rozwojem pracowników, z drugiej strony tester ma świadomość ścieżek swojego rozwoju w danej firmie. Taka ścieżka to bardzo dobre narzędzie do planowania swojej przyszłości.



Rys. 6.4. Rozwój testera w zależności od doświadczenia

## 6.12. Czego oczekuje się od testera na rynku pracy

Są dwie podstawowe cechy osobowości, które powinny wyróżniać testerów: rzetelność i dociekliwość. Można powiedzieć, że takie zalety

mogą przebić nawet techniczne przygotowanie, jeśli łączą się również z ciekawością poznania.

Ogólne obowiązki to:

- realizowanie testów systemów zgodnie z dokumentacją,
- przygotowanie własnych scenariuszy testowych,
- tworzenie skryptów testowych,
- wykonywanie testów i opracowywanie raportów,
- współpraca z innymi działami testowymi firmy,
- współpraca z działem programistycznym.

Zakres działań testerów jest ogólnie znany w branży i nie wymaga szerokiego komentarza, jednak w zależności od stosowanej metodyki lub po prostu specyfiki firmy zadania stawiane przed testerami mogą się różnić.

Jakie umiejętności i przygotowanie powinien mieć tester? Wymagania nie są wysokie, co nie znaczy, że być testerem jest tak łatwo, jak być na przykład ministrem. Wymaga to lat pracy i chęci samokształcenia, gdyż polskie uczelnie nie uczą testować. Tester powinien mieć:

- wiedzę z zakresu testowania oprogramowania,
- doświadczenie w testowaniu oprogramowania,
- wykształcenie wyższe (informatyczne, techniczne lub ekonomiczne) – preferowane,
- umiejętność pracy w zespole.

Jednakże dobrze są widziane dodatkowe umiejętności:

- pracowania mimo braku pełnej informacji,
- szybkiego uczenia się,
- dobrego komunikowania się słownie i pisemnie,
- ustalania priorytetów,
- samoorganizowania się.

Standardy są mocno ogólne, kiedy opisują wymagania, rolę czy obowiązki testera. W niektórych metodykach, np. RUP, opis staje się bardziej szczegółowy. W tej iteracyjnej metodzie tworzenia oprogramowania testerowi stawia się następujące wymagania:

- implementacja i wykonanie testów,
- zapisywanie wyników testów,
- analiza wykrytych błędów,
- dokumentowanie przypadków (rozumianych jako zachowanie sprzeczne z doświadczeniem, ale nie z dokumentacją).

RUP jest jednak metodą umiierającą i nie znajdzie się jej w zbyt wielu organizacjach.

## 6.13. Zawód z przyszłością

Testerzy oprogramowania są bardzo poszukiwani, a zawód ma przed sobą świetlaną przyszłość. Skoro czytasz tę książkę, to nie trzeba cię przekonywać, ale warto podać ci kilka dobrych powodów, którymi zachęcisz innych, by wstąpili do zakonu testerskiego.

1. Testerów oprogramowania brakuje na rynku – jest to ciągle zawód, który się rozwija i kolejne firmy oraz organizacje uświadamiają sobie, że muszą zatrudnić testerów, jednocześnie próbując odpowiedzieć sobie na pytanie, jak latami funkcjonowali bez nich.
2. Tester oprogramowania to zawód relatywnie prosty dla osób, które mają odpowiedni zestaw cech, a niekoniecznie przygotowanie techniczne. Mogą wejść do niego ludzie, którzy nigdy nie widzieli się w IT, ale w swoich zawodach, np. ekonomisty czy nauczyciela, nie mogą znaleźć pracy.
3. Testowaniem trudni się wiele osób przypadkowych, których będzie można szybko się z zawodu pozbyć i zrobić miejsce dla profesjonalistów. Kolejny rok trwa gorączkowe wypełnianie luki, jaką jest niedostatek testerów oprogramowania. W rezultacie do zawodu przyjmuje się ludzi, którzy nigdy nie powinni się w nim pracować. Sam zatrudniłem kilku takich i do dzisiaj tego żałuję. Kiedy tylko sytuacja się ustabilizuje, ludzie ci naturalnie z zawodu zostaną wyrugowaniu (miejmy nadzieję). Oto kilka przykładów testerów z przypadku:
  - Biznesmen – jest człowiekiem pieniądza, a nie testerem i jeśli jutro za bycie kasjerem w supermarkecie zaproponują mu 10 zł więcej, to szybko tam przejdzie.

- Dewastator – w CV pisze: „Lubię ludzi”, a jednocześnie obgady cię za plecami, będzie spiskować i knuć, dewastując projekty i organizacje.
- Nieudacznik – narzeka: „Nic mi w życiu nie wychodzi, jestem pantoflarzem, nie mam własnego zdania, może wyjdzie mi w testowaniu”. To najbardziej przypadkowy pracownik, bo niepowodzenia życiowe kompensuje sobie kosztem innych osób w projekcie.

Najgorsze, że są to zwykli judasze, którzy dziś mówią „Kocham testowanie”, a jak wiatr powieje w innym kierunku, powiedzą „Testowanie to zawód dla frajerów, każdy to potrafi”.

Przyszłość przed testowaniem jest świetlista. Do czasu wymyślenia samotestującego się oprogramowania będziemy potrzebni. Aktualnie obserwujemy, jak kolejne zawody umierają śmiercią naturalną, bo ludzi zastępują maszyny, roboty czy oprogramowanie. A kto to oprogramowanie tworzy? Programista. Kto to oprogramowanie testuje? Tester. Wydaje się, że wszystkich zawodów, które zostaną wyeliminowane w przyszłości, programista maszyn znajdzie się na ostatnim, a tester tychże maszyn na przedostatnim miejscu.

## 6.14. Zarobki testerów

Kiedy postanowisz pójść na rozmowę kwalifikacyjną lub do szefa po podwyżkę, na pewno przyjdzie ci odpowiedzieć na fundamentalne pytanie o twoje oczekiwania finansowe. Ile zaproponować, żeby nie było to ani za dużo, ani za mało?

Ile jest warta nasza praca? Tyle, ile ktoś chce za nią zapłacić. Pytanie brzmi, ile pracodawcy chcą płacić za naszą pracę. W Wielkiej Brytanii jest standardem, że ofertom pracy towarzyszy kwota zarobków oferowana na danym stanowisku. Podobnie staje się w Polsce. Założona przeze mnie grupa „Testowanie Oprogramowania” na Facebooku do swojego regulaminu wpisała możliwość publikowania ogłoszeń o pracy jedynie z widelkami zarobków. Kształtują się one różnie. Są oferty za 2000 zł brutto i takie za kilkadziesiąt tysięcy. Dzięki tej otwartości, uczestnicząc w procesie rekrutacyjnym, dowiemy się, ile pracodawca chce zapłacić za naszą usługę.

Stawki dla testerów zależą od wielu czynników, oto kilka z nich:

- Miasto – czy jest postrzegane jako zagłębie IT, czy też rynek dopiero się rozwijający. Pamiętaj, że w dużych miastach, takich jak Warszawa czy Wrocław, jest permanentny deficyt specjalistów IT i firmy mocno podbijają stawki, aby znaleźć ludzi do pracy.
- Koszty życia w danym mieście – ośrodki atrakcyjne biznesowo i turystycznie zawsze są droższe, więc pracodawcy muszą zapłacić tyle, by pensja starczała na więcej niż tylko na koszty życia.
- Czy pracodawcą jest firma polska czy korporacja zagraniczna – zagraniczne firmy potrafią odnosić zarobki do swoich oddziałów w innych lokalizacjach i choć nie zapłacą tyle, ile dostają Amerykanie czy Niemcy, to mimo wszystko więcej niż pracodawca polski.
- Jakie są oczekiwania na danym stanowisku – znajomość języków, automatyzacja, doświadczenie, certyfikaty. Oczywiście im wyższe wymagania, tym trudniej znaleźć pracownika i tym wyższe stawki oferuje się za pracę.

Ponieważ aktualnie na rynku brakuje testerów oprogramowania, firmy są dużo bardziej skłonne rozmawiać o pieniądzach. Z jednej strony, rekrutujący potrafią zadzwonić i powiedzieć: „Chciałbym panu zaproponować stanowisko z obecną pensją plus 20 procent”. Z drugiej strony, pracownicy mający już pracę lub pewność jej uzyskania definiują sobie na wstępnych etapach rekrutacji poziom pensji: „Chętnie się z państwem spotkam, jeżeli proponowana stawka jest nie niższa niż X zł netto”. Rynek pracy jest teraz bardziej rynkiem pracownika niż pracodawcy. Oznacza to, że pracownik może przebierać w ofertach, a pracodawcy muszą licytować się o pracowników. Warto z tego skorzystać, bo taka sytuacja nie będzie trwała w nieskończoność.

Wiele portali prowadzi badania dotyczące zarobków testerów. Dane na temat zarobków możesz znaleźć w sieci i odnieść się do średnich stawek, kiedy definiujesz swoje oczekiwania finansowe względem pracodawcy. Nie patrz na dane wybiórczo. Mapuj swoje obecne stanowisko, doświadczenie na informacje z zestawienia. Nie kieruj się średnim wynagrodzeniem za testowanie, bo może być zawyżone przez umowy firma–firma czy przez stanowiska menedżerskie.

Jeśli tylko jesteś w stanie oderwać się od swojego miasta i swojej roli, to warto obserwować trendy i we właściwym momencie rozważyć

zmianę miejsca zamieszkania w poszukiwaniu bardziej atrakcyjnego wynagrodzenia. Analitycy rynku pracy mówią, że jedna z największych naszych wad jako pracowników polega na tym, że nie umiemy się adaptować do zmian, co różni nas np. od Amerykanów, którzy potrafią podróżować po kraju w poszukiwaniu pracy. Najlepsza i najtrudniejsza decyzja, którą przyszło mi podjąć, dotyczyła wyjazdu na tzw. saksy w pierwszych latach moich kariery. Dwa lata pracy w Niemczech pozwoliły mi wzmocnić swoją pozycję przetargową po powrocie. Nie zachęcam cię do emigracji, bo dziś rynek wygląda inaczej niż w pierwszych latach XXI wieku. Dziś poszukiwanie pracy oznacza raczej zmianę miasta niż kraju.

Jak pokazują analizy, stawki też będą rosły wraz z doświadczeniem. Im dłużej pracujesz w zawodzie, tym większych pieniędzy możesz oczekiwać. Ten trend w którymś momencie się zatrzyma, jeśli nie będziesz awansować wraz z upływem lat na wyższe stanowiska. Twoją wartość na rynku definiują również certyfikaty. Papier w Polsce ma ciągle swoją wartość, więc warto z tego skorzystać.

Możesz też analizować formy zatrudnienia ze swoim pracodawcą. Ze względu na podatki i ubezpieczenia społeczne w Polsce coraz częściej preferowaną formą współpracy będzie umowa firma–firma. Dlaczego? Dlatego, że wiążą się z nią reguły dużo czytelniejsze dla potencjalnego pracodawcy (czy raczej zlecającego). W przypadku umowy o pracę nie wie on, ile otrzyma za miesięczną pensję. Wystarczy, aby pracownik się pochorował czy wziął urlop (np. na żądanie), i pracodawca ponosi koszty, nie dostając niczego w zamian. Osobę zatrudnioną na umowę o pracę jest również dużo trudniej zwolnić, gdy przyjdą cięższe czasy dla firmy. Pracodawcy naprawdę potrafią docenić elastyczność umowy biznes–biznes, proponując niejednokrotnie kwoty kilkakrotnie wyższe od pensji netto.

Jeśli jesteś testerem, twoja pensja może oscylować bliżej średniej niż najniższej krajowej, i to nawet jeśli dopiero zaczynasz przygodę z testowaniem. Stawka będąca dwukrotnością najniższej krajowej na początek i trzykrotnością po kilkumiesięcznym udowodnieniu swoich umiejętności nie powinna stanowić problemu w polskich centrach IT, takich jak Wrocław czy Warszawa.

Stawki dla testerów są ciągle niższe od programistycznych i to się zapewne nie zmieni. Różnice jednak nie są tak olbrzymie jak w latach dziewięćdziesiątych. Stawki różnią się w niewielkim zakresie i pokazują, że pracodawcy coraz częściej zrównują te dwa zawody.



## 6.15. Praca testera w innych publikacjach

Jeśli masz złudną nadzieję, że dzięki mojej książce zostaniesz testerem, to ją porzuć. Jedna książka nie uczyni cię lepszą osobą, nie zmieni całego twojego życia. W każdym temacie i każdym aspekcie trzeba poszukiwać. Testerzy są dobrzy w poszukiwaniach, więc nie mogą poprzestać na znajdowaniu awarii w oprogramowaniu, muszą również znaleźć własną odpowiedź na pytanie, czym jest testowanie. Do mojej definicji testowania i światopoglądu testerskiego doprowadziło mnie studiowanie książek, słuchanie prezentacji, czytanie blogów i rozmawianie z testerami. Część tych źródeł przemodelowało moje postrzeganie testowania, inne pomogły mi uporządkować chaotyczny zbiór zagadnień. Dalej wskazuję kilka najważniejszych i takich, które – choć niespójne ze sobą – pozwoliły mi skorelować pracę z teorią.

Wśród wielu publikacji opisujących pracę testera wyróżnia się jedna, autorstwa Jamesa Bacha. Napisał on testerskie zobowiązanie w formie dialogu z niewidzialnym programistą. Jest to kwintesencja naszej pracy, więc nie mogło jej zabraknąć w tej książce:

*Drogi Programisto,*

*Moja praca polega na tym, by pomóc ci w twojej pracy. Moja praca polega na wspieraniu cię, kiedy tworzysz jakość, na odciążaniu zamiast obciążania. W tym duchu podejmuję następujące zobowiązanie:*

- 1. Świadczę usługi. Jesteś moim bardzo ważnym klientem. Nie będę usatysfakcjonowany, dopóki ty nie poczujesz satysfakcji.*
- 2. Nie jestem strażnikiem jakości. Nie mam wyłączności na jakość. Dostarczenie dobrego produktu jest naszym wspólnym celem.*
- 3. Będę testował twój kod tak szybko, jak tylko potrafię, zaraz po dostawie. Wiem, że potrzebujesz moich rezultatów szybko (szczególnie poprawek defektów i nowych funkcji).*
- 4. Będę dążyć do testowania w taki sposób, aby wspomóc twoją produktywność. Nie będę wąskim gardłem.*
- 5. Dokonam każdego sensownego działania testowego, nawet jeśli będę miał jedynie częściową informację o produkcie.*
- 6. Poznam produkt szybko i użyję tej wiedzy do sprytniejszego testowania.*
- 7. Będę testował najpierw najważniejsze rzeczy i postaram się znaleźć najważniejsze problemy na początku. Będę również raportował o defektach, które możesz uznać za nieważne – tylko na wszelki wypadek, gdyby okazały się jednak istotne – ale spędzę nad nimi mniej czasu.*

8. *Będę dążył do testowania w interesie każdego, kogo opinia ma znaczenie, włączając w to ciebie, byś mógł podejmować lepsze decyzje o produkcie.*
9. *Będę pisał jasne, spójne, przemyślane i pełne szacunku raporty o problemach. Mogę wysunąć sugestie na temat projektu, ale nigdy nie ośmielę się być projektantem.*
10. *Dam ci znać, jak testuję i poproszę o komentarz. Będę się z tobą naradzał w kwestii małych rzeczy, które możesz zrobić, by produkt był łatwiejszy do testowania.*
11. *Zachęcam cię, byś do mnie przychodził ze specjalnymi prośbami, na przykład żebym coś dla ciebie sprawdził, pomógł ci coś udokumentować czy uruchomić specjalny test.*
12. *Nie będę marnował twojego czasu, a jeśli to zrobię, to będę się uczył na tym błędzie.*

*Z poważaniem*

*Tester*

Jednakże należy się przygotować na wiele trudów i wyrzeczeń. W pracy testera, nie ukrywajmy, zdarzają się gorsze dni i frustracje. Czasami musimy się mierzyć z rzeczami, które wydają się na pierwszy rzut oka nie do przeskoczenia i nie do obejścia. Aby radzić sobie ze stresem i nie denerwować się tym, na co nie mamy wpływu, trzeba zaakceptować stan rzeczy. Colin Cherry w swoim blogu opisał to najlepiej – w formie mantry (radzę ten tekst wydrukować i przykleić na biurko, a w stresowych sytuacjach czytać, oddychać spokojnie i uśmiechać się).

1. *Akceptuję, że wymagania nigdy nie będą doskonałe.*
2. *Akceptuję, że programiści mają swoje ograniczenia czasowe, tak jak wszyscy członkowie zespołu.*
3. *Akceptuję, że środowisko testowe nie będzie gotowe na czas.*
4. *Akceptuję, że terminy będą się zmieniać.*
5. *Akceptuję, że użytkownicy będą zmieniać zdanie.*
6. *Akceptuję, że będzie więcej defektów w nowym wydaniu, niż przewidywano.*
7. *Akceptuję, że eksperci z danego obszaru okażą się niedostępni, kiedy będzie konieczna konsultacja.*
8. *Akceptuję, że spotkania dotyczące usuwania defektów będą trwały dłużej, niż zakładano.*
9. *Akceptuję, że kierownik projektu nie zrozumie, dlaczego testowanie trwa tak długo.*
10. *Akceptuję, że nie pojawią się poprawki, których się dziś spodziewano.*
11. *Akceptuję, że żaden z użytkowników nie będzie wiedział, jak określić kryterium akceptacji.*
12. *Akceptuję, że najnowsza wersja oprogramowania pojawi się o 16:00 zamiast o 8:00 rano.*
13. *Akceptuję, że będę musiał pominąć niektóre przypadki testowe, aby zmieścić się w harmonogramie.*

14. Akceptuję, że niektóre z moich raportów defektów będą odrzucone jako „funkcja” lub „niemożliwe do zreprodukowania”.
15. Akceptuję, że testy wydajności będą opóźnione ze względu na niestabilność kodu.
16. Akceptuję, że ktoś spoza zespołu testowania będzie szacował czas na testy.
17. Akceptuję, że biuro kierownika projektu będzie chciało wiedzieć, ile przypadków zostanie dziś wykonanych.
18. Akceptuję koncept z „Dnia świstaka” i to, że już czwarty raz w tym tygodniu mamy pierwszy dzień testów!
19. Akceptuję, że niezależnie od wielkości planowania i przygotowań pierwszy test zakończy się niepowodzeniem.
20. Akceptuję, że moje dane testowe będą dobre dla wcześniejszej wersji.
21. Akceptuję, że nie starczy czasu na sprawdzenie wszystkich wyników, zanim rozpocznie się kolejny dzień pracy.
22. Akceptuję, że najważniejszego defektu nie uda mi się zreprodukować podczas spotkania z kierownikiem programistów.
23. Akceptuję, że jestem oceniany na podstawie liczby tych defektów, które znalazłem, a nie tych, którym zapobiegłem.
24. Akceptuję, że zdalny zespół, ulokowany za granicą, nie zrozumie najprostszego polecenia.
25. Akceptuję, że popełniam błędy i nie będę z tego powodu dla siebie surowy.

Nawet jeśli dziś nie wszystkie stwierdzenia w zobowiązaniu w mantrze są dla ciebie jasne, to zanim minie pierwszy rok twojej testerskiej pracy, zrozumiesz ich sens i wagę.

Kolejną książką wartą wspomnienia jest „How to break software” Jamesa Whittakera, poradnik testera oprogramowania. Już sam tytuł sugeruje, że testerzy mogą uszkodzić oprogramowanie. Pytanie, jakie można sobie zadać, brzmi: czy znajdując coś, co jest zepsute, rzeczywiście to psujemy? Jest to inne podejście do testowania: odkrywanie niskiej jakości oprogramowania jest postrzegane jako rzeczywiste uszkodzenie czegoś. Pamiętasz może taką sytuację z dzieciństwa: znajdujesz coś zepsutego i zanosisz rodzicom ze słowami: „To nie ja! To już było zepsute!”? Dziś w pracy testera mamy dokładnie taką samą sytuację. Testerów oskarża się o psucie, podczas gdy mówimy jedynie o tym, że coś już było zepsute. Nie oskarżamy nikogo, nie pokazujemy palcem winnego. Nie zmienia to faktu, że choć z metodologicznego punktu widzenia podejście prezentowane w tej książce może być kontrowersyjne, jednak jest to publikacja wartościowa, pokazująca, jak szukać problemów w oprogramowaniu.

Testerzy, mówiąc „Lubię testować, bo lubię psuć”, przypisują sobie „zasługi” innych osób. Ci, którzy tworzą – analitycy, architekci i programiści – mogą jednocześnie psuć. Powtórzę, testerzy jedynie dostarczają informacje o zepsuciu. Co prawda, tester również popełnia błędy, ale ich waga i konsekwencje są zazwyczaj pomijalne w porównaniu do wagi błędów popełnianych przez osoby odgrywające inne role projektowe. Znajdziemy to w ciekawej prezentacji błędów popełnianych przez testerów, „Classic Testing Mistakes”, autorstwa Matthew Heussera.

Błędy popełniamy w różnych fazach testowania i wykonując różne czynności testerskie. Błędem jest również traktowanie cudzej odpowiedzialności jako własnej.

Autor dzieli wszystkie problemy powiązane z testowaniem lub z niego wynikające na pięć obszarów:

#### ***Błędy w obszarze „rola testowania”***

- *Zespół testerski jest odpowiedzialny za zapewnienie jakości.*
- *Celem testowania jest znajdowanie defektów.*
- *Problemy użyteczności nie są defektami.*
- *Brak skupienia się na estymowaniu jakości (i na jakości tych estymat).*
- *Raportowanie o defektach bez odwołania się do kontekstu.*
- *Zbyt późne testowanie.*

#### ***Błędy w obszarze „planowanie wysiłku testowego”***

- *Koncentracja jedynie na testach funkcjonalnych.*
- *Niskie zaangażowanie w testowanie konfiguracji.*
- *Zbyt późne testowanie obciążenia i przeciążenia.*
- *Brak testów dokumentacji.*
- *Brak testowania instalacji.*
- *Zbyt duże zaufanie do testów beta.*
- *Zasada kończenia jednego zadania, zanim rozpoczniemy następne.*
- *Niepowodzenie w identyfikacji obszarów ryzyka.*
- *Upieranie się przy planie.*

#### ***Błędy w obszarze „czynniki ludzkie”***

- *Używanie testowania jako wstępu do programowania.*
- *Rekrutowanie testerów spośród nieudanych programistów.*
- *Testerzy, którzy nie są ekspertami w danej domenie.*
- *Nieszukanie testerów w grupie świadczącej usługę wsparcia dla klienta i wśród osób odpowiedzialnych za tworzenie specyfikacji technicznej.*
- *Zespół podobnych do siebie ludzi, brak budowania opartego na różnorodności.*

- *Naciskanie, aby testerzy programowali.*
- *Oddzielenie testerów od programistów.*
- *Wiara, że programiści nie mogą testować własnego kodu.*
- *Programiści, którzy nie są wytrenowani i zmotywowani do testowania własnego kodu.*

#### ***Błędy w obszarze „tester w pracy”***

- *Większe skupienie na uruchamianiu testów niż na ich projektowaniu.*
- *Brak przeglądów opartych na specyfikacji testów.*
- *Zbyt duża koncentracja na danych wejściowych i procedurach.*
- *Niedostrzeganie i nieeksplorowanie interesujących odchyleń.*
- *Brak testowania negatywnego.*
- *Testy zrozumiałe jedynie dla twórcy.*
- *Testowanie jedynie interfejsu graficznego.*
- *Niskiej jakości raporty o defektach.*
- *Dodawanie testów regresyjnych zamiast badania już istniejących.*
- *Brak notatek do następnego zadania testowego.*

#### ***Błędy w obszarze „technologiczny przesyt”***

- *Próba automatyzowania wszystkiego.*
- *Próba wykonania wszystkich testów manualnych za każdym razem.*
- *Używanie testów nagrywająco-odtwarzających do redukcji kosztów wytworzenia automatów testowych.*
- *Oczekiwanie, że testy regresyjne znajdą dużo nowych błędów.*
- *Zbytne pokładanie nadziei w pokryciu kodu.*
- *Usuwanie testów z zestawu regresyjnego tylko dlatego, że nie podwyższają pokrycia.*
- *Mierzenie wydajności testerów miarami uzyskanego pokrycia.*
- *Całkowita rezygnacja z pokrycia.*

Kolejną wartą polecenia publikacją jest artykuł „The Seven Habits of Highly Effective Testers” Stevena Millera, który opisuje różnicę między testerem a dobrym testerem w postaci siedmiu cech.

#### ***Cecha 1. Bądź proaktywny***

*Działaj aktywnie na rzecz jakości oprogramowania:*

- *Przyjmij odpowiedzialność za dobre wymagania.*
- *Analizuj zdolność do śledzenia jakości oprogramowania i jakość dokumentacji testerskiej.*
- *Komunikuj się efektywnie.*
- *Opisuj defekty efektywnie.*

#### ***Cecha 2. Zaczynając, myśl już o końcu***

*Dobre przygotowanie do projektu testerskiego to zdefiniowanie celu. Zanim poważnie zaczniesz, zastanów się, w którym miejscu uznasz, że odniosłeś sukces.*

### **Cecha 3. Najważniejsze rzeczy na początku**

*Priorytety, priorytety, priorytety. Nie marnuj energii na rzeczy wtórne i nieważne, najpilniejsze tematy stawiaj na pierwszym miejscu. Na przykład, zanim weźmiesz się za testy negatywne, sprawdź, czy pozytywne są ukończone.*

### **Cecha 4. Myśl w kategoriach Win/Win (zwycięstwo/zwycięstwo)**

*W odwiecznej walce programista – tester postaraj się znaleźć równowagę. Zamiast zrzucić na nich winę, spróbuj rozwiązać sytuację tak, by i wilk był syty, i owca cała. Kilka dobrych rad, jak to zrobić:*

- Dziel się wiedzą.
- Poznaj swój zespół nie tylko w czasie pracy.
- Chwal za sukcesy.
- Oferuj pomoc i pomagaj.

### **Cecha 5. Na początku zrozum, potem postaraj się być zrozumianym**

*Słuchaj opinii drugiej strony, zanim wypowiesz swoje zdanie. Postaraj się zrozumieć racje i w klarowny sposób odpowiedz swoimi argumentami. Nigdy nie stawiaj sprawy na ostrzu noża!*

### **Cecha 6. Staraj się o synergię**

*Każdy członek zespołu coś do niego wnosi. Swoje doświadczenie, wiedzę, charakter, złe i dobre nawyki. Znając swoich współpracowników, maksymalizuj efektywność współpracy z nimi. Współpracujący członkowie zespołu współdzielą kalendarze, materiały i dyskutują.*

### **Cecha 7. Bądź ostrzejszy niż brzytwa**

*Żyj testowaniem, czytaj najnowsze publikacje, bierz udział w aktywnych grupach testerskich i automatyzuj wszędzie tam, gdzie skraca to twój czas w pracy. Ładuj baterie z dala od testowania. Miej hobby i wypoczywaj na niezapomnianych wakacjach.*

Steve Miller, prezes Pragmatic Software, napisał swój artykuł, opierając się na kultowej już publikacji Stephena Coveya *The Seven Habits of Highly Effective People*.

Na naszych szkoleniach określamy cechy najlepszego testera. Warto przy tej okazji sięgnąć po celną uwagę Cem Kanera: *The best tester isn't the one who finds the most bugs or embarrasses the most programmers. The best tester is the one who gets the most bugs fixed*. Co to właściwie znaczy? Najlepszy tester to nie ten, który znajduje najwięcej defektów lub zawstydzi największą liczbę programistów, ale ten, którego zgłoszone defekty są naprawiane. Zgłaszając defekty, mamy nadzieję, że ktoś je poprawi, bez względu na wagę, pilność czy obszar. Bo po co o nich raportować, wiedząc, że i tak nie zostaną naprawione? Zgłaszając defekty, które zostają usunięte, udowadniamy, że wiemy, co jest ważne dla klienta, kierownika projektu czy nawet programisty.

Z kolei Mike Meurs opisał na swoim blogu to, co cechuje złego testera i co może świadczyć o tym, że nie nadajesz się do tego zawodu:

- *Denerwuje cię wadliwość oprogramowania.*
- *Nie znasz oczekiwań biznesowych swojej firmy.*
- *Męczyci cię wyjaśnianie czynności prowadzących do wystąpienia defektu.*
- *Nie czytasz książek, blogów, nie uczestniczysz w konferencjach na temat testowania oprogramowania.*
- *Wstydzisz się swojej roli w projekcie.*
- *Wiesz, jak coś sprawdzić, ale nie masz ochoty się w to zagłębiać.*
- *Nie jesteś na bieżąco z technicznymi aspektami IT.*
- *Nie lubisz się komunikować z innymi.*
- *Nie znasz całego cyklu rozwoju aplikacji.*
- *Nie umiesz się pogodzić z tym, że incydenty zgłoszone przez ciebie nie zostaną naprawione.*

Przytaczam tu wiele cudzych materiałów, ale jest ku temu kilka powodów. Najpoważniejszy to ten, że uważam je za wartościowe dla początkujących testerów. Mogłbym je, co prawda, parafrazować i ubrać w swoje słowa, ale nie czuję takiej potrzeby. Wystarczy mi, że poświęciłem czas i uwagę, aby je znaleźć i przetłumaczyć. Wiem, że w wielu miejscach tej książki można znaleźć rzeczy, które wydają się czytelnikom znajome. Nie ukrywam, że moje poglądy są wypadkową wszystkich książek, które przeczytałem, wpisów blogów, które przejrzałem, setek prezentacji, w których uczestniczyłem, i tysięcy rozmów, które przeprowadziłem. Są przefiltrowane przez moje postrzeganie świata testowania i podane w formie, którą uważam za optymalną. Pod treściami tak obficie tu cytowanymi mogłbym się podpisać obiema rękami, a ich autorom składałem hołd i buduję ten papierowy pomnik.

## 6.16. Praca w charakterze testera

Testowanie oprogramowania jako usługę można świadczyć na wiele sposobów. W każdym z tych modeli możesz, czytelniku, odnaleźć pracę. Mają one swoje zalety, mają również wady, a każda z nich funkcjonuje na rynku i warto je znać.

## Zasób

Rewolucja przemysłowa i wszelkiego rodzaju metody optymalizacji pracy doprowadziły do tego, że osoby pracujące na cudzą rzecz traktowane są przedmiotowo. Całość spotęgowało powstanie zarządzanych przez fundusze inwestycyjne międzynarodowych korporacji, w których człowiek jest jedynie kolejną cyfrą w tabeli. Jak inaczej można nazwać określanie ludzi jako „zasób”? Osobiście wolę się odwołać do wzorców, które mogą się kojarzyć z podejściem socjalistycznym lub z pracą fizyczną, i o pracownikach mówić jako o sile roboczej.

Prosty test na przejście od „pracownika” do „zasobu” – kiedy bezpośredni przełożony twojego przełożonego nie wie, jak masz na imię.

**Umowy testerskie.** Podstawą współpracy dwóch podmiotów zawsze będą umowy, które regulują wspólne odpowiedzialności stron i definiują zakres obowiązków. Brak takiej umowy jest pewnym ryzykiem podejmowanym przez testera.

**Praca bez umowy.** W świecie IT trudno spotkać pracujących „na czarno”. Pracodawcom w wielu przypadkach zależy, aby wartościowych pracowników do siebie przywiązać i robią to za pomocą umów, w tym o pracę. Co prawda, polski system składkowo-podatkowy zniechęca wszystkich, zarówno pracodawców, jak i pracowników, do formalizowania współpracy, ale w IT nikt się nie bawi w małe oszczędności. Nasz cel to tworzenie software’u, na software jest popyt, więc jest drogi. Mamy zatem z czego płacić państwu haracz.

**Umowa o pracę.** Firmy planujące rozwijać swoje oprogramowanie i dbać o jego jakość zatrudniają testerów i tworzą działy testów. Jest to najprostszy, choć kosztowo nie najbardziej efektywny model.

Praca najemna jest formą umowy między pracodawcą i pracownikiem, w której obie strony deklarują chęć współpracy na określonych warunkach. Osoba zatrudniana w zamian za ustaloną stawkę deklaruje gotowość do wykonywania prac powierzonych przez pracodawcę. Model ten przerzuca na pracodawcę większość obowiązków związanych z umową, organizacją zadań czy stanowiskiem pracy. Jest to model najprostszy dla pracownika.

Zalety modelu z perspektywy pracodawcy:



- dostępność – tester lub dział testów jest organizacją wewnętrzną i może wykonywać zlecone prace w dowolnym czasie. Długi czas rekrutacji kompensowany jest przez łatwiejszy dostęp do zasobów ludzkich w późniejszym czasie;
- bezpieczeństwo – informacje pozostają wewnątrz firmy, więc i ryzyko wycieku know-how na zewnątrz jest mniejsze.

Metoda rozliczania polega na tym, że pracodawca wypłaca pensję pracownikowi za czas przepracowany oraz za nieprzepracowany, jeśli wynika to z praw pracowniczych (urlopy, święta itp.). Możliwe są również premie uznaniowe za dobre wykonanie prac.

**Umowa zlecenie lub umowa o dzieło.** Część firm decyduje się na umowy potocznie nazywane śmieciowymi. Oferta pracy na umowę zlecenia czy o dzieło może wskazywać na to, że firma nie ma stabilnych projektów i potrzebuje elastyczność w zarządzaniu zasobami. Elastyczne zarządzanie zasobami jest tutaj oczywiście eufemizmem zastępującym stwierdzenie: musi mieć możliwość redukcji pracowników. Umowy tego typu są również próbą „optymalizacji składowej”, czyli ograniczenia zobowiązań wobec ZUS-u. Nie należy się jednak obawiać umów zlecenia. Jest to dobra propozycja przy pierwszej pracy. Jeśli pracodawca oferuje zatrudnienie i ten typ umowy, a ty masz niewielkie doświadczenie i dużą chęć spróbowania, to zyskują obie strony. Nie powinno to być jednak rozwiązanie długofalowe. Docelowo należy zmienić umowę zlecenia na umowę o pracę lub też na umowę firma–firma, co jest korzystne dla obu stron.

W umowie zlecenia, jak sama nazwa wskazuje, zazwyczaj zlecana jest jakaś praca do wykonania. Zlecenie szczegółowo opisuje, co należy do twoich obowiązków i jaką zapłatę za to otrzymasz. Podobnie jest z umową o dzieło, gdzie definiuje się zazwyczaj wytworzenia pewnego „dzieła” i dostarczenie go pracodawcy. Ponieważ o dziele zazwyczaj mówimy w przypadku pracowników, którzy w oczach urzędników naprawdę coś tworzą (typu budują pomniki), a nie piszą wirtualne dokumenty, to rozwiązanie jest rzadziej stosowane w testowaniu. Również tutaj pojawia się określona stawka za wykonaną pracę.

Zalety modelu z perspektywy pracodawcy:

- Relacja jakości do ceny – tester zgadzający się na taką umowę zazwyczaj będzie relatywnie tani i nie można od niego wymagać zbyt wiele. Dostaje się to, za co się zapłaciło.

- Elastyczność – testera na umowie cywilnoprawnej można dużo łatwiej zwolnić bez okresu wypowiedzenia.
- Taniość i przejrzystość – umowy cywilno-prawne nie dają pracownikowi prawa do urlopu czy zasiłku chorobowego. Pracodawca jest więc lepiej zorientowany w kosztach takiej umowy, czego nie będzie przy umowie o pracę.

Metody rozliczania to zazwyczaj zapłata tylko za przepracowany czas lub za wykonaną usługę.

**Własna działalność gospodarcza.** Wiele osób boi się ją rozpocząć. W mojej opinii własna działalność gospodarcza pozwala na lepszą kontrolę środków otrzymywanych od pracodawcy. Oczywiście podczas zakładania firmy pojawiają się kłopoty z rejestracją, problemy z urzędnikami, dodatkowe koszty (np. obsługi księgowej), ale będą też korzyści. Na przykład przez pierwsze dwa lata działalności składki ubezpieczeniowe są niższe, dzięki czemu więcej środków zostaje w kieszeni. Po za tym można zmniejszać podatek, inwestując i odliczając koszty. Twoje mieszkanie staje się twoim biurem, jest więc (przynajmniej częściowo) kosztem, samochód jest teraz firmowy i wydatki z nim związane również są kosztami obniżającymi podatek.

Zakładając własną działalność gospodarczą, możesz też liczyć na przychylniejszy stosunek pracodawcy – czy raczej zleceniodawcy – oraz wyższe stawki. Zleceniodawca, redukując swoje obciążenia związane z ZUS-em i urzędem skarbowym, może te środki przynajmniej częściowo oddać tobie. Pamiętaj, że jeśli zatrudniony na umowę o pracę dostaje na rękę 3000 zł, to koszt pracodawcy sięga 5000 zł. Różnica trafia do systemu zdrowotnego, emerytalnego i podatkowego i realnie nie masz nad nią żadnej kontroli, a jedynie deklaracje urzędników i polityków, że masz za to zapewnione leczenie, emeryturę i np. lepsze drogi. „Zaoszczędzone” 2000 zł pozwolą ci spokojnie opłacić leczenie i prywatne ubezpieczenie oraz wpłacić trochę mniej na drogi.

Nie zachęcam natomiast to rozpoczęcia swojej przygody z testowaniem od własnej działalności gospodarczej. Warto na początek rozejrzeć się na rynku i sprawdzić, czy są firmy gotowe współpracować w takim modelu. Gdy podczas rozmowy kwalifikacyjnej padnie takie pytanie, możesz zawsze zadeklarować wolę założenia własnej firmy.

Zalety modelu z perspektywy pracodawcy:

- wyższa cena za czytelne zasady – umowa firma–firma pozwala zleceniobiorcy określić koszt usługi. Poza tym umowa między dwoma podmiotami może wносить dodatkowe ograniczenia i zasady, które nie są możliwe w umowach z osobami cywilnymi. Przykładowo można wprowadzić zapisy odnośnie uczciwej konkurencji. W przypadku takiej współpracy włącza się również tryb „kreatywna księgowość”. Z różnych powodów różne podmioty muszą ograniczać zatrudnienie, które przez np. akcjonariuszy może być uznane za podniesienie kosztów funkcjonowania organizacji, czyli trend negatywny. Natomiast „zatrudnienie” osoby prowadzącej własną działalność gospodarczą, może to być traktowane jako inwestycja. Pokrętna logika, ale tak działa biznes.
- Elastyczność – testera na umowie firma–firma można dużo łatwiej zwolnić, wypowiadając umowę.

### **Płać, jeśli chcesz testować**

Bardzo dochodowa branża oprogramowania rozrywkowego wpadła na pomysł, by użytkownicy płacili za wersję demo gier. Co gorsza, firmy produkujące gry chcą, żeby gracze płacili również za wersje oprogramowania, które ukazuje się przed premierą, czyli *de facto* za wersje beta.

Przy spadającej jakości oprogramowania spodziewalibyśmy się raczej szerszego dostępu do wersji beta i płacenia testerom za pracę, którą powinni wykonywać producenci. Okazało się, że firmy chcą również zarabiać na niepełnowartościowym produkcie.

Jest pewne, że znajdą się maniacy, którzy będą chcieli zapłacić za obejrzenie gry jeszcze przed jej oficjalną premierą. Nie są to jednak specjaliści od jakości i mogą testować raczej *ad hoc* niż systematyczne.

## **6.16.1. Modele współpracy**

Testować można również na zasadach współpracy firma–firma (inna niż jednoosobowa działalność gospodarcza) czy też przez pozyskiwanie zasobów z globalnej sieci.

### **Warto słuchać starszych i mądrzejszych**

*Jeśli jest coś, czego nie potrafimy zrobić wydajniej, taniej i lepiej niż konkurenci, nie ma sensu, żebyśmy to robili i powinniśmy zatrudnić do*

wykonania tej pracy kogoś, kto zrobi to lepiej niż my – Henry Ford.  
Praca będzie lokowana i wykonana tam, gdzie ma to największy ekonomiczny sens – Nandan Nilekani, prezes firmy Infosys Technologies.

**Crowdsourcing.** Testowanie oprogramowania jest inwestycją, na którą nie każdą firmę wytwarzającą oprogramowanie stać. Koszty zbudowania kompetencji testerskich, zarządzania jakością, samego przygotowania i wykonania testów... Lista wydatków, jakie należy ponieść, może być naprawdę długa.

Model wykonywania pracy zdalnej przez testerów w internecie w Stanach Zjednoczonych jest już znany od wielu lat. W Polsce dopiero zdobywa on zainteresowanie jako kolejna próba oszczędzenia na testowaniu. *Crowd* w tym konkretnym przypadku jest tłumem użytkowników internetu. To słowo po raz pierwszy zostało użyte w magazynie „Wired”, a wymyślone przez Jeffa Howe’a w czerwcu 2006 roku.

Budowanie społeczności, które przejmują tradycyjne obowiązki pracowników najemnych, jest nowym i popularyzującym się modelem pozyskiwania wykonawców. Główną zaletą takiego podejścia jest to, że (potencjalni) odbiorcy końcowi najlepiej wiedzą, czego potrzebują, i że właśnie oni są w stanie stworzyć produkt najbardziej dla nich odpowiedni. Ocenianie produktu staje się w tym przypadku jakby procesem oddolnym. Zlecający oferuje możliwość wypowiedzenia się na temat produktu, którego jakieś osoby już używają lub chciałyby zacząć używać. Daje to także szansę na odświeżenie spojrzenia na zagadnienie i udoskonalenie produktu, nadążanie za zmianami i potrzebami grupy docelowej. Omawiam tu najbardziej rozpowszechnione modele pozyskiwania do projektu zasobów „z tłumu”:

- **Oplata za wykonane zadanie lub za przepracowane godziny** – model ten praktycznie niczym się nie różni od zadań wykonywanych zdalnie, z domu, dla konkretnego pracodawcy na podstawie np. umowy zlecenia. Tutaj jednak kontaktują się dwa podmioty, z których jeden chce coś otrzymać, a druga strona deklaruje wytworzenie czegoś lub wykonanie pracy. W tym wypadku zleceniodawca często ogłasza aukcję i wybiera najciekawszą (często najtańszą) ofertę. Model taki funkcjonuje

w portalach zagranicznych, takich jak [freelancer.com](https://www.freelancer.com) czy [odesk.com](https://www.odesk.com). Niestety polskie prawo nie nadąża za takim „nowinkami”, znanymi na Zachodzie od lat, i ciągle jeszcze model rozliczeń z wirtualnym i zdalnym wykonawcą wymaga sporządzenia umowy zlecenia czy umowy o dzieło, nawet jeśli chodzi o niewielkie kwoty.

- **Pay-per-bug – płać za defekty** – mowa o rozwiązaniu, w którym zlecający płaci jedynie za rzeczywiście znalezione defekty. Portale takie jak [utest.com](https://www.utest.com) przynoszą korzyści zarówno firmom produkującym oprogramowanie, jak i testerom pragnącym testować, nie wychodząc z domu. Jak to działa? Producent programu udostępnia go do testów, definiując ich zakres (błędy funkcjonalne, użyteczność, wydajność). Testerzy szukają defektów i otrzymują zapłatę jedynie za znalezione defekty. Z tym ciekawym model współpracy wiąże się kilka kontrowersji. Pierwsza to duplikaty – ludzie będą raportowali te same problemy albo z tej samej kategorii. Zapłatę dostaje oczywiście ten, kto był pierwszy. Drugi problem: kto określa, co jest defektem, a co nim nie jest? Albo co należy do poprawnej kategorii, a co nie? Jeśli decyzję zostawimy zlecającemu, to może się okazać, że większość zgłoszeń zostanie odrzucona, a później i tak naprawiona. Kolejnym problemem będą stawki. Możesz dużo przepracować i niewiele zarobić, bo stawki za pojedynczy defekt funkcjonalny to np. 2–5 zł. Nie nastawiaj się w takim razie na duże zarobki, ale korzystaj z tego modelu. Jest to naprawdę dobry sposób na poznanie innych testerów, ich metod raportowania i wynajdywania defektów.
- **Crowdcasting** – model ten jest zbliżony omówionych wcześniej. Testować może wiele osób, ale zarobi tylko najlepszy. Wszyscy szukają defektów, zgłaszają je, ale jest tylko jedna nagroda, np. za defekt najbardziej krytyczny. Tu również pojawiają się wspomniane kontrowersje i okazje raczej edukacyjne niż finansowe.

**Insourcing.** Dostarczanie ludzi o wysokich kompetencjach w testowaniu przez firmy oferujące konsultantów staje się coraz popularniejsze. Metoda taka nazywana jest insourcingiem. Agencje w Polsce i na świecie, które rejestrują osoby umiejące testować oprogramowanie dostarczają godziny czy też dni pracy konsultantów.

Zalety z perspektywy zamawiającego:

- Szybkość – w relatywnie krótkim czasie zleceniodawca znajduje osoby, które pomogą jego testerom w pracy.
- Unikatowe kompetencje – w niektórych przypadkach firmy nie potrzebują przeprowadzać pełnej rekrutacji pracownika o danych kompetencjach, jeśli jest to jedynie krótki test w specjalizowanym obszarze. Przykładem mogą być testy użyteczności, gdzie wystarczy zaangażować konsultanta na czas trwania prac nad interfejsem aplikacji.
- Kreatywność księgowa – niektórym organizacjom bardziej opłaca się kupować pracę od innych firm niż samodzielnie zatrudniać pracowników. W wielu przypadkach może chodzić o umowy z lokalnymi władzami (np. w zamian za zwolnienia podatkowe „inwestuje się” w lokalny rynek) lub wpisanie danego kosztu do innej rubryki księgowej (np. wydanie środków finansowych jest traktowane jako inwestycja, podczas gdy zatrudnienie – jako podnoszenie kosztów funkcjonowania).
- Limity – firma-matka organizacji (zazwyczaj zagraniczna centrala) może ograniczać wzrost i nie pozwalać na zatrudnienie. Kupując konsultantów, mamy ręce do pracy, nie zwiększając grupy naszych pracowników.
- Łatwość redukcji – konsultantów łatwo zwolnić (warunkują to zapisy umowy), gdy firma w krótkim czasie musi ograniczyć koszty funkcjonowania.
- Bezpieczeństwo – insourcing jest dla wielu organizacji bezpieczniejszy niż outsourcing. Konsultanci pracują zazwyczaj w siedzibie i na sprzęcie zamawiającego. Łatwiej więc zadbać o know-how i bezpieczeństwo informacji.

Podstawą rozliczania w tym modelu jest roboczogodzina (*manhour*). Firma wynajmująca konsultantów obciąża firmę kupującą za godziny pracy.

Nie ma szczególnych warunków bycia konsultantem, ale wiedza i doświadczenie zwiększają szansę na intratne zlecenie. Firmy pozyskujące konsultantów większym zainteresowaniem obdarzą osoby prowadzące własną działalność gospodarczą i mogące wystawiać faktury VAT. Konsultantem można zostać przez zatrudnienie lub zawarcie umowy z jedną z wielu na polskim rynku firm trudniących się insourcingiem testowym lub też próbować samodzielnie pozyskiwać zlecenia. Najczęściej poszukiwani są specjaliści do testów

wydajnościowych, bezpieczeństwa i użyteczności, do automatyzacji testów oraz kierownicy testów.

W tym modelu firmy zajmujące się insourcingiem dla innych mają przewagę nad prowadzącymi własną działalność gospodarczą ze względu na znacząco szersze kontakty i możliwość oddelegowania konsultantów do największych projektów. Jednakże takie firmy mocno zdeprecjonowały pojęcie konsultanta ze względu na proponowanie do tej roli osób bez wiedzy i doświadczenia.

**Outsourcing** jest metodą dostarczania informacji o jakości do projektów informatycznych. Podstawą tego modelu jest założenie, że nie każda firma czy organizacja musi mieć umiejętności, wiedzę i doświadczenie w testowaniu. Może ją więc pozyskać od firm specjalizujących się w tym obszarze.

Firm świadczących usługi testowania w Polsce jest wiele, a rynek ciągle się rozwija. Większość firm, które znajdziesz po wpisaniu do wyszukiwarki Google słów kluczowych „usługi testowania” ma jedynie witrynę internetową i znikome doświadczenie. Istnieje ciągle potencjał dla nowych firm chcących testować oprogramowanie na zlecenie.

Zalety modelu z perspektywy zamawiającego:

- Informacja o jakości wytwarzanego oprogramowania – firmy outsourcingowe są w stanie w szybkim czasie dostarczyć informacje o oprogramowaniu, które dana firma wytwarza. Nie ma więc konieczności tworzenia wewnętrznego działu testów.
- Informacja o jakości zamawianego oprogramowania – wiele organizacji nie zajmuje się tworzeniem oprogramowania, ale na potrzeby swojego funkcjonowania pozyskuje je od zewnętrznych firm. Odpowiedzialność za sprawdzenie tego oprogramowania spoczywa więc na firmie outsourcingowej.

Istnieją dwie możliwości rozliczania prac w tym modelu:

- w zakresie budżetu przekazanego przez zamawiającego testowanie (*fix price*),
- na podstawie kosztów testowania wyliczonych przez firmę outsourcingową (zazwyczaj) za poświęcony czas i inwestycje w same testy (*time and material*).

Ponieważ trudno oszacować koszt testów systemu, który opisany jest (czasami) wymaganiami albo dopiero się go zobaczyło, wykonuje się tak zwane badanie próbki. Jest to próba wyliczenia kosztów na podstawie próbnych testów wykonywanych przez firmę i oszacowania pracy do wykonania.

Każdy może założyć firmę testerską outsourcingową, ale trzeba się liczyć z dużymi kosztami promocji swojej oferty. Testowanie oprogramowania ma na polskim rynku ciągle relatywnie niewielki szacunek wśród firm i wiele organizacji nie jest gotowa do ponoszenia nakładów na testy. Aby pozyskać klientów, można korzystać ze znajomości w branży lub też dynamicznie promować swoją ofertę. Najczęściej poszukiwane są firmy wykonujące testy funkcjonalne, jednak przydają się również kompetencje w innych obszarach testowania, takich jak wydajność czy bezpieczeństwo.

Jeśli firma będzie się rozwijać, to warto uświadomić sobie inne opcje w ramach outsourcingu, czyli offshoring i nearshoring. Offshoring wiąże się z przeniesieniem wybranych działań testowych lub ich części poza granice kraju, najczęściej do regionów zamorskich. Mogą to być usługi świadczone w danym kraju, np. w Polsce, ale sprzedawane zagranicznym kontrahentom. Offshoring stanowi dla przedsiębiorców źródło przewagi konkurencyjnej, głównie ze względu na obniżenie kosztów pracy. Offshoring jest jednym z najbardziej widocznych przejawów globalizacji. Można wymienić różne jego kategorie, np.:

- outsource offshoring – usługę świadczą niezależni partnerzy zagraniczni,
- captive offshoring – w którym mamy do czynienia z jednostkami zależnymi, które powstały z myślą o przeniesieniu usługi poza granice kraju. Są to tzw. bezpośrednie inwestycje zagraniczne, realizowane przez wyspecjalizowany w tym celu dział firmy. Wadą takiego rozwiązania jest to, że jest on bardzo kapitałochłonny, zwłaszcza w początkowym okresie inwestycji w innym kraju.

Jeśli usługi przeniesiono do państwa położonego w tej samej części świata, w ramach tego samego kontynentu, bliskiego pod względem geograficznym i kulturowym, to używa się określenia „nearshoring”. Dzięki temu zostaje zachowany cykl produkcyjny (ta sama strefa czasowa) oraz sposób zarządzania personelem (brak różnic kulturowych). Te dwa czynniki oraz związane z przeniesieniem



obniżenie kosztów produkcji powodują, że nearshoring zyskuje na popularności.

# 7. Praktyka testowania

## 7.1. Wprowadzenie

Teoria jest przydatna w ogólnym przygotowaniu się do testowania, ale nic nie zastąpi praktyki. Piszac „praktyki”, mam na myśli zarówno praktykę innych ludzi, którzy chcą się dzielić wiedzą, jak i „praktykę”, którą można zdobyć podczas własnego wykonywania działań testowych.

Postanowiłem uzupełnić książkę o materiały bardziej praktyczne. Część z nich zbierana była do innej publikacji, która nie miała okazji się ukazać. Jest to zbiór testów, który definiowałem samodzielnie lub też przy pomocy pracowników mojej firmy jako uniwersalną bazę wiedzy dla testerów. Część z tych materiałów była wcześniej publikowana przez nas na nieistniejącym już [forum.testrzy.pl](http://forum.testrzy.pl), które prowadziliśmy w sieci, część to testy zaproponowane przez samych forumowiczów.

W tym rozdziale poruszam przede wszystkim temat przygotowania się do testowania, uruchomienia testowania i na końcu raportowania. Nie zakładam, że każde z tych zadań znajdzie się w zakresie formalnej odpowiedzialności testera, ale wiem z doświadczenia, że na pewno będzie w zakresie odpowiedzialności nieformalnej. Nawet jeśli ktoś inny powie ci jak, gdzie i jak długo masz testować, to za wypełnienie czasu od „teraz” do „koniec testów” odpowiadasz ty. Wymaga to przygotowania, zaplanowania pracy i realizowania ustalonego planu. Na koniec będziesz musiał poświęcić uwagę przygotowaniu i przekazaniu raportu. Nasza praca będzie niewiele warta, jeśli nie potrafimy pochwalić się jej rezultatami.

Wszystkie materiały odnoszę do poziomu zrozumienia osób początkujących, które z testowaniem miały do tej pory niezbyt wiele do czynienia.

## **7.2. Podejścia do testowania**

To jak podejździemy do testów oprogramowania ściśle zależy od kontekstu jego użycia, modelu dostawy, dostępności interfejsu i przede wszystkim celu, jaki sobie stawiamy. Co więcej, nie istnieje jedna, uniwersalna strategia testowania (terminy „podejście” i „strategia” będą stosować zamiennie). Wszystkie opisane podejścia mogą być łączone i modyfikowane, by wybrać to, które najlepiej zweryfikuje system w odniesieniu do postawionego zadania testerskiego.

### **7.2.1. Strategie wynikające z podziałów w testowaniu**

W testach możemy się zorientować na przedstawione już wcześniej kategorie dzielenia na obszary. Możemy więc definiować specyficzne podejście do testów, np. charakterystyk oprogramowania, w tym bezpieczeństwa lub niezawodności czy testowania kodu.

Możemy również nasze strategie oprzeć na nieopisanych w tej książce działaniach, takich jak rozbudowana automatyzacja czy też wyróżnienie testowania w cyklu życia oprogramowania (np. wytwarzanie nowej aplikacji kontra jej utrzymanie).

Aspekty podejść do testowania i specyfiki testów w tych obszarach pozostawiamy już testerom z większym stażem i umiejętnościami w danej domenie. Nie będą one szerzej opisane.

### **7.2.2. Strategia testowania oparta na modelu dostarczania**

Oprócz wcześniej wymienionych kategorii i podziałów w testowaniu możemy również spojrzeć na aplikacje pod kątem bardziej praktycznym. Inaczej będziemy testowali aplikacje desktopowe, mobilne, webowe i wbudowane. Inne będzie nasze podejście do testów, białoskrzynkowe, bo inny jest kontekst samej aplikacji.

Każdy typ oprogramowania będzie miał standardowy zestaw testów do uruchomienia. Pierwszym i najważniejszym będzie oczywiście funkcjonalność oprogramowania i jego zdolność do operowania w zdefiniowanym środowisku. W pierwszej kolejności będziemy próbowali weryfikować naszymi testami funkcje dostępne w interfejsie

(czarna skrzynka), aby później spróbować testować interfejsy np. do systemu operacyjnego czy też do innych systemów.

Każda z aplikacji będzie miała unikatowe dla siebie metody testów lub istotne obszary zainteresowania, ze względu na samą formę czy też platformę jego dostarczenia. Przykładowo:

- Aplikacje desktopowe oprócz weryfikacji funkcji i integracji testów będą się w dużej mierze koncentrowały na zasobach sprzętowych (pamięci operacyjnej i dyskowej, procesorze), zarządzaniu plikami, w tym ich tworzeniu, usuwaniu, modyfikowaniu oraz na konfiguracjach sprzętowo-software'owy. Będziemy chcieli sprawdzić wsparcie dla kolejnych wersji systemów operacyjnych, kompatybilności z innym oprogramowaniem zainstalowanym w środowisku, z uwzględnieniem potencjalnych konfliktów. Trzeba tu spojrzeć głębiej w aplikację i zrozumieć mechanizmy jej działania, np. czy odzyskiwane są dane po wyłączeniu sprzętu, na którym uruchomiona jest aplikacja, czy też skąd pobierany i jak przetwarzany jest czas systemowy.
- Aplikacje mobilne będą miały wiele wspólnego z aspektami wyróżnionymi dla aplikacji desktopowych, ale też ze specyfiką łączności bezprzewodowej (w tym obsługi utraty połączenia), ograniczeniem w interfejsie (np. brak myszy) i mocnych testów na styku sprzętowo-software'owym (np. tworzenie załączników do wiadomości ze zdjęć zrobionych z poziomu aplikacji). Sprzęt mobilny ma wiele elementów nietypowych dla komputerów, np. specyficzną wielowątkowość i gotowość na to, że nowy wątek niekoniecznie zostanie wywołany przez użytkownika, ale przyjdzie niespodziewanie z sieci bezprzewodowej (np. połączenie przychodzące). Sprzęt mobilny może mieć również wbudowaną lokalizację użytkownika, co powoduje, że wiele aplikacji otrzymuje dodatkowe i ważne źródło informacji o kontekście funkcjonowania użytkownika. Testy mobilne to również zmiana orientacji wyświetlania treści aplikacji z pionowej na poziomą (i na odwrót) i mnogość platform, rozdzielczości czy też mocy obliczeniowych urządzeń mobilnych. Dla lepszego zrozumienia specyfiki testowania aplikacji mobilnych polecam publikację Jonathana Kohla zawierającą opis podejścia I SLICED UP FUN.
- Aplikacje webowe będą musiały uwzględnić specyfikę internetu. Przy założeniu, że obsługujemy skończoną liczbę przeglądarek,

podczas testowania należy pamiętać, że nie każdy użytkownik będzie korzystać z naszej rekomendowanej przeglądarki. Aplikacja webowa to też różna nawigacja i użycie przez zaawansowanych użytkowników skrótów klawiaturowych, np. Wstecz czy Odśwież. Testy to również styk podejść czarnej i białej skrzynki. Ponieważ większość kodu (CSS, HTML, JScript) dostępna jest po stronie przeglądarki, możemy dokonywać wielu testów bez odwoływania się interfejsu graficznego. Mamy możliwość „utrudniania” pracy aplikacji przez modyfikację plików cookie, modyfikowanie parametrów zapytań w komunikacji za pomocą protokołu HTTP/HTTPS czy zablokowanie uruchomienia skryptów po stronie przeglądarki. Ciekawe testy to również próba otwierania tej samej sesji w różnych przeglądarkach oraz statyczna analiza przez sprawdzanie kodu w dostępnych online walidatorach poprawności. Na to wszystko możemy narzucić również testowanie w różnej, niekoniecznie obsługiwanej rozdzielczości itp. Zweryfikowanie, czy aplikacja sobie z tym poradzi i czy będzie w stanie świadczyć swoje usługi to test specyficzny dla tego środowiska. Dodatkowe testy będą dotyczyły standardowych komunikatów błędów odpowiedzi HTTP, czyli np. 404 – „Strony nie odnaleziono”. Pojawiają się zazwyczaj wtedy, gdy źle skonstruowano link do strony lub gdy dana strona została usunięta. Ponadto testowanie musi określić, czy nie ma stron niepodlinkowanych, czyli takich, do których nie można się dostać, nie znając ich dokładnego adresu.

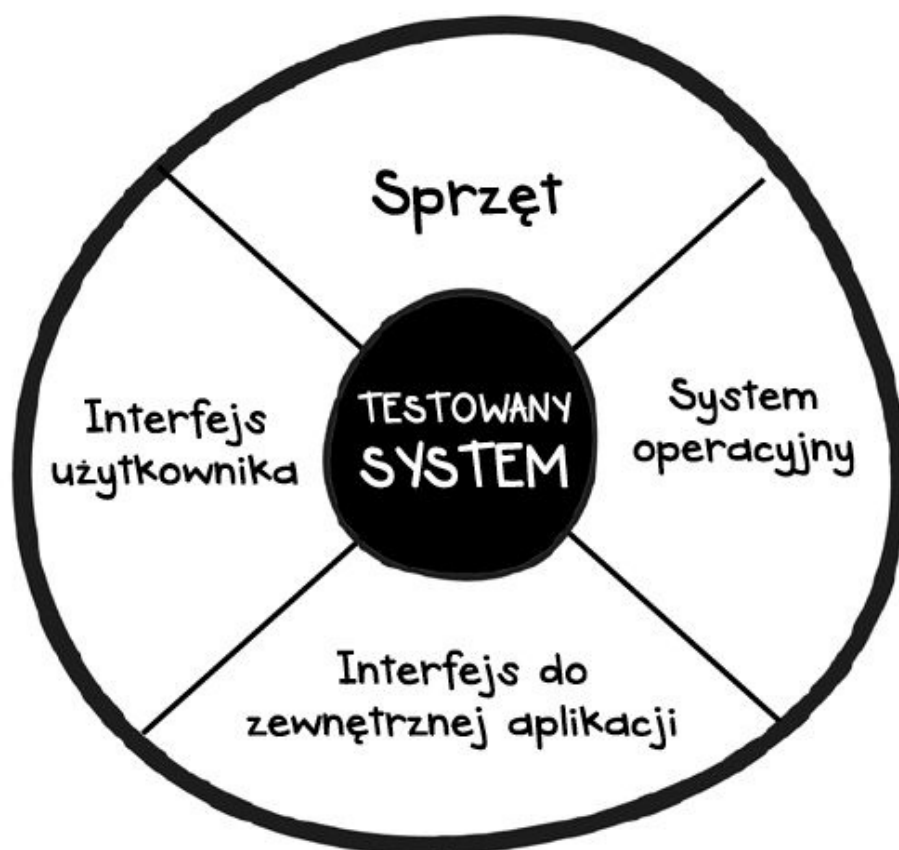
- W przypadku aplikacji wbudowanych zazwyczaj wiele testów będzie dotyczyć obszaru stanów i przepływów sterowania. Będziemy musieli również poradzić sobie z „inną” konstrukcją interfejsu, zazwyczaj opartego na wyświetlaczu i przyciskach. Często w systemach pojawi się problem kompletnego braku interfejsu w samym urządzeniu, a dostęp do niego możliwy będzie dopiero po podpięciu zewnętrznego urządzenia. Oprogramowanie wbudowane jest często krytyczne, a niepoprawność jego działania może narazić użytkowników na utratę życia i zdrowia, więc wymaga wzmożonej uwagi podczas testów.

Choć większość funkcji jest mocno powtarzalna, już metoda operowania na interfejsie ściśle zależy od środowiska dostarczenia oprogramowania. Czasami dostępna jest pełna klawiatura QWERTY, czasami jedynie uproszczona klawiatura numeryczna. Niekiedy

interfejs to fizyczne klawisze, a innym razem to ekran dotykowy. Czasami mamy ekran wysokiej rozdzielczości lub wyświetlacz, a czasami widzimy tylko zapalające się i gasnące lampki. Do tej mnogości metod i interfejsów niełatwo znaleźć pulę testów uniwersalnych.

### **7.2.3. Podejście negatywne do testów, czyli atak na oprogramowanie**

Definiując testy, zazwyczaj dzielimy je na pozytywne, które ogólnie sprawdzają działanie oprogramowania w najprostszych lub najbardziej realnych scenariuszach, oraz polegające na podejściu negatywnym, które może być traktowane jako atak na oprogramowanie. Wiele książek opisuje te bardziej destrukcyjne metody, których celem jest udowodnienie, że istnieją okoliczności, w których aplikacja się wyłączy lub utraci zdolność poprawnego działania. Oczywiście z łatwością możemy do tego doprowadzić, więc nie szukamy odpowiedzi na pytanie, czy aplikację można sparaliżować, ale jak się ona zachowa w trakcie tego paraliżu oraz potem, gdy czynnik paraliżujący przestanie działać.



**Rys. 7.1.** Wizualizacja kierunków ataków na aplikację

Każdą aplikację można w takim ujęciu zaatakować z czterech podstawowych kierunków, jak to pokazano to na rys. 7.1:

- System operacyjny – aplikację bardzo łatwo sparaliżować, odbierając jej w systemie operacyjnym uprawnienia, jakich wymaga do funkcjonowania w danym środowisku. Jeśli odbierzemy aplikacji na przykład prawo do zapisywania na dysku, a jest to podstawa przechowywania przez nią danych, to nie będzie ona w stanie poprawnie działać. Interesuje nas więc, czy nie pojawią się komunikaty lub też działania aplikacji, które mogą ujawnić przechowywane w niej dane, i czy po przywróceniu prawa do zapisu aplikacja będzie bezawaryjnie kontynuować swoją pracę.
- Platforma sprzętowa – w wielu przypadkach testowanie będzie zbliżone do testowania systemu operacyjnego, ale zamiast odbierać uprawnienia do posługiwania się sprzętem, będziemy go fizycznie usuwać. Możemy na przykład usunąć układ pamięci, na którym

operował dany system, i zbadać, czy oprogramowania zaadaptuje się do zmienionego środowiska i czy zacznie działać po przywróceniu układu.

- Interfejsy do zewnętrznych aplikacji (API) pozwalają na wiele testów zależnych od specyfiki danego API. Może to być oczywiście przesyłanie komunikatów nieoczekiwanych lub nieobsługiwanych. Może to być również zasypywanie API olbrzymią liczbą komunikatów. I tu znów chcemy się upewnić, czy aplikacja mimo „dociążonego” interfejsu jest w stanie działać, a po zaprzestaniu ataku – kontynuować pracę.
- Interfejs użytkownika (zazwyczaj GUI) – jest bramą dla wszelkiego typu ataków, głównie klasyfikowanych jako testy penetracyjne i bezpieczeństwa. Głównym celem będzie sprawdzenie, czy aplikacja nie udostępnia zastrzeżonych informacji osobom niepowołanym. Będzie to również weryfikacja, czy nieupoważniona osoba ma możliwość usunąć, dodać lub zmodyfikować dane lub funkcje.

Każdy z tych testów ma wymiar bardzo agresywny i ich przeprowadzenie musi być uzgodnione z innymi członkami zespołu i zaakceptowane przez kierownictwo.

#### **7.2.4. Podejście do testowania zależnie od dostępności specyfikacji**

W zależności od dostępności i jakości specyfikacji testowanie oprogramowania można przeprowadzić na wiele sposobów. Każdy z nich może być osobnym podejściem do testów lub też metodą testowania. Można po prostu usiąść i testować, można też sumiennie i systematycznie się do tego przygotować. Różne podejście do testów, zwłaszcza eksploracyjnych, można porównać do podróży przez ciągle budowane miasto. Jedną z ciekawszych form opisu „zwiedzenia aplikacji” jest heurystyka o nazwie FCC CUTS VID, opisana przez Michaela Kelly’ego.

Spróbujmy tę metaforę zastosować do opisu różnych typów testów. Założmy podróż z punktu A do B, przy założeniu, że A jest oddalone od B. Można na różne sposoby przemierzyć miasto, udając się do docelowego punktu, czyli nasza podróż może się znacząco różnić.



- Podróż z listą kroków. Wiedząc, że musimy się udać z punktu A do B, możemy, korzystając z różnych źródeł, map, wskazówek, czy po prostu opisać na kartce drogę, którą chcemy przebyć. Na przykład: ulicą X (50 m), w prawo w ulicę Y... Takie podejście w testowaniu możemy określić jako podróż z przygotowanymi własnoręcznie przypadkami testowymi.
- Podróż z nawigacją. Znajdując się w punkcie A, mamy przygotowaną przez inne osoby mapę dojścia do punktu B. Możemy więc mówić o testowaniu na podstawie scenariusza składającego się z gotowych przypadków testowych.
- Podróż tylko z celem. Wiemy tylko tyle, że jesteśmy w A, a chcemy się udać do B. Bez planu mamy jedynie ograniczoną wiedzę, gdzie jest B, a nie wiemy, ile czasu zajmie podróż. Nikt jednak nie zabroni nam korzystać z własnej wiedzy i doświadczenia. Wiemy, że rynek znajduje się w centralnym punkcie miasta, a centralne punkty zazwyczaj są dużo ciasniej zabudowane. Wiemy, że większość domów buduje się oknami na południe, i na podstawie analizy kilku z nich możemy określić północny kierunek. Na swojej drodze możemy dostrzec wiele wskazówek, a nawet spotkać ludzi, którzy wskażą nam drogę. W takim wypadku zbliżamy się w testowaniu do eksploracji. Przemierzając miasto bez przygotowania, w końcu dotrzemy (miejmy nadzieję) do punktu B.
- Podróż z punktami do odwiedzenia. Informacja, że na drodze między A i B można minąć most, przejść koło kościoła i zapewne przeciąć park, ułatwi nam dojście do celu, choć niekoniecznie najkrótszą drogą. Podobnie jest w wypadku testowania z użyciem list kontrolnych czy heurystyk. Ten typ nie będzie szerzej omawiany w niniejszej publikacji, gdyż nie jest to podejście często stosowane przez adeptów testowania.

## Heurystyka

Heurystyką zazwyczaj nazywamy zbiór prawd przekazanych nam przez jakiś autorytet. Na ich podstawie zakładamy poprawność zachowania się oprogramowania bez udowadniania, że tak rzeczywiście jest. Na przykład w raporcie o defektach możemy się powołać na konkretną heurystykę.

- Podróż z dobrą radą. Nasz kolega, który był w tym mieście, przeszedł już z A do B i opowiedział nam o tym. Wie, jak osiągnąć B

– niekoniecznie w najkrótszym czasie, ale drogą, którą sam pokonuje (np. najmniej męczącą). Takie przejście z rekomendacjami możemy określić jako testowanie z przypadkami użycia: pojedyncza osoba lub też grupa osób opisuje, jak chciałaby używać lub używa aplikacji, a my na tej podstawie ją testujemy. Również ta technika, ze względu na niezbyt często spotykaną formę zapisu wymagań, zostanie w książce pominięta.

- Podróż „na głupa”. A jeśli po prostu wejdziemy do miasta i będziemy ślepo przemierzali jego uliczki? Bez zastanawiania się, gdzie idziemy i dlaczego? A jeśli naszym celem jest B, ale nawet nie do końca wiemy, jak to B wygląda, czym jest i co się w nim znajduje? W takim razie, nawet dochodząc do B, możemy je przegapić. Jest jednak szansa, że się tam znajdziemy. Takie podejście nazywamy testowaniem *ad hoc* – nieprzygotowanym i niesformalizowanym, niewymagającym zbyt wiele od testera.
- Podróż z wiedzą. Odwiedziliśmy już to miejsce wcześniej. Znamy je, bo spędziliśmy dużo czasu na przemierzaniu jego ulic, ale nie było nas tu parę lat. Kilka budynków wyburzono, kilka nowych postaviono. Część miasta popadła w ruinę, a inna ożyła. Pewne rzeczy pozostały jednak po staremu, tak jak je pamiętamy. Jeśli ktoś mówi: „Idź z A do B”, a oczami wyobraźni widzimy B, to możemy szybko obmyślić plan dotarcia do B. Może najkrótszą drogą, a może odwiedzając miejsca, które się zmieniły, by to zobaczyć. A może tego konkretnego miasta nigdy nie odwiedziliśmy, ale jest podobne do wielu innych, budowanych w tym samym stylu. Znając jedno, bez problemu odnajdziemy się w drugim. Takie przejście będzie nazywane testowaniem na podstawie wiedzy, często określanej „domenową”. Wiedza domenowa pozwala nam łatwiej i szybciej testować aplikację.

Każde z tych podejść jest unikatowe i każde z nich może być uzupełnieniem innych. Mówimy o komplementarności podejść do testów, które zwiększają naszą zdolność do weryfikacji oprogramowania. Stosowanie eksploracji i przypadków testowych zwiększy zakres pokrycia oprogramowania i może ujawnić więcej defektów. Porzucając metafory, spróbujemy zdefiniować część opisanych tu testów.

Zrozumienie działania oprogramowania wiąże się nierozłącznie ze znalezieniem odnośnika, który pomoże nam jednoznacznie wskazać,

czy program działa poprawnie, czy też nie. Nazywa się to wyrocznią testową.

### Czym jest wyrocznia?

Co prawda, wyrocznia to pojęcie żywcem wyjęte z ISTQB, ale szczególnie przypadło mi do gustu i żałuję, że nie przyjęło się w świecie testerów. Wyrocznia jest pojęciem bardzo szerokim i zakłada, że istnieje źródło wiedzy na temat poprawnego zachowania się oprogramowania. Najbardziej oczywistą wyrocznią będzie specyfikacja, potem (istotny) użytkownik i każde inne źródło, które powie, czy coś działa, czy też nie.

A oto przykładowe wyrocznie w testowaniu wymagalności pola i przekazania danych w formularzu:

1. Możesz się odwołać do specyfikacji lub istotnego użytkownika.
2. Możesz sprawdzić samodzielnie, klikając Wyślij – powinny się pojawić komunikaty.
3. Możesz się odwołać do kodu. Przy wymaganych polach powinien pojawić się przykładowy kod: `class="validate['required']"`, który oznacza wymagalność pola. Czy możesz zaufać takiemu sprawdzeniu? W tym wypadku musisz zaufać programiście, że dobrze to zaimplementował.

Testowanie oprogramowania jest niezmiernie łatwe, jeśli wiesz, od czego zacząć. Dalsze kroki zależą od otoczenia, w jakim jesteś, i materiałów, jakie otrzymasz. Jeśli nie masz wyroczni, musisz ją zdobyć i to opisuje kolejny podrozdział. Jeśli jednak masz wyrocznię w postaci specyfikacji wymagań, to twoje zadanie będzie tylko odrobinę prostsze.

**Testowanie bez specyfikacji.** Wyobraź sobie, że dostajesz jedynie aplikację, bez specyfikacji opisującej jej działanie. Twoim orężem są:

1. Logiczne myślenie – choć może być zawodne, bo twoja logika wcale nie musi być spójna z logiką biznesu. Możliwe również, że logiczne myślenie nie jest twoją mocną stroną. Czy wtedy testowanie nie jest zawodem dla ciebie? Niekoniecznie. W testowaniu są również zadania bardziej mechaniczne, powtarzalne, gdzie inni tworzą przypadki testowe, a osoby pełniące funkcję (zazwyczaj) operatora

- je wykonują. Musisz jednak wiedzieć, że brak umiejętności logicznego myślenia znacznie zawęzi twoją umiejętność testowania.
2. Doświadczenie wyniesione z używania takich lub podobnych aplikacji w przeszłości. Jeśli ci go brakuje, to okaże się to kłopotliwe oraz problematyczne w dłuższym okresie. W tym przypadku musisz polegać na doświadczeniu innych ludzi. Twoim narzędziem pracy może się stać lista kontrolna z gotowymi pomysłami na testy.
  3. Wiedza o tym, kiedy i jakie błędy może popełnić programista, wiedza o konstrukcji oprogramowania, wiedza o typowych defektach oprogramowania, wiedza wszelaka. Tej wiedzy nigdy za mało, ile byś jej nie posiadał zawsze możesz ją poszerzyć, czytając książki, wiadomości, blogi i ciekawostki. Możesz rozmawiać z innymi testerami, programistami i analitykami. Każdy z nich zna dziesiątki anegdot, jak to kiedyś ktoś przypadkowo nacisnął format `c:` i `Enter` i była kupa śmiechu.
  4. Inni ludzie – zawsze znajdzie się ktoś, kogo można zapytać, jak działa dana aplikacja. Nie nadużywaj jednak jego cierpliwości i dobrze przygotuj swoją listę pytań. Inni ludzie mają swoje zadania i obowiązki i zabierasz im cenny czas. Z wyprzedzeniem zadaj sobie pytania, jakie chcesz zadać im, a nuż okaże się, że jednak znasz na nie odpowiedź. Może wystarczy pliki pomocy samej aplikacji? A może wyszukiwarka internetowa? Nie bój się eksperymentować z aplikacją. Czasami samo zachowanie aplikacji powie ci, czy działa ona tak, czy inaczej. Nie musi to być zachowanie poprawne, ale zawsze stanowi podpowiedź. Jeśli dopiero startujesz w zawodzie, to zlecający ci to zadanie powinien przewidzieć, że możesz coś zepsuć. Zapewne jest jakaś kopia zapasowa albo jesteś w środowisku testowym, albo masz ograniczone uprawnienia, albo twoje akcje nie mają większego znaczenia. Jeśli z jakiegoś powodu tak nie jest, to znaczy, że zadanie dostałeś od krety i należy mu się mała lekcja. Ty zepsujesz, ale to on będzie winny.

Otrzymując aplikację, zazwyczaj od razu masz świadomość jej otoczenia. Dostajesz ją razem ze środowiskiem. Widząc ją na ekranie komputera, szybko ocenisz, czy jest to aplikacja desktopowa, czy też internetowa. Widząc tylko ekran, możesz prawie natychmiast rozpoznać środowisko działania aplikacji. Może to być równie dobrze aplikacja mobilna, dostępny jest jedynie ekran dotykowy. Może to być aplikacja wyświetlana na ekranie telewizora, gdzie nie ma dostępu do

pełnej klawiatury. Może to być również oprogramowanie wbudowane, z małym ekranem i kilkoma klawiszami do obsługi. Środowisko samo w sobie warunkuje już pewną pulę testów. Specyfikacja może więc być częściowo zastąpiona przez uniwersalne weryfikatory dla danej platformy.

**Testowanie *ad hoc*.** W wypadku testowania nieformalnego i nieprzygotowanego nie można mówić o strategii testowania. Jest to zwykła próba szybkiego sprawdzenia działania/niedziałania oprogramowania. Weryfikacja polega na prześlizgnięciu się po aplikacji bez poważniejszego zagłębiania się w poszczególne jej obszary. Dlaczego więc o niej mówimy? Dlatego, że stanowi pierwszy krok do profesjonalnego testowania. Zgodnie ze starym powiedzeniem Lao-tsy, *Nawet najdłuższa droga zaczyna się od pierwszego kroku*. Ponadto nieformalne testowanie jest uzupełnieniem technik bardziej systematycznych. Mówimy wtedy o tzw. wolnym testowaniu (*free test*). Tam, gdzie uruchomiono już wszystkie przypadki testowe i gdzie ciągle pozostają wątpliwości co do jakości oprogramowania, zawsze możemy spróbować przeanalizować aplikację, nie opierając się na dokumentacji wymagań, przypadkach czy podręcznikach użytkownika. Możemy w ten sposób zidentyfikować obszary, które z jakiegoś powodu nie zostały jeszcze „dotknięte” testerską dłonią. Testowanie *ad hoc* zazwyczaj kojarzy się z tzw. małym testowaniem, czyli ślepym klikaniem, i w takiej formie może być uznane wyłącznie za pierwszy z etapów uczenia się aplikacji.

**Testowanie eksploracyjne.** Prawdopodobnie żadna inna technika, metoda, podejście czy też grupa testów nie wzbudzą takich emocji, jak testowanie eksploracyjne. Przyjęło się je nazywać techniką testowania oprogramowania, ale jest to za wąskie pojęcie, aby w pełni oddać wiążące się z nim implikacje dla testów. Pozostańmy więc przy określeniu TE (*exploratory testing* – ET) jako „podejścia”. Jest to najbliższe propozycjom twórców oraz krzewicieli testowania eksploracyjnego.

Nie ma jednej dobrej definicji TE. Testerzy formalni mówią o „nieformalnej technice projektowania testów, w której tester projektuje testy w czasie, gdy są one wykonywane, i wykorzystuje informacje zdobyte podczas testowania do projektowania lepszych

testów” (za Słownikiem wyrażen̄ związanym z testowaniem). Niekonsekwencję i deprecjonowanie wiadać juŹ w pierwszych słowach tej definicji. Skoro jest to technika projektowania testów, to dla czego nieformalna, skoro samo projektowanie testów jest formalne? Jak tester moŹe wykonywać testy, skoro nie zostały jeszcze napisane? Definicja twórców TE nie jest jednoznaczna, gdyŹ nie ma jednej uznanej szkoły tego podejścia. Możemy jednak w różnych definicjach wyróżnić wiele wspólnych mianowników. Specjaliści tego podejścia w swoich wystąpieniach i publikacjach mówią o TE jako:

- uczeniu się, projektowaniu, wykonaniu i interpretacji (Cem Kaner),
- symultanicznym uczeniu się, projektowaniu testów i ich wykonaniu (James Bach),
- testowaniu bez predefiniowanego planu testów (James Whittaker).

W dużym uproszczeniu o testowaniu eksploracyjnym mówimy wtedy, gdy przypadki testowe (jeśli w ogóle powstają) są wytwarzane po uruchomieniu i zbadaniu wycinka oprogramowania. Podejście ma na celu znalezienie możliwie największej liczby awarii w możliwie najkrótszym czasie. Próbuje więc obniŹać koszty testowania poprzez podnoszenie efektywności działania. Nastawiamy się bardziej na osiągnięcie rezultatów i szybkich korzyści (*quick wins*) niŹ na sformalizowane i procesowe zarządzanie kontrolą jakości. W tab.7.1 zestawiono i skonfrontowano formalną definicję z jej interpretacją nieformalną.

Eksploracja ma wiele zalet, które można wykorzystać, by zwiększyć skuteczność testów. Podstawą do formalnych testów jest formalna lub mniej formalna specyfikacja wymagań. TE stosujemy, gdy musimy sobie poradzić z niepełnymi lub nieudokumentowanymi wymaganiami. Jak wiadomo, jest to częsty kłopot większości projektów informatycznych.

**Tab. 7.1.** Formalna definicja testowania vs. interpretacja nieformalna

Formaliści testowania	Praktycy TE
Nieformalna ...	Unikanie formalizmów nie powoduje, Źe technika jest nieformalna. Testowanie eksploracyjne doczekało się wielu rozszerzeń, które formalizują wykonanie testów, pozostawiając źródłową technikę braku

	udokumentowania przypadków testowych przed przystąpieniem do testowania
... technika ...	Podejście lub strategia
... projektowania testów ...	Nie mówi się o projektowaniu testów, a raczej o definiowaniu celów osiągnięcia pewnej misji. Definicja misji pokrewna jest pośredniemu celowi do zrealizowania w oprogramowaniu i często nazywana ideą testową
... w której tester ...	Definicja słowa „tester” po raz kolejny znacznie zawęża samo TE. Praktycy eksploracji pytają: „Dlaczego tylko tester?”. Przecież to podejście może stosować każda osoba zaangażowana w tworzenie oprogramowania. Formaliści mówią jeszcze o „doświadczonym testerze”, podczas gdy praktycy pytają: „Dlaczego tylko doświadczony? Czy tylko osoby, które umieją chodzić, mogą chodzić?”. Dziecko, które dopiero się urodziło, ma oczywiście niewielkie szanse na zwycięstwo w chodzie sportowym, ale jego próby są formą uczenia się. Podsumowując: doświadczony tester osiągnie więcej, ale technikę eksploracyjną jako naturalną metodę może stosować każdy
... projektuje testy ...	Projektowanie testów ma tutaj odrębny wymiar i nie pokrywa się z definicją teoretyków. Praktycy interpretują projektowanie testów jako opis działań testera potrzebny do odpowiedniego zdefiniowania końcowego raportu z prac
... w czasie, gdy są one wykonywane...	Wykonywanie testowania jest wyzwalaczem opisu, a nie odwrotnie
... wykorzystuje informacje zdobyte podczas testowania ...	Tutaj „wykorzystanie informacji” zostaje zastąpione określeniem „uczenie”, „poznawanie”, „analizowanie” i ostatecznie „eksplorowanie”
... do projektowania nowych i lepszych testów	Nie projektuje się fizycznie większej liczby testów, a raczej szuka się nowych celów do osiągnięcia, nowych idei do realizacji i odkrywa nowe obszary do analizowania i szukania defektów

Eksploracja ma wady, które punktują zwolennicy formalizacji testów, między innymi brak formalnych miar pokrycia. Odpowiedzią na ten zarzut jest zarządzanie testowaniem oparte na sesjach, gdzie cele

i czas trwania sesji pozwalają zbierać konkretne wartości liczbowe. Są to miary inne niż pokrycie wymagań czy ryzyk i analizowane są w innym wymiarze, ale to nie znaczy, że mniej skutecznie podsumowują jakość oprogramowania.

Świat testerów podzielił się na dwie grupy. Najbardziej ortodoksyjni zwolennicy testowania eksploracyjnego traktują je jako jedyną dopuszczalną i ekonomicznie uzasadnioną metodę weryfikacji oprogramowania. Na drugim biegunie mamy grupę miłośników formalizmów testowych, którzy do testowania eksploracyjnego odnoszą się z politowaniem, nazywając je „marnym substytutem prawdziwego testowania”. TE przypięto również łatkę testowania *ad hoc* i w gruncie rzeczy nieprzygotowanego. Zwolennicy TE wskazują jednak dziesiątki metod i narzędzi zarządczych, które zmieniają tę prostą metodę weryfikacji w potężne narzędzie kontrolowania jakości.

Nie ma potrzeby stawania po którejkolwiek ze stron sporu, bo mówimy raczej o akademickich dywagacjach niż faktach. Decyzję, jakie testowanie wykonać, zazwyczaj wymusza otoczenie działania i budżet przeznaczony na sprawdzenie produktu.

### **Ostateczny dowód?**

Teoria i praktyka ostro się ścierają na polu testowania eksploracyjnego, ale ostatecznym dowodem będą dwa proste eksperymenty.

- Doświadczonemu testerowi dajesz oprogramowanie i mówisz, że ma X czasu (30 minut, godzinę, dzień), aby przekazać swoją opinię o przygotowaniu tego oprogramowania do wydania klientowi. Czy zacznie pisać przypadki testowe, czy rozpocznie eksplorację?
- Doświadczonemu testerowi dajesz oprogramowanie i mówisz, że ma X czasu (30 minut, godzinę, dzień), aby wskazać możliwie najwięcej defektów w oprogramowaniu. Czy zacznie pisać przypadki testowe, czy rozpocznie eksplorację?

Z jego odpowiedzi powinno wynikać, jakie testowanie jest najbardziej wskazane, gdy brakuje czasu.



A hand-drawn sketch of a web form titled "Utwórz nowego użytkownika" (Create new user) in the top header, which also contains a close button 'X'. The form has three input fields: "Login:", "Hasło:", and "Powtórz hasło:". Below these fields is a button labeled "Utwórz".

**Rys. 7.2.** Testowanie bez specyfikacji, przykład interfejsu

**Przemyślane testowanie bez specyfikacji.** Dla przeprowadzonej tu analizy przyjmiemy najprostszy przypadek, kiedy aplikacja wymuszająca utworzenie nowego użytkownika dostępna jest na komputerze, jak pokazano na rys. 7.2.

Opierając się na pozyskanych informacjach, zastanów się, czego jako użytkownik się spodziewasz:

1. Podając poprawny login i dwa razy to samo poprawne hasło, spodziewasz się utworzenia konta.
2. To już samo w sobie rodzi pytania, czym jest poprawny login i hasło, czyli jakie reguły walidacji muszą zostać spełnione, jakie są wytyczne co do długości ciągu znaków itd.
3. Możesz założyć z dużą dozą prawdopodobieństwa, że w systemie nie może być dwóch użytkowników o tym samym loginie. Z perspektywy bazy danych tylko login ich wyróżnia. Chyba że użytkownikom nadawany jest automatycznie identyfikator, który pozwala ich rozpoznawać. Takie rozwiązanie jest mało prawdopodobne, ale możliwe. Łatwo to sprawdzić, próbując dwa razy zarejestrować użytkownika o tym samym loginie.
4. Za polem Login kryje się logika i walidacja. Jest to spełnienie kryterium długości ciągów znaków (w tym dopuszczalnych).

5. Za polem Hasło kryje się logika i walidacja. Jest to spełnienie kryterium długości ciągów znaków (w tym dopuszczalnych).
6. Za polem Powtórz Hasło kryje się logika i walidacja. Jest spełnione, kiedy hasło powtórzone jest takie samo jak podane wcześniej.
7. Walidacja pól może być realizowana w momencie wprowadzania wartości do pola, jego opuszczenia lub dopiero kiedy naciśnięty zostanie przycisk Utwórz.
8. Przycisk Utwórz uruchamia walidację pól, sprawdzenie istnienia użytkownika w bazie, próbę utworzenia użytkownika i kilka rzeczy, które niekoniecznie muszą być widoczne w interfejsie użytkownika końcowego.
9. Po naciśnięciu przycisku Utwórz spodziewamy się (przy poprawnych danych) utworzenia użytkownika, a dla niepoprawnych danych komunikatu czy komunikatów niepoprawności.

Oprócz tego można przeanalizować jeszcze kilka aspektów:

1. Czy można utworzyć użytkownika bez loginu i/lub hasła?
2. Czy aplikacja rozpozna, że do pól wprowadziło się spację, zero albo null?
3. Czy aplikacja będzie próbowała wykonywać polecenia, które wprowadzi się do pola (bezpieczeństwo)?
4. Czy w aplikacji działa Tab do przemieszczania się między polami?
5. Czy istnieje inna możliwość skrótów klawiaturowych?
6. Jak jest możliwość wprowadzania danych do pól? Czy tylko klawiatura fizyczna? A może klawiatura ekranowa?

Aspektów do przeanalizowania jest więcej i każdy ma swoją wartość albo wiąże się z nim przeprowadzenie testu. Nie trzeba jednak testować od razu wszystkiego. Można skupić się na ważnym wycinku, zostawiając resztę na przyszłość (jeśli starczy czasu). Otrzymując aplikację do przetestowania, zazwyczaj otrzymujemy również skończony czas na testowanie. Nie możemy więc zużyć go na pierwszą funkcję. Musimy go logicznie rozplanować na wszystkie funkcje, nadając im priorytety. Czym się kierować? Logiką, doświadczeniem, wiedzą, opinią innych.

Poszczególnym testom można przypisać priorytety, np. utworzyć skalę 1–5, uznając, że:

- 1 – musi zostać uruchomiony,
- 2 – powinien zostać uruchomiony,
- 3 – warto by go uruchomić,
- 4 – zostanie uruchomiony, jeśli starczy czasu,
- 5 – do uruchomienia tylko jak będzie dodatkowy czas.

Moje priorytety oparte na wiedzy i doświadczeniu zawarłem w tab. 7.2.

**Tab. 7.2. Priorytety autora**

Test	Priorytet autora	Twój priorytet
Poprawne założenie konta użytkownika	1	
Próba założenia konta użytkownika przy niepoprawnych danych – zły login	1	
Próba założenia konta użytkownika przy niepoprawnych danych – złe hasło	1	
Próba założenia konta użytkownika przy niepoprawnych danych – różne wpisy w polach Hasło i Powtórz hasło	2	
Poprawność reguł walidacji	1	
Unikatowość loginu	2	
Przycisk << Powrót	2	
Dodatkowe akcje po przyciśnięciu Utwórz	3	
Czy można utworzyć użytkownika bez loginu i/lub hasła?	2	
Znaki specjalne spacja, zero, null	4	
Nawigacja po polach i skróty klawiaturowe	2	

Możesz je skonfrontować ze swoimi i nic dziwnego, że będą różne. Dlaczego? Dlatego, że myślimy inaczej, mamy różne doświadczenia i wiedzę. Nie twierdzę, że moja perspektywa jest lepsza od twojej, liczę również, że nie uznasz, iż twoja jest poprawniejsza. Kiedy otrzymujemy możliwość interpretacji, to znaczy, że ktoś liczy się z naszym zdaniem albo daje nam wolną rękę w wyborze testów. Warto z niej skorzystać.

Więcej o przykładowych testach piszę dalej w tej książce.

**Testowanie ze specyfikacją przypadków testowych, czyli testowanie systematyczne.** Testowanie systematyczne to przede wszystkim testy oparte na różnego rodzaju dokumentach w projekcie. W sformalizowanych projektach testowania mamy przypadki testowe. Czym są przypadki? Każde liczące się źródło ma własną definicję przypadku testowego. Nie możemy uznać, że któraś z nich jest bardziej lub mniej poprawna, ale możemy powiedzieć, że niektóre z nich są bardziej poprawne od innych w danych okolicznościach.

Przypadek testowy to:

- zestaw danych wejściowych, warunków wykonania i spodziewanych rezultatów, przygotowanych w celu przetestowania programu lub weryfikacji zgodności z wymaganiami – wg IEEE Std 610-1990;
- dokumentacja określająca dane wejściowe, spodziewane rezultaty oraz warunki wykonania testu – wg IEEE Std 829-1983;
- szczególne dane wejściowe, których używamy, oraz procedury, których przestrzegamy, testując oprogramowanie – wg Rona Pattona;
- pytanie, które zadajemy programowi, aby uzyskać informację, sformułowane na wiele sposobów w zależności od tego, jaką informację chcemy uzyskać – wg Cema Kanera.

Bez względu na definicję każdy przypadek, jak każde działanie, musi mieć swój określony cel, dla którego go uruchamiamy. Może nim być:

- znalezienie defektów,
- wstrzymywanie niedojrzałych wersji systemu,
- wsparcie szefów projektu w podjęciu decyzji o wstrzymaniu lub wprowadzeniu systemu przez dostarczenie informacji o jakości oprogramowania,
- minimalizacja kosztów wsparcia technicznego przez eliminację defektów przed przekazaniem oprogramowania do produkcji,
- ocena zgodności ze specyfikacją,
- sprawdzenie dostosowania oprogramowania do przepisów,
- minimalizacja ryzyka związanego z niepoprawnością działania funkcji lub parametrów niefunkcjonalnych,
- znalezienie bezpiecznych scenariuszy użytkowania systemu,
- ogólna ocena jakości,

- weryfikacja poprawności działania systemu.

Pomijając fakt, że formalizmy nie są rozpowszechnione w polskich projektach informatycznych, to w wielu z nich spotyka się przypadki testowe niskiej jakości albo niewnoszące niczego wartościowego do projektu. Należy zawsze szukać dobrych wzorców przypadku testowego i na ich podstawie stworzyć własny wzorzec, dopasowany do typu oprogramowania i projektu.

Przypadek testowy przypadkowi testowemu nierówny. Zawsze będą odmienne, w zależności od organizacji, poziomu testów i metod wytwarzania oprogramowania. Pokazujemy tu jeden ze wzorców. W normie IEEE Std 610-1990 na przypadek składa się wiele elementów. Najważniejszy jest cel, który chcemy osiągnąć. Kolejne to:

- elementy **podstawowe**:
  - ID – unikatowy identyfikator,
  - warunki wstępne uruchomienia przypadku testowego,
  - kroki do wykonania testu,
  - oczekiwany rezultat,
  - warunek końcowy;
- elementy **ważne**:
  - tytuł,
  - dane testowe,
  - środowisko uruchomienia,
  - powiązanie z wymaganiem (śledzenie testowe),
  - kategoria/typ,
  - tagi – słowa kluczowe powiązane z przypadkiem,
  - autor (wraz z danymi kontaktowymi),
  - informacje o wersji,
  - wiele innych, ponieważ przypadek testowy może być nośnikiem olbrzymiej ilości informacji.

Wzorzec ten posłuży nam do zdefiniowania przypadku testowego. Do powstania przypadku zazwyczaj potrzebujemy wymagania dotyczącego oprogramowania. Zdefiniujmy sobie więc prostą funkcję logowania: **Do aplikacji internetowej można się zalogować, podając istniejący (poprawny) login i hasło.**

Szczegółowy przypadek testowy z danymi testowymi mógłby wyglądać następująco:

## ID 1.1

Tytuł: Poprawne zalogowanie się do aplikacji

Środowisko: przeglądarka FireFox X.Y., system operacyjny W8, SP 1

Warunek wstępny: włączona przeglądarka na stronie <http://...../user/login/>; użytkownik nie jest zalogowany

Kroki do wykonania:

- Podaj istniejący login: jkowalski.
- Podaj istniejące i przypisane do loginu hasło: 2012.
- Naciśnij klawisz Zaloguj.

Oczekiwany rezultat: zalogowanie się jako użytkownik jkowalski z przekierowaniem na stronę główną.

Warunek końcowy: użytkownik jest zalogowany.

Opisany tu przypadek jest przypadkiem pozytywnym, ponieważ weryfikuje podstawową funkcjonalność. Przypadek testowy negatywny może być weryfikacją komunikatów podczas nieudanej próby logowania się.

Przypadek testowy może być jednak znacznie bardziej ogólny.

## ID 1.1

Środowisko: obsługiwana przeglądarka

Kroki do wykonania: podaj login i hasło

Oczekiwany rezultat:

Poprawny login/hasło – zalogowany

Niepoprawny login/hasło – niezalogowany

Pierwszy przypadek jest na tyle szczegółowy, że nawet niedoświadczona osoba może go uruchomić. Jego napisanie wymaga za to dużo więcej czasu. Nazywamy go przypadkiem testowym niskiego poziomu. Drugi przykład to przypadek wysokiego poziomu. Jest oczywiście mniej pracochłonny w wytworzeniu, natomiast wymaga więcej wiedzy i doświadczenia od osoby go uruchamiającej. Szczegółowość przypadków testowych jest zawsze pewnym balansem

między kosztami jego wytworzenia i późniejszego utrzymania a wiedzą, jaką muszą posiadać testerzy, aby móc go uruchomić. Wiele organizacji boryka się z problemem szczegółowości przypadków testowych. Czy warto opisywać je z dokładnością do pojedynczej danej? A może warto zostawić przestrzeń do wolnego testowania?

W poszukiwaniu różnych bytów testowych można wyróżnić dwa dodatkowe elementy, które wspomagają notację w świecie testowania. Są to:

- Lista kontrolna (za Wikipedią, z poprawkami uwzględniającymi testowanie):
  1. Wykaz czynności (zwykle z możliwością zaznaczenia opcji TAK/NIE) przygotowywany dla skomplikowanych zadań w celu zapewnienia właściwej (optymalnej) kolejności i nie pominięcia żadnego istotnego etapu testowania.
  2. Wykaz czynności kontrolnych dla porównania stanu istniejącego ze stanem wzorcowym, opisanym tą listą, często z oceną stopnia zgodności.
- Idea testowa to pojęcie zaczerpnięte z testów eksploracyjnych, w których przez eksplorację chcemy osiągnąć jakiś cel. Cel ten zdefiniowany jest jako atomowa idea, czyli pomysł na test. Jest to pomysł na weryfikację systemu przez sformułowanie krótkiego opisu tego, co będziemy chcieli osiągnąć podczas testowania. Czasami mówi się o opisie, który nie może być dłuższy niż 140 znaków.

Próbując opisać relacje między nimi, możemy zobrazować je przez wyróżnienie formalizmów oraz dokładności opisów, tak jak pokazano na rys. 7.3.

Organizacje minimalizujące formalizmy i stawiające na testowanie przez doświadczonych testerów mogą pozwolić sobie na testy eksploracyjne oparte na ideach testowych. Mogą one również stosować przypadki testowe wysokiego poziomu (nie jest to jednak standard).

Testowanie oparte na przypadkach testowych niskiego poziomu skuteczne będzie tam, gdzie istnieje grupa doświadczonych testerów tworzących przypadki testowe dla mniej doświadczonych kolegów. Przypadek taki nie tylko służy bardziej precyzyjnemu raportowaniu, ale również jest formą przekazywania wiedzy.

Opcja pośrednia będąca listą kontrolną, bardziej szczegółową i bardziej

Opcją pośrednią będą listy kontrolne, bardziej szczegółowe i bardziej formalne od idei testowych, ale niedające szczegółowego opisu poszczególnych działań.



**Rys. 7.3.** Wizualizacja różnych produktów testowania w zależności od dokładności i formalizmu zapisu

Odwołując się do naszego wcześniejszego wymagania z logowaniem się do aplikacji idea testowa mogłaby brzmieć: **Przetestuj logowanie się do aplikacji.**

Z kolei lista kontrolna zawierać może dwie proste propozycje:

- **Użytkownik z poprawnym loginem i hasłem może się zalogować**
- **Użytkownik z poprawnym loginem i niepoprawnym hasłem nie może się zalogować.**

W dalszej części pokazane są podtypy testów formalnych lub systematycznych – w zależności od źródła wiedzy stanowiącego podstawę do wytworzenia przypadku testowego.

**Testowanie ze specyfikacją – Testowanie oparte na kompletnej dokumentacji.** Testować można, opierając się na wielu dokumentach różnego rodzaju. Zazwyczaj spisany dokument jest podstawą do



wytworzenia przypadków testowych czarnoskrzynkowych. Uszczegółowienie techniczne dokumentów wytwarzanych przez biznes stanowi podstawę do wytworzenia przypadków używanych na poziomie białej skrzynki. Główny podział wynika ze źródła wiedzy o zachowaniu aplikacji. Wyróżniamy następujące typy testów:

1. Testy oparte na wymaganiach. Przypadki testowe powstają na podstawie informacji zawartych w wymaganiach użytkowników względem oprogramowania. Mówimy tu głównie o testach funkcji i charakterystykach (niefunkcjonalności) opisanych w specyfikacji. Dokument wymagań zazwyczaj stworzony jest w języku zrozumiałym zarówno dla zamawiającego, jak i wytwarzającego oprogramowanie, przeważnie w języku naturalnym.
2. Testowanie oparte na procesie biznesowym. Oprogramowanie biznesowe zawsze powstaje jako narzędzie wsparcia dla mniej lub bardziej formalnego procesu danego biznesu. Sam proces może więc stać się specyfikacją opisującą działanie aplikacji. Wówczas przypadki testowe będą weryfikowały zgodność oprogramowania z procesem.
3. Testowanie oparte na przypadkach użycia, które są inną formą zapisu wymagań. Przypadki użycia są łatwym narzędziem komunikacji między użytkownikiem, który w oprogramowaniu ma pewien cel, a programistą, który musi tak przygotować oprogramowanie, aby użytkownik mógł ten cel osiągnąć. Przypadek użycia przyjmuje zazwyczaj formę scenariusza, w którym użytkownik krok po kroku opisuje swoją pracę. Opierając się na takiej procedurze pracy, można łatwo wytworzyć przypadki testowe służące do weryfikacji poprawności działania oprogramowania.
4. Testowanie oparte na projekcie (*design*) jest formą testowania, w którym podstawą do wytworzenia przypadków testowych jest specyfikacja systemowa i architektura. Jest to szczegółowy opis działania systemu skierowany do osób z wykształceniem technicznym, dlatego na podstawie takiego dokumentu powstają przede wszystkim przypadki białoskrzynkowe.

### **Projekt, czyli nieprojekt**

Nie jest tajemnicą, że informatyka nie miała łatwo w czasach PRL-owskich, a i inne cele rozwojowe lat 80. i 90. ubiegłego wieku przyćmiły rozwój polskiego słownictwa informatycznego. Dziś przychodzi nam

ponieść za to surową karę. Polski słownik informatyczny to głównie zapożyczenia z języka angielskiego. Co więc mamy zrobić z nieszczęsnym *design* tłumaczonym jako „projekt” i *project* tłumaczonym jako „projekt”? Kiedy mówimy „dokumentacja projektowa”, mamy na myśli dokumenty stworzone przez kierownika projektu czy też przez projektanta systemu?

Oczywiście istnieją również testy oparte na innych dokumentach projektowych. Dobry tester oprogramowania wykorzysta każdy fragment tekstu jako podstawę do opracowania testu. Jest to oczywiście nierekomendowane, ale czasami przypadki testowe pisze się, wykorzystując e-maile od użytkowników czy interpretację defektu przez programistę. Jest to metoda rozpaczliwa, ale w większości projektów pozbawionych specyfikacji – skuteczna. Musimy pamiętać, że przypadek testowy zawsze będzie tak dobry, jak dobrze przygotowany jest dokument źródłowy.

**Testowanie ze szczątkową specyfikacją.** Na koniec zostawiam ten element, który spotykałem w większości swoich projektów. Specyfikacja istnieje, ale jest przestarzała i w niewielkim zakresie pokrywa obecną funkcjonalność. Będzie utrudniała zadanie doświadczonym członkom projektu i stanie się wstępnym materiałem dla tych, którzy aplikacji mają się sami nauczyć od podstaw. Do studiowania dokumentu należy podejść jak do czytania relacji dawnych kronikarzy w poszukiwaniu prawdy historycznej. Wiadomo, że część zapisów nie jest prawdą, a jedynie subiektywną opinią autora o tym, co się wydarzyło lub powinno się wydarzyć. Część zapisów nie będzie aktualna, bo znając efekty przeszłych działań, wiemy, że część historii potoczyła się zupełnie inaczej itd. Nie możemy wierzyć w ani jedno słowo w dokumencie, ale opierając się na nim, możemy sobie wyrobić opinię na temat ówczesnego postrzegania kształtu i funkcjonalności oprogramowania. Nie mogło ono zejść aż tak bardzo z pierwotnie obranego kursu i ważne funkcje pozostały relatywnie niezmiennie. Przecież jest mało prawdopodobne, że próbujemy budować system zarządzania relacjami z klientem, a skończyliśmy na systemie zarządzania produkcją.

Dokument ma nam dać jedynie wstępne poczucie misji i celu oprogramowania.

**Misja (cel)**

Korporacje zepsuły postrzeganie słowa „misja” (cel). Kiedyś, posługując się pojęciem misji, mówiliśmy o rzeczach wielkich, np. o wyprawie na Księżyc. Dziś misja wpisywana i powielana dziesiątki razy na korporacyjnych stronach kojarzy nam się jednoznacznie negatywie. Producent butów pisze „Naszym celem jest dostarczenie naszym klientom wysokiej jakości produktu w cenie adekwatnej do jakości”. Brzmi to ładnie, ale średnio rozgarniętemu odbiorcy takiego komunikatu przed oczami staną źle opłacani azjatyccy pracownicy, którzy pracują w warunkach niebezpiecznych dla zdrowia za stawki urągające ludzkiej godności. Wyprodukowanie buta kosztuje 2 euro, w sklepie trzeba zapłacić za niego 50 razy więcej, a budżet w znaczącej części przeznaczany jest na marketing i zyski dla akcjonariuszy. Czy to jest ta „misja”?

Innymi dostępnymi w projekcie dokumentami mogą być mniej lub bardziej sformalizowane opisy zarządzania projektem, dokumenty architektury czy konstrukcji oprogramowania. Mogą to być również materiały marketingowe albo wskazanie produktu podobnego do tego, na którym się wzorujemy. Każdy z nich na swój sposób może być dla nas interesujący i stanowić źródło wiedzy o aktualnej funkcjonalności oprogramowania. Postaramy się na ich podstawie wysunąć możliwie najwięcej pomysłów na weryfikację oprogramowania.

Zastosowane przez nas metody będą połączeniem testowania bez specyfikacji tam, gdzie nasz szcątkowy opis nie sięga, i testowaniem ze specyfikacją tam, gdzie otrzymujemy kompletną (lub jakąś) informację.

### **7.2.5. Testowanie oparte na ryzyku**

Z jednej strony, testowanie oparte na ryzyku jest jedną z częściej opisywanych strategii testowania i technik projektowania testów, z drugiej strony – ma niezbyt duże zastosowanie na początkowych etapach rozwoju testera. Wspominam o niej, ale jej szczegółowe rozwinięcie pozostawiam innym publikacjom na temat testowania. Testowanie oparte na ryzyku ma obniżyć poziom ryzyka zidentyfikowanego dla naszego oprogramowania. Chodzi zarówno o ryzyko samego oprogramowania (poprawność jego działania), jak i użycia go w jego środowisku. Aby zaprojektować testy, należy na

początku zidentyfikować ryzyko, bo niezidentyfikowane nie może być pokryte przez testy.

Temat ryzyka jest mocno teoretyczny i sformalizowany w opisie projektów informatycznych. Początkujący testerzy albo nie mają wystarczającej wiedzy, aby identyfikować ryzyka, albo źle diagnozują ich krytyczność w projekcie. Zbyt często zdarza się nam przesadzić w ocenie konsekwencji danego zdarzenia. Dlatego też testowanie oparte na ryzyku planują i analizują osoby bardziej doświadczone.

## 7.3. Planowanie

Uruchomienie testów zaczyna się wtedy, kiedy aplikacja testowa jest dostępna, a testerzy mają przypisane zadania. Przedtem jednak trzeba wykonać wiele kroków. Planowanie jest domeną menedżerów, ale ponieważ często sobie z tym nie radzą, spada to na testera. Do testów trzeba się odpowiednio przygotować, jeśli tylko mamy na to czas, a żeby zaplanować działania, należy chociaż wstępnie je przemyśleć.

Nie zrozum mnie źle. Nie uważam, że początkujący tester oprogramowania ma kopniakiem wywierać drzwi do projektu i przejmować wszystkie role i zadania projektowe. Testowanie jest skomplikowanym i precyzyjnym instrumentem i jedynie zręczny rzemieślnik z pasją może go nastroić. Nie powierzam więc definiowania strategii procesowej i produktowej oraz planowania młokosom, którzy ledwo co wytarli mleko matki spod nosa. Nie chcę, aby armia nieprzygotowanych i niedoświadczonych amatorów kierowała rozwojem i jakością oprogramowania. Chcę za to, aby zaczęli praktykować u swoich mistrzów. Chciałbym, aby małymi kroczkami przygotowywali się do tego, że pewnego dnia sami zostaną mentorami kolejnego pokolenia testerów. Aby to osiągnąć, trzeba zacząć już od wczesnych momentów zaangażowania w projekt. Planuj małe zadania, planuj dzień pracy, planuj jedną wersję oprogramowania, ale realnie planuj. Nie używaj metody Ctrl C – Ctrl V z marnych wzorców. Nie używaj nadmuchanych do granic możliwości standardów IEEE 829 czy ISO 29119. Planuj testowanie w zakresie odpowiednim do kontekstu. Domyślam się, że w życiu planujesz pewne rzeczy: zakupy, weekend, kupno mieszkania. Wiesz, że plany nie zawsze wychodzą. Możesz zdiagnozować, dlaczego tak jest i dzięki temu możesz następnym razem

zaplanować lepiej. Nie inaczej jest z zadaniami w projektach i całymimi projektami. Planuj.

Część planu przyjdzie z góry. Twój bezpośredni przełożony w najkrótszej formie może ci powiedzieć, że dziś testujesz to i to. Może dać ci jednak szczegółowy plan na cały dzień. Bez względu na szczegółowość jego planu, zawsze znajdzie się w nim obszar nie do końca opisany, luka, którą musisz wypełnić treścią samodzielnego planowania. Planuj więc. Definiuj pomniejsze cele, które chcesz osiągnąć, i określaj ścieżki dojścia do nich. Pomyśl o tym wszystkim, o czym nie pomyślał ktoś inny.

To wcale nie znaczy, że masz się zagłębić w żmudną pracę zakonnika przepisującego święte księgi. Wprost przeciwnie. Im plan ma krótszy zasięg czasowy, tym mniej w nim wpiszesz treści. Twoja lista zakupów, będąca planem jak każdy inny, zawiera przecież tylko hasła wywoławcze. Nie mówi, że po wejściu do sklepu masz się udać do regału 10. Plan nawet nie musi być spisany. Ja osobiście plany spisuję, bo dzięki temu, jeśli zapomnę, gdzie podążam, sprowadzają mnie na właściwe tory. Ten plan, który mam na myśli, ma ci przypominać, gdzie idziesz i jaka praca przed tobą. Jednocześnie nie bój się odchodzić od planu. Zmieniają się okoliczności projektu, pojawiają się i odchodzą ludzie, programiści zbyt wolno kodują, a klienci proszą o modyfikację wymagań. Adaptuj się ze swoim planem. Ze skrótowym planem jest jak z małą firmą – szybciej reaguje na zmiany. Zachowaj tak krótki plan, jak to możliwe.

Plan testów jako formalny dokument jest jednym z podstawowych w procesie testowym. Zawiera kilka elementów, takich jak wprowadzenie do testowanego produktu oraz strategia testów. Określ ją, aby doprowadzić do osiągnięcia pożądanego celu. Aby dobrze przeanalizować zakres testów, warto sobie wypisać funkcje, które będą im podlegać. Może się do tego przydać (jeśli tylko jest) wszelkiego rodzaju dokumentacja projektu i produktu. Definiując funkcje, można również wypisać środowiska, w których chcemy lub będziemy testować nasz produkt. Jeśli pewne elementy nie będą testowane, warto o nich również wspomnieć wraz z uzasadnieniem, dlaczego właśnie ten element planujesz pominąć w testach.

Jako testerzy mamy swoje obowiązki względem projektu. Oznacza to, że wykonaną pracę musimy jakoś udokumentować, czyli dostarczyć raport lub podsumowanie. Oczywiście dostarczymy również informację o znalezionych defektach i awariach. Warto przed rozpoczęciem

testowania przeanalizować również moment zakończenia testowania. Jak pisałem już przy różnych procesach testowych, określamy to jako kryterium zakończenia testów.

Osobiście jestem zwolennikiem koncepcji jednostronicowego planu testów opisanego w książce Lisy Crispin i Janet Gregory „Agile Testing”. Wystarczy, że wymienione tu elementy potraktujesz jako nagłówki, wypełnisz treścią i plan testów jest gotowy. Pozostań jednak na poziomie jednej strony. Im dłuższy dokument, tym większe prawdopodobieństwo, że nikt go nigdy nie przeczyta.

Jakie elementy planu uważam za nadmiarowe i zbędne? Te, które się pojawiają w standardach albo w zbyt, moich zdaniem, sformalizowanych organizacjach:

- Cel dokumentu. Możesz opisać tylko pod warunkiem, że musisz sobie przypomnieć powody napisania danego dokumentu.
- Wprowadzenie. Zazwyczaj w planie testów ten akapit jest pomijany bez czytania, a ponieważ nikomu nie chce się go uzupełniać, zostaje przyklejony z innego dokumentu.
- Testowany element. Mam nadzieję, że wiesz, co testujesz, a ponieważ nasz lekki plan testów ma niewielki zakres, nie zapomnisz tego. Opis oprogramowania zostawmy klientom i analitykom biznesowym.
- Kryteria zaliczenia lub niepowodzenia testu. Nie. Po prostu nie. Znasz swoje wyrocznie, które mówią ci, czy test przeszedł, czy też nie.
- Kryteria wstrzymania projektu testowego i jego odwieszenia. Planujesz na tak krótki okres, że tego nie potrzebujesz. Im krótszy okres, tym prawdopodobieństwo krytycznych zmian w projekcie jest mniejsze.
- Dostawa z testów, czyli to, co zamierzasz dostarczyć do projektu. Ważny element, ale nie dla pojedynczego testera, którego plan dostaw został zdefiniowany przez przełożonego.
- Odpowiedzialności. Oczywiście nie, ponieważ redukujemy odpowiedzialność do pojedynczej osoby.
- I wiele innych, które mają znaczenie na poziomie planowania odpowiedzialności menedżera, ale nie na poziomie testera, np. potrzeby szkoleniowe pracowników, liczba i rola zasobów ludzkich do przeprowadzenia testów, harmonogram prac, zdefiniowane ryzyka jakościowe i metody ich minimalizacji.

Oczywiście nie wymaga się też formalizacji nieformalnego dokumentu. W biurokratycznych organizacjach plan zdefiniowany przez menedżera testów musi zostać zatwierdzony przez menedżera wyższego szczebla, zazwyczaj kierownika projektu. Takiej akceptacji nie potrzeba, jeśli plan powstaje jedynie dla nas.

Elementy planu testów, które uważam za ważne:

- Identyfikator dokumentu, służący również do wersjonowania. Może być zawarty w nazwie pliku. Jeśli o tym jeszcze nie wspomniałem, to uzupełniam: plan testów powinien pozostawać w formie elektronicznej. Wydrukowany plan testów jest tak przydatny jak zeszłotygodniowa gazeta.
- Testowane elementy (w tym charakterystyki). To element niezbędny, bo musisz znać zakres. Możesz się hasłowo odwołać do funkcji i cech, ale to jest twój zakres i musisz go spisać.
- Podejście do testów, ale opisane jedynie hasłowo. Dostosuj swoje podejście do poznanych strategii testów i nazwij je po ogólnie przyjętym imieniu.
- Zadania w testach, czyli to, co będziesz robił w zaplanowanym czasie. Oczywiście nie zapisane szczegółowo.
- Środowisko. Elementy składowe środowiska testowego wraz z podanymi wersjami są istotne dla określenia, w jakich okolicznościach i w jakich wersjach pojawił się defekt lub awaria.

Plan w takim formacie zawiera jedynie najważniejsze informacje. Oczywiście, w zależności od potrzeb, można go rozszerzać, jednocześnie zachowując umiar. Pamiętaj, że sobie opisujesz, jak testy mają wyglądać.

Warto podkreślić, że to jedynie wycinek z elementów, które mogą się pojawić w planie testów. Więcej można znaleźć w kompleksowym opracowaniu planu testów w normach lub też na [testerzy.pl](http://testerzy.pl).

## 7.4. Testowanie

Dla wielu „testowanie” jest równoznaczne z zaprojektowaniem testów i ich uruchomieniem. Zanim jednak przejdziemy do przykładów testów, skoncentrujmy się na podstawowych składowych aplikacjach:

- elementach,
- funkcjach,
- logice lub procesach.

Dodatkowo uzupełniam ten zestaw o formularze, ponieważ są standardem w wielu aplikacjach z interfejsem graficznym.

Jest to zbiór, z którego tworzy się aplikacje udostępniane potem użytkownikom. Nieskończony potencjał wewnętrzny oprogramowania, mnogość działań i operacji obsługuje się zazwyczaj skończonym zbiorem elementów interfejsu i jego funkcji, opisując to oczywiście pewną logiką lub – jak wolą niektórzy – procesem.

### 7.4.1. Element

Aplikacje składają się z wielu elementów, widocznych zazwyczaj w ich interfejsie. Są to elementy nawigacyjne, graficzne czy elementy formularzy. Za większością z nich kryją się drobne funkcje i małe wycinki procesów, które uruchamiamy. Poprawność ich działania nie jest może tak ważna, jak opisane dalej działanie funkcji, ale stanowi ważną składową oceny jakości aplikacji przez użytkownika. Elementy mają działać przede wszystkim spójnie i zgodnie z tym, czego użytkownik od nich oczekuje, co z kolei wiąże się z testowaniem ich użyteczności.

Przykłady elementów:

- przyciski wyboru,
- listy rozwijane,
- linki (w tym grafiki linkujące),
- przyciski w odtwarzaczu,
- strzałki nawigacyjne.

Elementy nie zawsze muszą być widoczne w interfejsie użytkownika. Mimo to ich obsługa jest ważna dla użytkownika w posługiwaniu się aplikacją, a dla testera to dodatkowy obszar do weryfikacji. Mowa m.in. o rozpoznawaniu gestów użytkownika w ramach aplikacji mobilnej czy skrótach klawiaturowych.

Co jest, a co nie jest elementem? Do podstawowego sprawdzenia elementu tester będzie zazwyczaj potrzebował pojedynczego kliknięcia



lub gestu. Są to więc rzeczy bardzo małe i (teoretycznie) proste do sprawdzenia.

### **7.4.2. Formularze**

Testowanie formularzy to sól pracy testera. Przykładowy formularz pokazany na rys. 7.4 to tylko jeden z wielu możliwych. Wszystkie te zależności, pojawiające się i znikające pola, dane wymagane i niewymagane, różnorodne elementy interfejsu, nawigacja po polach, podpowiedzi, to chleb powszedni testerów aplikacji z graficznym interfejsem użytkownika. Tu wypisywanie przypadków testowych nie ma większego sensu, ponieważ mnogość formularzy implikuje mnogość testów.

Imię	<input type="text"/>
Nazwisko	<input type="text"/>
Email	<input type="text"/>
Hasło	<input type="password" value="Zarządzaj"/>
Kraj	<input type="text" value="Wybierz"/>
Strefa czasowa	<input type="text" value="Wybierz"/>
Kod pocztowy	<input type="text"/>
Data urodzenia	<input type="text"/> <input type="text"/> <input type="text"/>
Płeć	<input type="text" value="Mężczyzna/Kobieta"/>
Jednostki	<input type="text" value="Wybierz"/>
Waga	<input type="text"/>
Wzrost	<input type="text"/>
Telefon	<input type="text"/>
Newsletter	<input type="password" value="Zarządzaj"/>



Rys. 7.4. Przykładowy formularz w aplikacji

Jedną z pierwszych rzeczy po prostym zweryfikowaniu możliwości ukończenia formularza będzie określenie metody walidacji. Może ona przebiegać na różne sposoby:

- walidacja blokująca wprowadzenie danych do pola np. pole Numer

- walidacja blokująca wprowadzanie znaków do pola, np. pole numer rachunku przyjmuje tylko cyfry;
- walidacja w czasie trwania wpisywania, usuwająca znaki niezgodne z jej regułami;
- walidacja po wprowadzeniu tekstu i opuszczeniu pola (zazwyczaj komunikat lub automatyczne zablokowanie pola);
- walidacja po zatwierdzeniu całego formularza (komunikat o błędzie).

Testowanie walidacji to:

- dwa testy: wprowadzenie znaków: „z palca” i „wklejanie”,
- dwie klasy znaków: poprawne i niepoprawne,
- testy dla ciągu o określonej długości: ciąg zerowy (zawsze stosowany tam, gdzie pole jest wymagane), wartości na granicy – minimalna i maksymalna, oraz długość ciągu powyżej maksymalnej.

Do tego dochodzą następujące testy:

- czy formularz można ukończyć (bez względu na typ walidacji);
- testy użyteczności (przydatności i czytelności) komunikatów błędów;
- testy obciążające przy dużej liczbie znaków w polu, głównie testowanie ograniczeń na polach;
- testy bezpieczeństwa, czyli wprowadzanie ciągów mogących uszkodzić lub ujawnić dane i funkcje (np. ataki na bazę);
- testy elementów dodatkowych formularza (np. klasyczny i bardzo niepotrzebny przycisk Wyczyść formularz);
- testy nawigacji przy użyciu klawisza tabulacji (Tab);
- test czyszczenia formularza – czy pola formularza są opróżniane częściowo, czy w całości po niepoprawnym uzupełnieniu jednego z pól.

Inne ciekawe testy negatywne i dane testowe dla liczb przedstawione są w tab. 7.3. Warto podkreślić, że podane wartości nie są wyłącznie danymi testowymi, ale również reprezentują długość ciągów znaków do wprowadzania.

**Tab. 7.3.** Szczególne dane testowe dla systemów informatycznych

---

Dana testowa/test	Opis
0	Zero to podstawowa dana testowa. Jej przydatność wynika z jej częstego niepoprawnego użycia. Przykładowo użytkownik wstawi w pole wymagane wartość 0, a program sprawdzi, czy wartość w polu nie jest równa NULL; może to powodować, że choć pole nie jest puste, to komunikat będzie głosił, że puste jest. Zero to również początek zbioru liczb nieujemnych i naturalnych. Jest często stosowane jako wartość graniczna
127	W języku C jest tzw. <i>char</i> , w którym zapisuje się długość ciągu znaków. Podana wartość jest na granicy
128	W języku C jest tzw. <i>char</i> , w którym zapisuje się długość ciągu znaków. Podana wartość jest zaraz za granicą
255	Wartościowe jako wartość zaraz za granicą wartości 256, opisanej poniżej
256	Wartość ta w informatyce ma wiele zastosowań. Przykładowo: bajt ma 256 możliwych wartości, liczba znaków w rozszerzonym kodzie ASCII itd.
257	Wartościowe jako wartość zaraz za granicą wartości 256
1000	Dana testowa o wielu zastosowaniach i znaczeniach zarówno w matematyce, jak i w definiowaniu czasu
1024	2 do potęgi 10. Szerokie zastosowanie tej wartości w inżynierii komputerowej czyni z niej ciekawą daną testową
2000	Wartość ciekawa przede wszystkim z powodu błędu roku 2000 (Y2K). Warto znać i pamiętać
32767	W języku C jest tzw. <i>short int</i> , w którym zapisuje się wartości zmiennych. Podana wartość jest na granicy
32768	W języku C jest tzw. <i>short int</i> , w którym zapisuje się wartości zmiennych. Podana wartość jest zaraz za granicą
65536	W języku C jest tzw. <i>short int bez znaku</i> , w którym zapisuje się wartości zmiennych. Podana wartość jest na granicy
65537	W języku C jest tzw. <i>short int bez znaku</i> , w którym zapisuje się wartości zmiennych. Podana wartość jest zaraz za granicą
2147483647	W języku C jest tzw. <i>long int</i> , w którym zapisuje się wartości zmiennych. Podana wartość jest na granicy
2147483648	W języku C jest tzw. <i>long int</i> , w którym zapisuje się wartości zmiennych. Podana wartość jest zaraz za granicą
4294967295	

	W języku C jest tzw. <i>long int bez znaku</i> , w którym zapisuje się wartości zmiennych. Podana wartość jest na granicy.
4294967296	W języku C jest tzw. <i>long int bez znaku</i> , w którym zapisuje się wartości zmiennych. Podana wartość jest zaraz za granicą
Użycie poprawnej notacji naukowej	Przykład: 1E-16. Oczywiście zakładamy, że nie wszystkie systemy muszą obsługiwać taką notację
Wartości negatywne	Szeroka definicja wszystkich wartości, które nie powinny być obsługiwane w systemie, albo ich użycie powinno być zakończone pojawieniem się komunikatu błędu
Wartości zmiennoprzecinkowe z zapisem z kropkami	Przykład: 0.0001
Wartości zmiennoprzecinkowe z zapisem z przecinkami	Przykład: 1,234,567
Znaki diakrytyczne	Przykład: à á â ã ä å ç è é ê ë ì í î ï ð ñ ò ó ô õ
Czcionki azjatyckie	Przykład z języka chińskiego: 測試 czy z języka japońskiego: 検査
Ograniczniki i znaki specjalne	Przykład: “ ‘ `   /\, ; : & < > ^ * ? Tab
Puste	Niepodawanie żadnej wartości
Spacja	Podanie znaku spacji
Wiele spacji	Podanie wielu spacji
Spacja na końcu	Podanie spacji na końcu ciągu znaków
Znak końca wiersza	Podanie wartości ^M

Olbrzymi zestaw przykładowych danych testowych, które mogą ujawniać defekty lub zmuszać aplikację do pokazania nieobsługiwanego wyjątku, znajdziesz na stronie projektu Big List of Naughty Strings.

Warto pamiętać, że podane tu testy i dane testowe mają bardzo szerokie zastosowanie. Ich prawdziwa wartość ujawnia się jednak dopiero przy próbie ich łączenia, np. jak zachowa się aplikacja, jeśli do pola wprowadzimy ciąg o maksymalnej dopuszczalnej długości i na końcu dodamy spację?

### 7.4.3. Funkcja

Wiele źródeł nie rozpoznaje testowania funkcji jako ważnej techniki testowania oprogramowania. Jest to poważna luka w ocenie poprawności działania oprogramowania. Na każdą aplikację składa się kilka, kilkanaście funkcji, a czasami nawet setki i tysiące. Ważne jest zweryfikowanie w pierwszej kolejności, czy dane funkcje działają, a nie czy działają pojedyncze kroki prowadzące do wykonania funkcji. Co prawda, podczas testowania formularza zakładania konta w portalu możemy sprawdzić osobno, czy działa pole Login, czy działa pole Hasło, Powtórz hasło, Zaakceptuj regulamin itd., ale istotne jest sprawdzenie przede wszystkim, czy istnieje możliwość założenia konta. To powinno być naszym pierwszym i podstawowym działaniem w testach. Jeśli chcemy być postrzegani jako osoba, która wnosi coś do procesu wytwarzania oprogramowania, to naszym głównym celem będzie sprawdzenie, czy ważne funkcje działają, czy też nie. Dopiero potem sprawdzimy, czy ścieżki alternatywne w aplikacji działają poprawnie np. czy dane z krańców przedziałów są walidowane w poprawny sposób po ich wprowadzeniu.

Każde oprogramowanie ma swoje bardziej i mniej krytyczne funkcje, których (nie)poprawność działania wpłynie na postrzeganie całości. Czasami są one unikatowe dla danego oprogramowania, czasami powtarzają się w większości znanych aplikacji (por. rys. 7.5).

- Test: działanie przycisku Zapisz  
Oczekiwany rezultat: poprawne wykonanie funkcji związanej z przyciskiem, czyli zapisanie danych.
- Test: działanie przycisku Anuluj  
Oczekiwany rezultat: wykonywana operacja jest przerywana, dane się nie zapisują.
- Test: działanie przycisku Cofnij zmiany  
Oczekiwany rezultat: cofnięcie ostatniej wprowadzonej zmiany.
- Test: działanie przycisku Przywróć domyślne ustawienia  
Oczekiwany rezultat: wszystkie ustawienia zostają przywrócone do domyślnych.
- Test: zmiana ustawień preferowanego języka aplikacji  
Oczekiwany rezultat: wszystkie informacje i komunikaty w aplikacji prezentowane są w wybranym języku.

- Test: zmiana ustawień strefy czasowej  
Oczekiwany rezultat: czas pokazywany przy zalogowanej aktywności użytkowników prezentowany jest zgodnie z wybraną strefą czasową.



**Rys. 7.5.** Przykłady funkcji wywoływanych przez klawisze

I tak dalej. Każda funkcja będzie miała swój zerojedynkowy weryfikator spełnienia. W przypadku funkcji zdefiniowanej jako działa/nie działa mówimy, że została zaimplementowana albo że implementacja się nie powiodła.

Przedstawiłem tutaj proste funkcje dostępne po kliknięciu pojedynczego przycisku. Dla wielu aplikacji funkcja jest czymś znacznie bardziej złożonym i niedającym się pokryć pojedynczym działaniem. Te najbardziej popularne zostaną opisane w dalszych przykładach (testach).

#### **7.4.4. Logika lub proces**

Testowanie logiki aplikacji w wielu źródłach określa się jako testowanie białoskrzynkowe, ze zrozumieniem wewnętrznej budowy oprogramowania. Wolę tę nazwę od częściej używanej „testowanie

procesów” czy też „testowanie warunków – efekt”, bo wydaje się lepiej oddawać sens testowania poprawności działania aplikacji w zależności od wewnętrznych reguł. Co prawda, wypłatę z bankomatu można nazwać procesem decyzyjnym, ale jest to perspektywa systemu. Użytkownik wymaga wbudowanego w oprogramowanie logicznego ciągu zdarzeń, które pozwolą mu osiągnąć sukces przy wykonywaniu operacji. Jeśli na przykład bankomat działa poprawnie ORAZ ma wystarczająco gotówki, ORAZ posługuję się poprawną kartą płatniczą i poprawnym PIN-em, ORAZ nie został przekroczony limit wypłat, ORAZ mam środki na koncie, TO środki muszą być wypłacone. Możemy to oczywiście nazwać złożeniem wielu przyczyn i końcowym skutkiem, ale czy jest to nomenklatura użytkownika? Użytkownik powie: „Skoro wszystkie warunki zostały spełnione, to logiczne jest, że wypłacę środki z bankomatu”. Czasami następują zdarzenia, które nie są w pełni widoczne dla użytkownika, a ich zaistnienie zablokuje możliwość wypłaty. Przykładowo bank zablokuje jego kartę bez jego wiedzy. Informacja o tym dotrze do niego dopiero w momencie, kiedy bankomat odmówi jej oddania i wypłacenia środków. Logika z perspektywy użytkownika zostaje zaburzona.

Testowanie logiki jest wyższą szkołą testowania oprogramowania i wymaga świadomości działania oprogramowania. Logicznie musi zostać spełnionych  $n$  warunków na wejściu, aby na wyjściu uzyskać pewien rezultat lub pewne rezultaty. Zadaniem testera będzie nie tylko sprawdzenie, czy ma to miejsce, ale czy istnieje możliwość zaburzenia logiki. Przykładowo użytkownik próbuje dokonać kolejnej wypłaty, która przekroczy jego limit. Logicznie rzecz biorąc, powinien zostać przed tym powstrzymany, ale czy naprawdę tak się stanie?

Testowanie logiki będzie zawsze wyzwaniem zmuszającym testera do przewidzenia wielu różnorodnych scenariuszy, jakie mogą pojawić się w systemie. Tak jak testowanie funkcji, również testowanie logiki będzie wpisane w wiele z pokazanych tu przykładów, ale nie wyczerpie pełnego zakresu możliwych testów.

## Przykłady

Przedstawiona w tym rozdziale lista testów to jedynie sugestie i oczywiście nie wyczerpują one wszystkich możliwych przypadków.

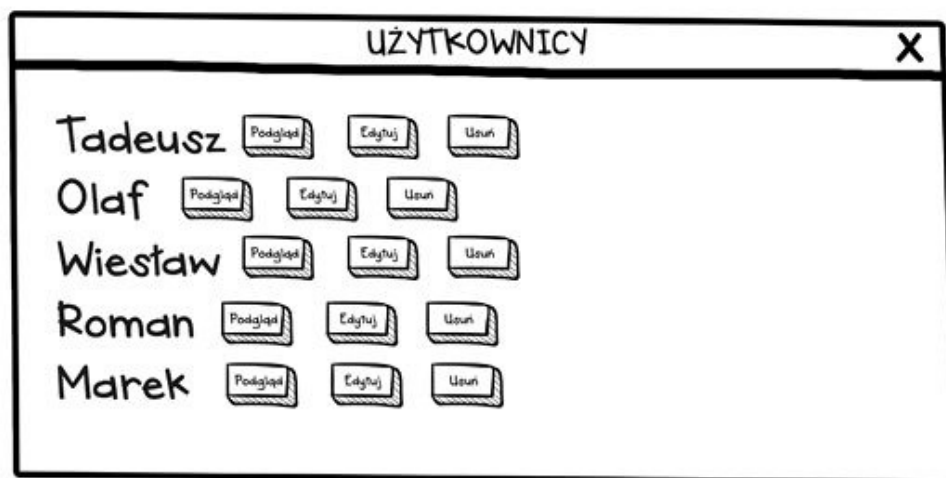


Stanowią natomiast podstawową listę kontrolną, która mogłaby być użyta w testach aplikacji z danymi funkcjami. Również notacja jest raczej uproszczona i nie uwzględnia wszystkich elementów standardowo wymaganych od przypadku testowego.

**Testowanie wyświetlania interfejsu.** To będzie prawdopodobnie najbardziej znienawidzony przez testerów zestaw testów do przeprowadzenia. Nie jesteśmy w końcu buddyjskimi mnichami (buddyjskich mnichów, którzy czytają tę książkę bardzo przepraszam za wykluczenie), aby poświęcać swoją pełną uwagę usypianym z piasku mandalom. Większość nie lubi pracy wymagającej porównywania małych szczegółów w interfejsie użytkownika, grzebania w kolorach czy mierzenia cyfrową linijką odległości między elementami. Taka praca może się jednak pojawić, jeśli naszym zadaniem będzie testowanie interfejsu użytkownika. Testujemy naprawdę mało ważne elementy, ale niektóre z nich z perspektywy użytkownika urastają do krytycznych defektów. Zdarzyło mi się słyszeć od użytkowników m.in.:

- „Miał być kolor ecru, a jest waniliowy”,
- „W kolumnie wszystkie elementy są równo ułożone do lewej, a ten jeden jest przesunięty o dwa piksele”,
- „Trójkąt symbolizujący „uruchom” jest równoramienny, a powinien być równoboczny”.

Czasami zgłoszenia związane z interfejsem dotyczą chaosu np. w rozłożeniu elementów, co może utrudniać posługiwanie się oprogramowaniem i być traktowane jako poważny defekt. Przykład takiego niepoprawnego rozłożenia elementów interfejsu pokazano na rys. 7.6.



**Rys. 7.6.** Przykładowy niepoprawny interfejs utrudniający użycie funkcji

Mogłoby to dowodzić, że w aplikacji nie ma się do czego przyczepić, ale takie elementy zwrócą uwagę szybciej, niż niepoprawność działania funkcji. Zwłaszcza osoby, które są negatywnie nastawione do produktu, będą koncentrować swoją uwagę na takich szczegółach. Jednakże nie tylko one. Części użytkowników będzie to przeszkadzać, co przełoży się na gorszy odbiór aplikacji. Zgłoszenia takich defektów są traktowane jako trywialne i otrzymują najniższy priorytet naprawy.

Warto pamiętać, że takie testy są bliższe charakterystyce użyteczności niż funkcjonalności. Możemy przeprowadzić dwa typy testów: weryfikujące zgodność z wymaganiami oraz sprawdzające tzw. dobre praktyki. Druga ocena jest bardziej subiektywna, więc zgłoszenia z tego obszaru mogą wymagać szerszego uzasadnienia.

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Sprawdzenie zdefiniowanych w wymaganiach reguł rozmieszczenia elementów i ich kształtów	Elementy znajdują się w miejscu, w którym powinny być i mają zdefiniowany kształt	Zbadaj wiele źródeł wymagań klienta/użytkownika	Dowolne oprogramowanie z interfejsem graficznym

2	Sprawdzenie niezdefiniowanych w wymaganiach, ale niezgodnych z dobrymi praktykami reguł rozmieszczenia elementów	Rozmieszczenie elementów jest zgodne z regułami użytymi w podobnej aplikacji	Warto się odwołać do aplikacji popularnej/z naszej branży/uważanej za najlepszą	Dowolne oprogramowanie z interfejsem graficznym
3	Sprawdzenie zdefiniowanych w wymaganiach reguł kolorystyki	Elementy interfejsu mają kolory zgodne z wymaganiami	Zbadaj wiele źródeł wymagań klienta/użytkownika	Dowolne oprogramowanie z interfejsem graficznym
4	Sprawdzenie niezdefiniowanego w wymaganiach, ale niezgodnego z dobrymi praktykami doboru kolorów	Użycie kolorów jest zgodne z regułami użytymi w podobnej aplikacji lub z dobrymi praktykami	Warto się odwołać do aplikacji popularnej/z naszej branży/uważanej za najlepszą. Warto pamiętać o zasadach: duży kontrast między tekstem a tłem, unikanie kolorów jaskrawych itd.	Dowolne oprogramowanie z interfejsem graficznym

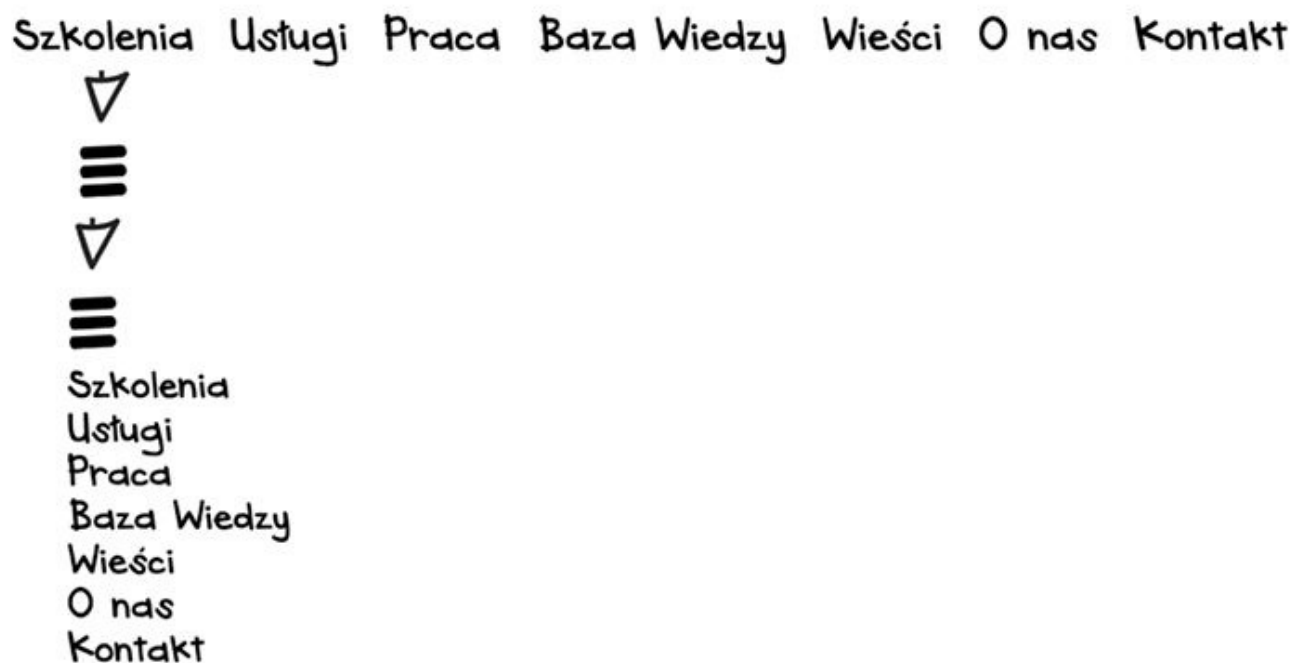
**Testowanie aplikacji responsywnej.** Zaprojektowane responsywnie strony (*responsive web design* – RWD,) to działania mające na celu stworzenie strony internetowej, która sama będzie się adaptować do rozmiaru i rozdzielczości ekranu, na którym jest wyświetlana (por. rys. 7.7).



**Rys. 7.7.** Wizualizacja dla różnych ekranów, na których może być prezentowane oprogramowanie

To zagadnienie szczególnie ważne w czasach zmiany przyzwyczajeń dotyczących dostępu do treści w sieci. Kiedyś naszym głównym i jedynym narzędziem był komputer. Teraz strony oglądane są na ekranach telewizorów, komputerów (stale), tabletów i telefonów. Strona nie może być już sztywnym rozwiązaniem, ale musi się elastycznie dopasować do urządzenia. Jeden z mistrzów użyteczności i projektowania interfejsów Jakob Nielsen twierdzi, że prawdziwą użyteczność osiągniemy jedynie przez ręczne (a nie automatyczne) dopasowanie interfejsu do danego typu wyświetlacza czy też platformy. Oznaczałoby to konieczność budowania osobnej aplikacji na różne wyświetlacze. Jest to głos odosobniony. Po drugiej stronie barykady stoją administratorzy firmy Google, którzy przekonują do projektowania uniwersalnych aplikacji, które same dopasują się do wyświetlacza urządzenia. Jest to nie tylko rozwiązanie tańsze, ale jednocześnie łatwiejsze do konstrukcji i do testowania. Realnie testujemy jedną aplikację w wielu środowiskach, a nie wiele aplikacji w wielu środowiskach.

Jeśli aplikacja skonstruowana jest do pojedynczego urządzenia, to oczywiście nie testujemy jej adaptacyjności. Jeśli jednak stworzona jest zgodnie z koncepcją RWD, to musimy ją przetestować. Główne założenie tej koncepcji to elementy interfejsu, które nie są na stałe „przypięte” do ekranu. Mogą się w pewnych granicach przesuwać lub zmieniać rozmiar. Oprócz tego modyfikacje nie idą zbyt daleko. Chodzi również o to, aby doświadczenie korzystania z aplikacji mobilnej było zbliżone do pracy na komputerze. Jedną z głównych i najpoważniejszych zmian w interfejsie będzie zamiana zwykłego menu aplikacji w przycisk menu (por. rys. 7.8). Oczywiście trzeba uwzględnić pewne ograniczenia we wprowadzaniu danych (brak klawiatury) i nawigowaniu po interfejsie (brak myszy) oraz to, co daje interfejs mobilny, czyli między innymi dotykowy ekran, kontekstową klawiaturę (dopasowaną do informacji, którą chcemy wprowadzić) czy widok wertykalny i horyzontalny.



**Rys. 7.8.** Zastąpienie klasycznego poziomego menu przyciskiem, po którego kliknięciu otrzymujemy pionowe menu

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz
1	Sprawdzenie poprawności wyświetlania elementów w różnej rozdzielczości/w różnych przeglądarkach/na różnych platformach	Elementy interfejsu wyświetlają się w różnej rozdzielczości/w różnych przeglądarkach/na różnych platformach tak samo lub bardzo podobnie	Zbadaj w najbardziej popularnych rozdzielczościach/przeglądarkach/na najbardziej popularnych platformach
2	Sprawdzenie adaptacji strony do zmieniającej się orientacji (wertykalna <-> horyzontalna)	Strona powinna się dynamicznie dopasować do zmieniającej się szerokości ekranu	Powinna być możliwa adaptacja wszystkich stron internetowych, chyba że sama przeglądarka nie obsługuje dwóch orientacji
3	Sprawdzenie czy tekst na różnych rozdzielczościach jest możliwy do przeczytania (adaptacji rozmiaru czcionki)	Rozmiar czcionki pozwala na czytanie tekstu na stronie	Zbadaj w wielu środowiskach
4			

	Sprawdzenie zmiany funkcjonowania i obsługi strony przy różnych interfejsach	Funkcje stron są dopasowane do interfejsu, w którym są obsługiwane	Przykładowe funkcje, których zachowanie będzie inne w środowisku mobilnym i stacjonarnym: formularz, wyskakujące okna, galerie itd.
5	Sprawdzenie pobierania obrazów dopasowanych do możliwości danego urządzenia/rozdzielczości	Pobrane obrazy są zoptymalizowane pod kątem wyświetlacza i rozdzielczości	Grafika dopasowana do rozdzielczości to m.in. szybsze ładowanie się stron na urządzeniach mobilnych

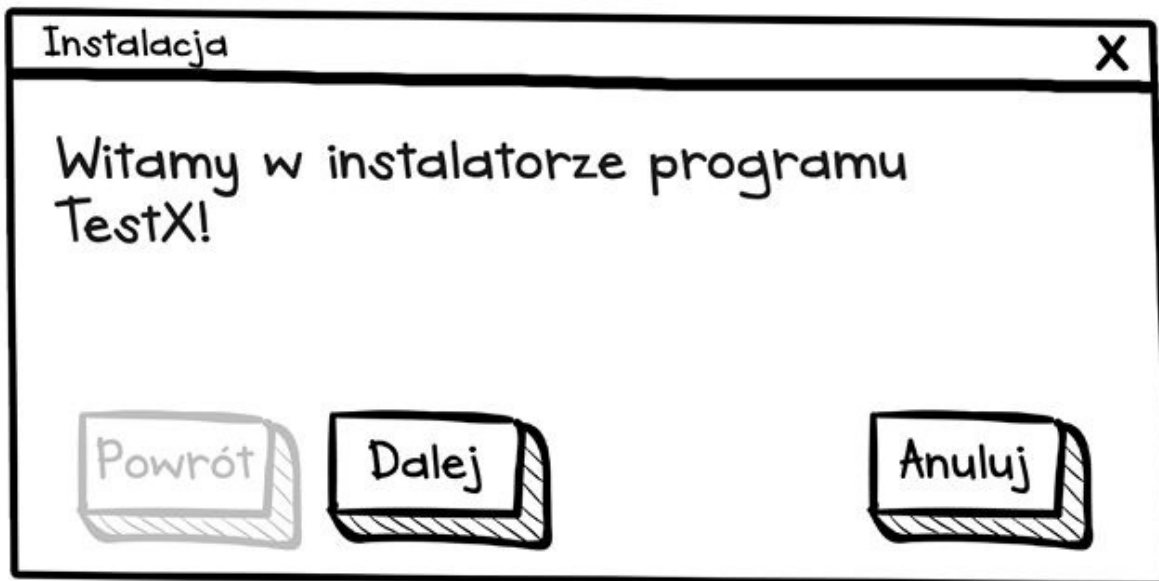
Zoptymalizowanie wyświetlania to jeden element. Drugim jest kompresowanie przesyłanych danych – z uwzględnieniem tego, że zazwyczaj dane pobierane są przez łącze o ograniczonej szybkości i/lub jest to połączenie taryfowane (płatne). Obrazy czy filmy nie muszą być w najwyższej rozdzielczości i jakości, a muszą (mogą) być dopasowane do możliwości sprzętu, który je prezentuje. Grafiki na ekranach mobilnych nie muszą być więc tak duże jak na ekranie komputera.

**Testowanie instalacji, aktualizacji i dezinstalacji.** Uruchomienie aplikacji w danych warunkach wymaga zazwyczaj jej wcześniejszego zainstalowania w danym środowisku. Proces ten jest podstawowy ze względu na kilka elementów. Nikt nie lubi instalacji. Instalacja to ostatni krok, który dzieli nas od używania samej aplikacji, zwłaszcza jeśli jest to na przykład gra lub aplikacja, na którą z niecierpliwością czekaliśmy.

Użytkownik będzie miał swój pierwszy kontakt z aplikacją właśnie podczas instalacji. Ponieważ zgodnie ze starym powiedzeniem mamy tylko jedną szansę na dobre pierwsze wrażenie, należy ją optymalnie wykorzystać. Proces instalacji musi być możliwie jak najmniej denerwujący. Instalacja w standardzie nie powinna wykonywać niczego, na co nie zgodził się użytkownik lub nie powinna wykorzystywać jego nerwowego klikania Dalej do zakończenia procesu. Wiele organizacji wykorzystuje to do wprowadzenia ustawień lub też zainstalowania dodatkowego oprogramowania. Musimy się tu jasno opowiedzieć po stronie dobra. Dobre jest jasne i klarowne informowanie o działaniu instalatora.

Kolejnym ważnym elementem cyklu życia oprogramowania będzie aktualizacja. Może być wykonywana ręcznie lub też automatycznie.

Życie aplikacji w naszym środowisku zazwyczaj kończy się wraz z dezinstalacją.



**Rys. 7.9.** Wizualizacja instalatora z funkcjami standardowymi

### Przykładowe testy

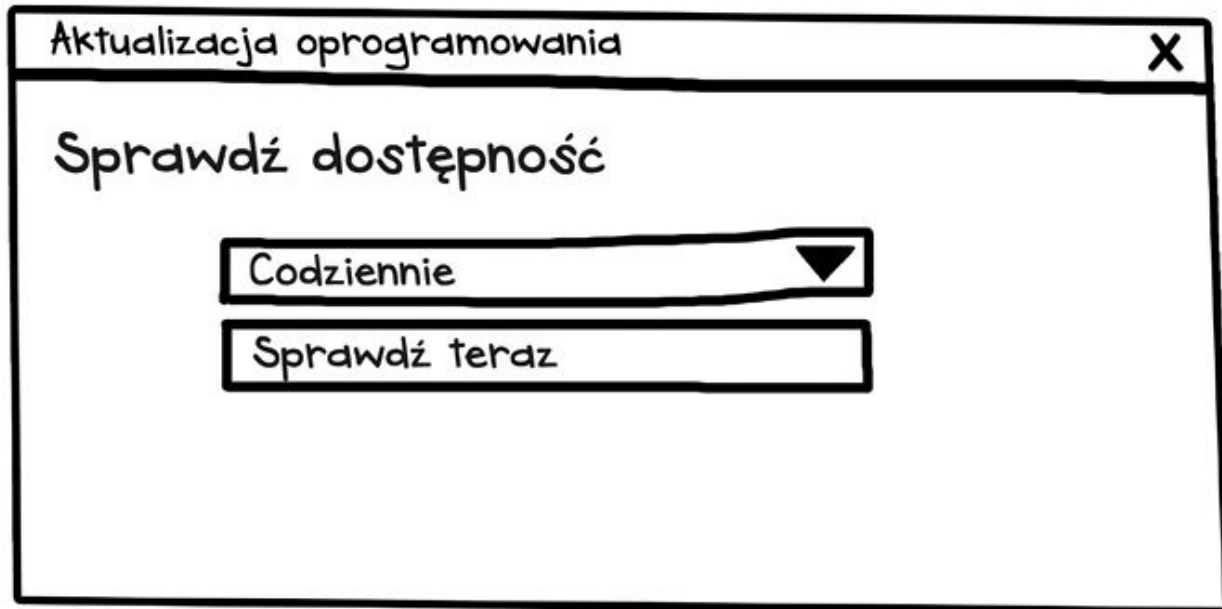
Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Sprawdzenie działania funkcji automatycznego uruchomienia instalatora	Automatyczny start instalatora po włożeniu nośnika, np. płyty	Brak	Dowolne oprogramowanie instalowane z zewnętrznego nośnika
2	Instalacja programu w domyślnej lokalizacji	Program instaluje się w lokalizacji sugerowanej przez producenta oprogramowania	Brak	Dowolne oprogramowanie instalowane na sprzęcie
3	Zmiana domyślnej lokalizacji (ręczne wpisanie ścieżki)	Instalacja programu w miejscu wpisanym przez testera, ewentualne pytanie o utworzenie folderu	Niemożliwość ustawienia własnej lokalizacji dla aplikacji nie powinna być uznana za defekt, jeśli nie ma takiego wymagania	Dowolne oprogramowanie instalowane na komputerze
4	Zmiana domyślnej lokalizacji (ręczne wpisanie ścieżki);	Informacja o błędnej ścieżce, prośba o podanie prawidłowej	Niemożliwość ustawienia własnej lokalizacji dla	Dowolne oprogramowanie

	podanie błędnej ścieżki (znaki specjalne, nieistniejący dysk, błędna składnia np.: c:test/instalacja\)		aplikacji nie powinna być uznana za defekt, jeśli nie ma takiego wymagania	instalowane na komputerze
5	Zmiana domyślnej lokalizacji (wskaż miejsce docelowe – lista)	Instalacja programu w miejscu wskazanym przez użytkownika	Brak	Dowolne oprogramowanie instalowane na sprzęcie
6	Zmiana domyślnej lokalizacji (wskaż miejsce docelowe – lista); wskazanie błędnego miejsca docelowego bez prawa lub możliwości zapisu	Informacja o niepoprawnej ścieżce, prośba o podanie prawidłowej	Brak	Dowolne oprogramowanie instalowane na sprzęcie
7	Instalacja przy braku wymaganego miejsca na dysku (pamięci)	Informacja o braku miejsca na dysku w podanej lokalizacji, prośba o wybór innej lokalizacji Brak lub zwolnienie miejsca na dysku	Brak	Dowolne oprogramowanie instalowane na sprzęcie
8	Sprawdzenie działania przycisków Dalej i Wstecz w instalatorze graficznym	Możliwość powrotu do poprzedniego lub następnego okna instalatora	Brak	Dowolne oprogramowanie instalowane na sprzęcie
10	Działanie przycisku Anuluj	Możliwość przerywania pracy instalatora na różnych etapach postępu instalacji, po uprzednim zaakceptowaniu „Czy na pewno przerwać instalację?”	Brak	Dowolne oprogramowanie instalowane na sprzęcie
11			Brak	



	Akceptacja regulaminu	Zatrzymanie instalacji w przypadku braku zaakceptowania regulaminu, informacja o konieczności jego zaakceptowania		Dowolne oprogramowanie instalowane na sprzęcie
12	Wybór typu instalacji np. minimalna, maksymalna, użytkownika	Instalacja wg wybranych kryteriów, w przypadku instalacji użytkownika możliwość wyboru elementów programu do zainstalowania	Brak	Dowolne oprogramowanie instalowane na komputerze
13	Wybór elementów w instalacji użytkownika (samodzielnie definiowanej)	Brak możliwości odznaczenia składników głównych programu, które są konieczne do poprawnego działania oprogramowania	Brak	Dowolne oprogramowanie instalowane na komputerze
14	Tworzenie skrótów do uruchomienia aplikacji	Utworzenie skrótów do programu w wybranych miejscach	Brak	Dowolne oprogramowanie instalowane na komputerze
15	Wybór języka instalatora	Zmiana wszystkich elementów tekstowych instalatora na wybraną wersję językową	Brak	Dowolne oprogramowanie instalowane na komputerze
16	Podanie klucza/licencji	Brak możliwości kontynuowania instalacji w przypadku podania błędnego klucza	Brak	Dowolne oprogramowanie z kluczem instalowane na komputerze
17	Informacja o zakończeniu instalacji	Wyświetlenie się informacji o ukończeniu instalacji	Brak	Dowolne oprogramowanie instalowane na sprzęcie
18	Sprawdzenie prośby o restart systemu po	Sprawdzenie, czy program uruchamia się	Wiele aplikacji można uruchomić mimo braku restartu.	Dowolne oprogramowanie

	zakończonych instalacji	mimo braku restartu systemu	Czy takie zachowanie jest defektem, czy też nie, możemy wywnioskować z instrukcji użytkownika	instalowane na sprzęcie
19	Automatyczne uruchomienie programu po ukończeniu instalacji	Program uruchomi się automatycznie, jeśli zaznaczymy taką opcję w instalatorze	Brak	Dowolne oprogramowanie instalowane na komputerze
20	Sprawdzenie kolejności pokazywanych ekranów instalatora	Ekrany instalatora ustawione są w ustalonej i/lub logicznej kolejności	Brak	Dowolne oprogramowanie instalowane na sprzęcie
21	Domyślne ustawienie zaznaczenia na przycisku Dalej	Na kolejnych ekranach przycisk Dalej jest domyślnie zaznaczony, co pozwala na zainstalowanie aplikacji w standardowych ustawieniach – wystarczy nacisnąć klawisz Enter	Brak	Dowolne oprogramowanie instalowane na komputerze
22	Próba instalacji w innym niż zalecane środowisko	Instalator nie próbuje instalować aplikacji ze względu na nieobsługiwane środowisko	Brak	Dowolne oprogramowanie instalowane na sprzęcie
23	Sprawdzenie instalacji bez wymaganych dla aplikacji składowych środowiska np. Bibliotek	Instalator wstrzymuje proces i sugeruje (albo pomaga zainstalować) wymagane składowe	Brak	Dowolne oprogramowanie instalowane na komputerze



**Rys. 7.10.** Wizualizacja sprawdzania aktualizacji ze standardowymi funkcjami

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Poprawność działania opcji automatycznej aktualizacji	Aplikacja podczas uruchamiania próbuje wyszukać automatycznie aktualizację	Brak	Dowolne oprogramowanie z opcją automatycznej aktualizacji
2	Możliwość decydowania o instalacji aktualizacji	W razie dostępności aktualizacji pojawia się komunikat o nowszej wersji aplikacji i możliwości jej pobrania	Brak	Dowolne oprogramowanie z opcją automatycznej aktualizacji
3	Sprawdzenie wyłączonej opcji automatycznej aktualizacji	Aplikacja nie wyszukuje aktualizacji i nie próbuje się aktualizować	Brak	Dowolne oprogramowanie z opcją automatycznej aktualizacji

4	Sprawdzenie możliwości ręcznego wyszukania aktualizacji	Po wyborze opcji szukania aktualizacji następuje próba jej znalezienia	Brak	Dowolne oprogramowanie z opcją aktualizacji
5	Sprawdzenie procesu pobierania oraz instalacji aktualizacji	Aktualizacja zostanie pobrana i rozpoczyna się instalacja	Brak	Dowolne oprogramowanie z opcją automatycznej aktualizacji
6	Kontrola poprawności obsługi danych po instalacji	Dotychczasowe dane użytkownika nie zostaną uszkodzone po aktualizacji do najnowszej wersji	Brak	Dowolne oprogramowanie z opcją aktualizacji
7	Weryfikacja ręcznego pobrania i instalowania aktualizacji	Użytkownik ma możliwość ręcznie pobrać aktualizację i ją przeprowadzić	Brak	Dowolne oprogramowanie z opcją aktualizacji
8	Próba instalowania aktualizacji w wersji wcześniejszej niż aktualnie zainstalowana	Informacja o aktualnie zainstalowanej nowszej wersji aplikacji	Brak	Dowolne oprogramowanie z opcją aktualizacji



**Rys. 7.11.** Wizualizacja dezinstalatora ze standardowymi funkcjami

## Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Możliwość uruchomienia dezinstalatora	Rozpoczęcie dezinstalacji	Brak	Dowolne oprogramowanie z opcją dezinstalacji
2	Sprawdzenie działania przycisków Dalej i Wstecz po uruchomieniu dezinstalatora	Możliwość przejścia do poprzedniego/następnego okna dezinstalatora	Brak	Dowolne oprogramowanie z opcją dezinstalacji
3	Sprawdzenie działania przycisku Anuluj	Przerwanie działania dezinstalatora po uprzednim zaakceptowaniu komunikatu, np. „Czy na pewno przerwać dezinstalację?”	Brak	Dowolne oprogramowanie z opcją dezinstalacji
4	Możliwość pozostawienia pewnych plików np. konfiguracji, zapisanych danych itp.	Wymienione dane/pliki/foldery nie zostają usunięte	Brak	Dowolne oprogramowanie z opcją dezinstalacji
5	Sprawdzenie usunięcia folderów/plików/skrótów programu	Dezinstalator usuwa wszystkie foldery/skróty programu z dysku	Dla części oprogramowania pełne usunięcie danych następuje dopiero po restarcie systemu	Dowolne oprogramowanie z opcją dezinstalacji
6	Sprawdzenie usunięcia informacji z rejestru	Dezinstalator usuwa wszystkie wpisy programu z rejestru systemu operacyjnego	W części oprogramowania pełne usunięcie danych z rejestru następuje dopiero po restarcie systemu	Dowolne oprogramowanie z system operacyjnym z rejestrem i z opcją dezinstalacji
7	Sprawdzenie „odpisania” typu pliku od programu	System poprosi o wskazanie programu do otwarcia danego typu pliku zamiast wyszukiwania	Głównie dla aplikacji desktopowych	Dowolne oprogramowanie z opcją dezinstalacji

		odinstalowanego programu		
8	Weryfikacja informacji o nieusunięciu pewnych elementów	Pliki/foldery pozostawione na dysku zgadzają się z informacją podaną przez dezinstalator	Brak	Dowolne oprogramowanie z opcją dezinstalacji
9	Informacja o zakończeniu dezinstalacji	Wyświetlenie informacji o zakończeniu dezinstalacji	Brak	Dowolne oprogramowanie z opcją dezinstalacji
10	Instalowanie aplikacji po dezinstalacji	Możliwość zainstalowania po wcześniejszej jej dezinstalacji	Test ten sprawdzi, czy wcześniejszy proces dezinstalacji przeszedł poprawnie, a także czy pliki, które zdecydowaliśmy się zachować, mogą być użyte do ponownego zainstalowania oprogramowania	Dowolne oprogramowanie z opcją dezinstalacji

**Testowanie zakładania, aktywacji i usuwania konta w serwisach.** Jedną z najważniejszych funkcji online. Wiele serwisów wymaga od nas utworzenia konta, zanim otrzymamy dostęp do zawartości lub funkcjonalności. W ostatnich latach funkcja ta została zredukowana do uwierzytelnienia się za pomocą profili w serwisach społecznościowych i ogranicza się do przeklikania autoryzacji. Ciągłe jednak wielu użytkowników zakłada konta i autoryzuje się w aplikacjach (por. rys. 7.12).

Hand-drawn sketch of a web form titled "Zakładanie konta w serwisie TestX". The form contains three input fields labeled "Login:", "Hasło:", and "Powtórz hasło:". Below the fields is a button labeled "Utwórz".

**Rys. 7.12.** Wizualizacja zakładania nowego konta w serwisie

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Rejestracja z poprawnymi danymi	Użytkownik zostaje zarejestrowany	Brak	Dowolne oprogramowanie z opcją rejestracji
2	Rejestracja z domyślnymi ustawieniami pól (nieuzupełnione)	Komunikat o błędnie wypełnionych polach, użytkownik nie zostaje zarejestrowany	Brak	Dowolne oprogramowanie z opcją rejestracji
3	Pole login lub nazwa użytkownika wypełnione niezgodnie z regułami walidacji, gdy pozostałe	Komunikat o błędzie walidacji loginu (np. brak wypełnienia pola, za mała liczba znaków, za duża liczba znaków, niedozwolone symbole, login zajęty), użytkownik	Brak	Dowolne oprogramowanie z opcją rejestracji
3	wymagane pola wypełnione są poprawnie	nie zostaje zarejestrowany, a wprowadzone dane		

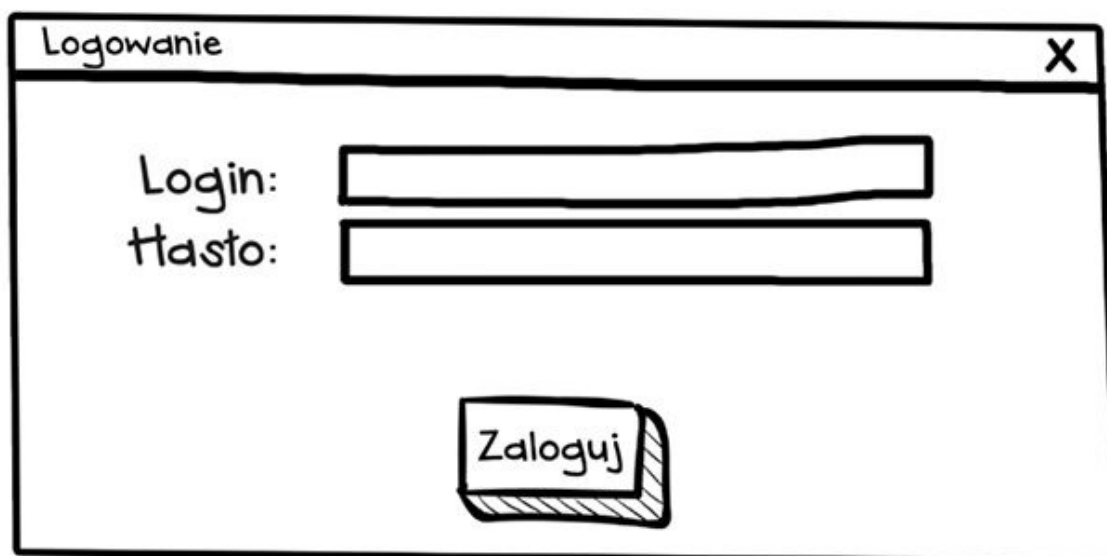
		pozostają w odpowiednich polach		
4	Pole hasło wypełnione niezgodnie z regułami walidacji, gdy pozostałe wymagane pola wypełnione są poprawnie	Komunikat o błędzie walidacji loginu, użytkownik nie zostaje zarejestrowany, a wprowadzone dane pozostają w odpowiednich polach	Brak	Dowolne oprogramowanie z opcją rejestracji
5	Pole powtórzenia hasła wypełnione inaczej niż pole hasła, gdy pozostałe wymagane pola wypełnione są poprawnie	Komunikat o błędzie walidacji powtórzenia hasła i różnicy względem pola hasło, użytkownik nie zostaje zarejestrowany, a wprowadzone dane pozostają w odpowiednich polach	Brak	Dowolne oprogramowanie z opcją rejestracji
6	Pole e-mail wypełnione niepoprawnie	Komunikat o błędnie wypełnionym polu (np. Brak adresu, adres w formacie innym niż xxx@xx.xx), użytkownik nie zostaje zarejestrowany, a wprowadzone dane pozostają w odpowiednich polach	Istnieje wiele standardów poprawnego adresu e-mail; przy raportowaniu defektów należy powołać się na jeden z nich	Dowolne oprogramowanie z opcją rejestracji
7	Pole CAPTCHA wypełnione niepoprawnie	Komunikat o błędnie wypełnionym polu CAPTCHA. Pojawia się nowy kod CAPTCHA, użytkownik nie zostaje zarejestrowany, a wprowadzone dane pozostają w odpowiednich polach	CAPTCHA ( <i>Completely Automated Public Turing test to tell Computers and Humans Apart</i> ) to mechanizm, za pomocą którego użytkownik musi wykazać, że jest człowiekiem,	Dowolne oprogramowanie z opcją rejestracji (z funkcją CAPTCHA)
7				



			a nie automatem, zazwyczaj przepisując tekst z obrazka	
8	Weryfikacja poprawności działania dwuetapowej weryfikacji użytkownika	Wymagany jest drugi krok w ramach weryfikacji do poprawnej autentykacji w systemie	Drugi etap weryfikacji to np. kod w SMS-ie, weryfikacja telefoniczna	Dowolne oprogramowanie z opcją rejestracji z dwuetapową weryfikacją
9	Brak zatwierdzenia regulaminu korzystania z aplikacji	Komunikat o konieczności akceptacji regulaminu, użytkownik nie zostaje zarejestrowany, a wprowadzone dane pozostają w odpowiednich polach	Brak	Dowolne oprogramowanie z opcją rejestracji
10	Otwarcie strony regulaminu	Wyświetlenie całego regulaminu w nowym oknie lub w oknie wyskakującym	Brak	Dowolne oprogramowanie z opcją rejestracji
11	Wysyłanie e-maila aktywacyjnego przy poprawnie uzupełnionych danych	Na podany podczas rejestracji adres e-mail zostaje wysłana wiadomość z linkiem aktywacyjnym dla konta	W niektórych systemach weryfikacja e-mailowa jest pozorowana i aktywacja w systemie następuje nawet bez kliknięcia linku aktywacyjnego. Dotyczy to zazwyczaj aplikacji, w których użytkownicy mocno stawiają na anonimowość	Dowolne oprogramowanie z opcją rejestracji wymagającej weryfikacji przez pocztę elektroniczną
12	Aktywacja konta linkiem z e-maila	Po kliknięciu linku przesłanego e-mailem użytkownik zostaje przekierowany na stronę z potwierdzeniem, że konto zostało aktywowane	Brak	Dowolne oprogramowanie z opcją rejestracji wymagającej weryfikacji przez pocztę elektroniczną
13			Brak	

	Usunięcie konta z serwisu	Konto zostaje usunięte, użytkownik zostaje wylogowany bez możliwości ponownego zalogowania się na dane konto		Dowolne oprogramowanie z możliwością usunięcia konta
--	---------------------------	--	--	--

**Logowanie, wylogowywanie i przypomnienie hasła w aplikacji.** Uwierzytelnianie się w aplikacjach stało się konieczne z wielu powodów. Pilnujemy bezpieczeństwa danych i funkcji, kontrolujemy kto, kiedy i dlaczego dokonuje zmian. Trudno sobie w dzisiejszych czasach wyobrazić poważną aplikację, w której nie ma opcji logowania (rys. 7.13).



**Rys. 7.13.** Wizualizacja okna logowania

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Logowanie za pomocą poprawnych danych	Użytkownik zostaje zalogowany	Brak	Dowolne oprogramowanie z opcją logowania
2	Próba zalogowania się	Komunikat o niewypełnieniu	Brak	Dowolne oprogramowanie

	bez wypełnienia wymaganych pól	pól Login i Hasło, użytkownik nie zostaje zalogowany		z opcją logowania
3	Zalogowanie z użyciem loginu nieistniejącego w systemie	Komunikat o braku możliwości zalogowania się z powodu niepoprawnego loginu lub hasła	Brak	Dowolne oprogramowanie z opcją logowania
4	Zalogowanie z użyciem loginu istniejącego w systemie i bez podania hasła	Komunikat o braku wypełnienia pola hasła	Brak	Dowolne oprogramowanie z opcją logowania
5	Zalogowanie z użyciem loginu istniejącego w systemie i z podaniem błędnego hasła	Komunikat o braku możliwości zalogowania się z powodu niepoprawnego loginu lub hasła	Brak	Dowolne oprogramowanie z opcją logowania
6	Pamiętanie ustawień o automatycznym zalogowaniu użytkownika (lub niewylogowaniu)	Użytkownik zostaje automatycznie zalogowany po ponownym uruchomieniu aplikacji	Brak	Dowolne oprogramowanie z opcją logowania (z opcją nie wylogowywania po zamknięciu aplikacji)
7	Blokada możliwości dalszego logowania po przekroczeniu dozwolonej ilości prób niepoprawnego logowania	Blokada możliwości logowania z podpowiedzią o innych możliwościach zalogowania się np. Przez przypomnienie hasła	Brak	Dowolne oprogramowanie z opcją logowania i z opcją ograniczonej liczby prób logowania
8	Próba odzyskania zapomnianego hasła przez wpisanie adresu e-	Wysłanie użytkownikowi wiadomości informującej	Brak	Dowolne oprogramowanie z opcją logowania i z opcją

	mail przypisanego do konta w aplikacji	o procedurze zmiany hasła		przypomnienia hasła
9	Próba odzyskania zapomnianego hasła przez wpisanie adresu e-mail nieprzypisanego do konta w aplikacji	Informacja o wysłaniu użytkownikowi wiadomości z procedurą zmiany hasła	Brak	Dowolne oprogramowanie z opcją logowania i z opcją przypomnienia hasła
10	Wylogowanie użytkownika z aplikacji	Użytkownik jest wylogowany	W przypadku aplikacji webowej mimo użycia strzałek powrotu w przeglądarce użytkownik nie może wrócić do opcji dostępnych dla zalogowanych	Dowolne oprogramowanie z opcją logowania
11	Automatyczne wylogowanie po określonym czasie bezczynności	Użytkownik zostaje wylogowany i zostaje wyświetlony komunikat o wylogowaniu z powodu bezczynności	Brak	Dowolne oprogramowanie z opcją logowania i z opcją automatycznego wylogowania po danym czasie bezczynności

**Zmiana ustawień konta.** We wszystkich systemach, gdzie istnieje możliwość dodawania, edycji i usuwania danych, stosujemy testy typu CRUD. Ten angielskojęzyczny skrót można rozszyfrować w następujący sposób:

C – stwórz (*create*), np. dodaj użytkownika,  
R – przeczytaj (*read*), np. przeglądaj dane użytkownika,  
U – zaktualizuj (*update*), np. edytuj i zapisz dane użytkownika,  
D – usuń (*delete*), np. usuń użytkownika z systemu.

Są to więc typowe testy związane z operacjami na danych i będą

... miały szerokie zastosowanie w większości systemów informatycznych.

Konto może mieć czasami wiele ustawień, o których należy pamiętać. W następnych podrozdziałach prezentuję kilka przykładowych testów związanych z edycją danych.

## Zmiana ustawień

**Rys. 7.14.** Wizualizacja dla zmiany ustawień w aplikacji

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Definiowanie ustawień wysyłania powiadomień	Aplikacja wysyła odpowiednie powiadomienie na adres e-mail użytkownika zgodnie z ustawieniami	Istnieje wiele opcji na wysyłanie powiadomień (w zależności od aplikacji), ich częstotliwości czy treści	Oprogramowanie z opcją wysyłania i konfigurowania powiadomień
2	Zmiana adresu e-mail dla konta	Aplikacja wysyła e-mail potwierdzający na nowy adres użytkownika	Brak	Dowolne oprogramowanie z możliwością zmiany adresu email
3	Zmiana hasła konta z podaniem starego hasła	Możliwość logowania tylko za pomocą nowego hasła	Brak	Dowolne oprogramowanie z możliwością zmiany hasła
4	Opcja ukrywanie informacji o zalogowaniu użytkownika w aplikacji	Inni użytkownicy aplikacji nie widzą, że użytkownik jest zalogowany	Mogą to być inne opcje widoczności lub statusy definiowane przez użytkownika, np. Zajęty	Oprogramowanie pokazujące zalogowanych użytkowników

**Edycja profilu użytkownika.** Gdy już utworzyliśmy sobie konto, czas na edycję danych w profilu (rys. 7.15). Również tutaj pojawi się zestaw testów dla CRUD. Mnogość ustawień i danych w koncie zmusza nas do utrzymania tych testów na wysokim poziomie ogólności.



**Rys. 7.15.** Wizualizacja edycji profilu użytkownika

#### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Weryfikacja możliwości zmiany danych w profilu, np. nr telefonu	Zmiana danych zakończona powodzeniem	Część zmian może wymagać dodatkowego kroku weryfikacji, np. potwierdzenia danych przez kliknięcie linku w e-mailu lub podanie starego hasła przy zmianie na nowe	Oprogramowanie z opcją zmiany danych w koncie
2	Weryfikacja możliwości zmiany danych w profilu na niepoprawne	Zmiana danych nie zakończona powodzeniem	Brak	Oprogramowanie z opcją zmiany danych w koncie

**Wyszukiwanie.** Wyszukiwarka (rys. 7.16) to podstawowy element wielu stron internetowych, aplikacji desktopowych i mobilnych, a dla niektórych aplikacji jest główną funkcją.



**Rys. 7.16.** Wizualizacja wyszukiwarki

Dobre testowanie wyszukiwarki polega na zrozumieniu jej działania. Wyszukiwarki rzadko przeprowadzają wyszukanie w czasie rzeczywistym. Zazwyczaj wyszukanie opiera się na już zaindeksowanych hasłach lub całych stronach internetowych. Mówimy więc o mniej lub bardziej rozbudowanej bazie danych, do których należy skonstruować odpowiednie zapytanie oparte na przygotowanym algorytmie. Niektóre wyszukiwarki stosują rankingi wyszukiwanych haseł, wskazując te o najbliższym pokrewieństwie względem zapytania. Mało kto wie, że algorytmy wyszukiwarek potrafią się uczyć. Wyszukując pasujące nam hasło i otwierając jedną ze wskazanych stron, możemy podnieść pozycję danej strony w rankingu.

Aby się upewnić, że liczba zwróconych w interfejsie użytkownika wyników zapytania jest poprawna, można również skonstruować zapytanie bazodanowe o takiej samej treści i porównać, czy ilość rekordów wyszukanych jest zgodna z liczbą rekordów w bazie.

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Wyszukiwanie wg pojedynczego ciągu znaków występującego na stronie	Aplikacja wyświetla wszystkie miejsca, w których znajduje się podany ciąg	Mechanizm wyszukiwania zazwyczaj nie jest szczegółowo opisany i testy mają pomóc zrozumieć jego działanie	Dowolne oprogramowanie z opcją wyszukiwania
2	Wyszukiwanie bez wypełnienia	1. Komunikat o braku kryteriów wyszukiwania (lub)	Wariantowość bierze się z wielu	Dowolne oprogramowanie

	pola z szukanym ciągiem	2. Nieuruchomienie wyszukiwania (lub) 3. Wyszukiwanie domyślnych stron (lub) 4. Wyświetlenie wszystkich elementów	możliwości obsłużenia takiego zdarzenia	z opcją wyszukiwania
3	Wyszukiwanie wg kryteriów niezwracających wyników	Komunikat o braku wyników wyszukiwania	Przykładowo poprzez wyszukanie ciągu znaków, które nie znajdują się na żadnej podstronie aplikacji internetowej	Dowolne oprogramowanie z opcją wyszukiwania
4	Wyszukiwanie wg treści z opcją na zwracanie uwagi na wielkość znaków	Aplikacja przy wyszukiwaniu zwraca uwagę na wielkość liter treści podanych w kryteriach i prezentuje tylko wyniki, które mają zgodną wielkość liter i treść z kryterium	Gdzie opcja wyszukiwania z uwzględnieniem wielkości znaków dostępna	Dowolne oprogramowanie z opcją wyszukiwania, zazwyczaj dotyczy edytorów tekstu lub oprogramowania desktopowego
5	Wyszukiwanie wg przynajmniej dwóch oddzielnych ciągów znaków (operator logiczny „lub” – OR)	Aplikacja prezentuje miejsca w systemie, w których znajdują się ciągi, wyniki posortowane są wg liczby ciągów znajdujących się w danym miejscu (wszystkie ciągi, brak jednego z ciągów, brak dwóch z ciągów, ..., pojedyncze ciągi)	Funkcja taka nie jest często spotykana w systemach masowych	Dowolne oprogramowanie z opcją wyszukiwania, zazwyczaj dotyczy oprogramowania desktopowego lub dużych systemów bazodanowych
6	Wyszukiwanie wg przynajmniej dwóch oddzielonych ciągów znaków (operator	Aplikacja prezentuje tylko miejsca w systemie, w których znajdują się wszystkie ciągi wypisane w kryterium	Funkcjonalność taka nie jest często spotykana w systemach masowych	Dowolne oprogramowanie z opcją wyszukiwania, zazwyczaj dotyczy oprogramowania



	logiczny „i” – AND)			desktopowego lub dużych systemów bazodanowych
7	Wyszukiwanie wg przynajmniej dwóch oddzielonych ciągów znaków (wyszukiwanie frazy)	Aplikacja prezentuje tylko miejsca w systemie, w których znajdują się wszystkie ciągi i w takiej samej kolejności, w jakiej zostały wpisane w kryterium wyszukiwania	Brak	Dowolne oprogramowanie z opcją wyszukiwania
8	Wyszukiwanie z błędnie podanymi kryteriami	Komunikat o rodzaju błędu, np. za mała liczba znaków, za duża liczba znaków, znaki niedozwolone w wyszukiwaniu	Brak	Dowolne oprogramowanie z opcją wyszukiwania
9	Wyszukiwanie plików z określonym rozszerzeniem	Aplikacja prezentuje wszystkie pliki z podanym w kryteriach rozszerzeniem	Gdzie istnieje możliwość wyszukiwania plików	Dowolne oprogramowanie z opcją wyszukiwania
10	Wyszukiwanie w określonej części systemu (np. Folderu)	Aplikacja prezentuje wyniki tylko z danej części systemu	Brak	Dowolne oprogramowanie z opcją wyszukiwania w konkretnej lokalizacji sprzętowej lub systemowej
11	Wyszukiwanie plików o określonej wielkości (lub z określonego zakresu)	Aplikacja prezentuje znalezione pliki tylko o podanej wielkości (zakresie)	Gdzie istnieje możliwość wyszukiwania plików	Dowolne oprogramowanie z opcją wyszukiwania
12	Wyszukiwanie plików graficznych wg rozdzielczości (lub zakresu)	Prezentowane są tylko pliki graficzne o określonej rozdzielczości (zakresie)	Gdzie istnieje możliwość wyszukiwania plików graficznych po ich atrybutach	Dowolne oprogramowanie z opcją wyszukiwania

--	--	--	--

**Sortowanie.** Osobnym tematem w ramach testowania wyszukiwania jest testowanie prezentacji informacji, sortowania (rys. 7.17) i stronicowania wyników. Ten temat jest szerszy od zakresu samego wyszukiwania, więc został opisany oddzielnie.

Zaprezentowane dane muszą zostać odpowiednio posortowane. Sortowanie może wynikać z logiki samej aplikacji albo z dopasowania do wyników np. wyszukiwania. Czasami dane będą sortowane w oparciu o dostępne funkcje sortowania, czasami zmiana kolejności sortowania nie będzie możliwa.

Email ▼▲	Imię▼▲	Nazwisko ▼▲	Status ▼▲
jan.kowal@wp.pl	Jan	Kowal	Aktywny
jan.nowak@wp.pl	Jan	Nowak	Zablokowany

**Rys. 7.17.** Wizualizacja dla funkcji sortowania

#### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Sortowanie zgodne z wewnętrznymi regułami prezentacji danych	Aplikacja wyświetla dane ułożone zgodnie ze zdefiniowanymi regułami	Brak	Dowolne oprogramowanie prezentujące zbiór danych
2	Sortowanie zgodne z dostępnymi funkcjami sortowania	Dane prezentowane są zgodnie z użytą funkcją sortowania	Dotyczy zazwyczaj prezentacji tabelarycznej, z możliwością sortowania po odpowiednich danych w kolumnach, czasami reguły sortowania wynikają z wcześniej dokonanych wyborów, np. „Pokaż najnowsze na początku”	Dowolne oprogramowanie prezentujące zbiór danych

**Stronicowanie** (rys. 7.18) pojawia się, jeśli istnieją ograniczenia liczby wyświetlanych rekordów. Takie ograniczenia zawsze powinny istnieć, ponieważ próba wyświetlenia „wszystkich” rekordów może zająć bardzo dużo czasu.

Wraz ze stronicowaniem zazwyczaj pojawia się nawigacja po stronach – od pierwszej po ostatnią.



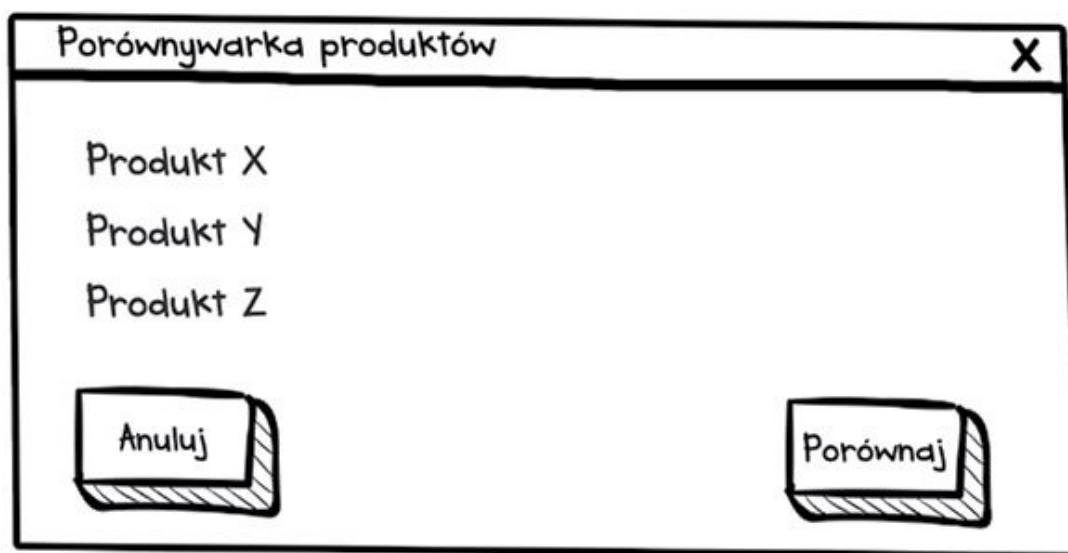
**Rys. 7.18.** Wizualizacja dla stronicowania

#### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Weryfikacja poprawności stronicowania	Stronicowanie pojawia się tam, gdzie mamy więcej rekordów niż zdefiniowanych rekordów do wyświetlania na pojedynczej stronie	Może to oznaczać, że opcje stronicowania są niedostępne lub nieaktywne, jeśli wyświetlono wszystkie rekordy	Dowolne oprogramowanie z funkcją stronicowania rekordów
2	Liczba rekordów na stronę stronicowania	Liczba rekordów jest zgodna ze zdefiniowaną regułą	Zazwyczaj jest to 10, 20, 50 lub inna wielokrotność dziesiątki na stronę; wyjątek stanowi przypadek, kiedy rekordów jest mniej niż limit na stronę	Dowolne oprogramowanie z funkcją stronicowania rekordów
3	Nawigacja po stronach stronicowania	Jest możliwe przejście na dowolną podstronę stronicowania	W większości aplikacji podstrony będą opisane jako 1, 2, 3 itd. Czasami pojawi się dodatkowo Pierwsza, Ostatnia lub/oraz Poprzednia, Następna	Dowolne oprogramowanie z funkcją stronicowania rekordów

**Porównywanie** stało się w ostatnim czasie bardzo ważną funkcją wielu aplikacji webowych. Porównuje się ceny, parametry czy szkoły.

Zestawienie dwóch lub więcej elementów ma ułatwić „konsumentowi” wybór i podjęcie decyzji. Przykładowe testy porównywarki (rys. 7.19) mogą również odwoływać się do wszelkiego rodzaju zestawień tabelarycznych, w których pokazuje się cechy różnych przedmiotów.



**Rys. 7.19.** Wizualizacja dla porównywarki

#### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Porównanie więcej niż jednego produktu	Otwarcie strony, na której przedstawione są obok siebie produkty wraz ze szczegółami	Porównanie pojedynczego produktu powinno również być możliwe z opcją dodania kolejnych przedmiotów do porównania	Dowolne oprogramowanie z funkcją porównywania (zazwyczaj aplikacje internetowe i mobilne)
2	Próba porównania przedmiotów bez ich wskazania (np. naciśnięcie przycisku Porównaj bez zaznaczenia elementów do porównania)	Komunikat o niewybraniu przedmiotów do porównania	Niektóre aplikacje oferują pusty ekran, z możliwością wybrania produktów do porównania	Oprogramowanie z funkcją porównywania (zazwyczaj aplikacje internetowe i mobilne)

3	Dodanie kolejnych przedmiotów do porównania	Przeniesienie na stronę porównywania z uwzględnieniem dodatkowego przedmiotu, który został właśnie wybrany	Możliwa opcja dalszego dodawania elementów do porównywania	Oprogramowanie z funkcją porównywania (zazwyczaj aplikacje internetowe i mobilne)
4	Usuwanie przedmiotów ze strony porównania	Usunięcie przedmiotu z listy przedmiotów porównywanych i wyświetlenie porównania pozostałych przedmiotów	Brak	Oprogramowanie z funkcją porównywania (zazwyczaj aplikacje internetowe i mobilne)

**Administrowanie użytkownikami.** Administrator to najważniejszy użytkownik każdego systemu, a testy weryfikujące możliwość wykonywania przez niego działań są podstawowe dla sprawdzenia poprawności funkcjonowania aplikacji.

Każdy system powinien mieć administratora, który ma uprawnienia do czytania, dodawania, edycji i usuwania danych (rys. 7.20). Przykładowo w systemach CMS (*content management system*) mogą to być także użytkownicy.

Panel administracyjny

X

Użytkownicy

Dodaj użytkownika

szukana fraza

status

Wszystkie ▼

wyniki

10 ▼

Filtruj

email	imię	nazwisko	status	akcje
jan.kowal@wp.pl	Jan	Kowal	Aktywny	<div>Edytuj</div> <div>Blokuj</div> <div>Usuń</div>
jan.nowak@wp.pl	Jan	Nowak	Zablokowany	

Rys. 7.20. Wizualizacja dla funkcji administracyjnych

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Możliwość przeglądania danych np. przez szukanie/filtrowanie	Wyświetlane wyniki są zgodne z zapytaniem lub ustawionym filtrem	Mocna zależność od formatu prezentacji danych	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
2	Sprawdzenie możliwości dodania nowych danych	Dane zostały dodane	Brak	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
3	Sprawdzenie możliwości edytowania danych	Dane zostały wyedytowane i zapisane jako nowe	Brak	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
4	Sprawdzenie możliwości usunięcia danych	Dane zostały usunięte z systemu po	Brak	Oprogramowanie (zazwyczaj bazodanowe)

		potwierdzeniu przez administratora		z użytkownikiem administrator
5	Możliwości nadawania uprawnień użytkownikowi	Użytkownik o danym poziomie autoryzacji ma możliwość wykonania tylko działań na danym poziomie autoryzacji	Uprawnienia w aplikacjach są zazwyczaj związane z konkretnymi rolami, np. kierownik ma prawo przeglądać wszystkie dane i generować raporty; czasami zdarza się, że uprawnienia można przyznawać zgodnie z rozbudowaną macierzą, co znacząco zwiększa liczbę testów do przeprowadzenia	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
6	Sprawdzenie możliwości odebrania uprawnień użytkownikowi	Użytkownik traci możliwość wykonywania działań dla danego poziomu autoryzacji	Testy zbliżone do powyższych	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
7	Weryfikacja blokowania użytkownika na określony czas lub permanentnie	Użytkownik traci możliwość wykonywania działań dla danego poziomu autoryzacji w zdefiniowanym czasie	Funkcja zazwyczaj dostępna w aplikacjach internetowych	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
8	Weryfikacja odblokowania użytkownika po jego zablokowaniu	Użytkownik odzyskuje możliwość wykonywania działań dla danego poziomu autoryzacji	Funkcja zazwyczaj dostępna w aplikacjach internetowych	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
9	Weryfikacja możliwości zresetowania hasła użytkownika	Po wybraniu opcji resetowania hasła nowe hasło (lub prośba o jego ustawienie) jest	Brak	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator

		wysłane do użytkownika		
10	Sprawdzenie możliwości tworzenia grup dla danych lub dla użytkowników	Utworzono grupy, do których można przypisać dane lub użytkowników	Funkcja grupowania użytkowników pojawia się przy konieczności nadania szczególnych uprawnień. Nie należy mylić grupowania z definiowaniem roli; role mają zazwyczaj zasięg globalny, a grupy raczej zasięg lokalny	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
11	Weryfikacja możliwości przypisania danych lub użytkownika do grupy	Dane lub użytkownicy są zgrupowani wg ustalonych kryteriów	Brak	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
12	Weryfikacja możliwości „odpięcia” danych lub użytkownika od grupy	Dane lub użytkownik nie jest częścią danej grupy	Brak	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator
13	Sprawdzenie poprawności wykonywania operacji (np. usunięcia) na wielu danych	Poprawne wykonanie operacji na wielu danych	Dotyczy opcji Zaznacz i Wykonaj dla wielu	Oprogramowanie (zazwyczaj bazodanowe) z użytkownikiem administrator

Opisane tutaj funkcje administratora ograniczają się do podstawowych operacji na danych. Oczywiście jest to wycinek działań użytkownika o takim poziomie uprawnień. Do jego zadań będzie również należało:

- wykonywanie kopii zapasowej i przywracanie danych z kopii (gdy zajdzie taka konieczność),
- monitorowanie wydajności aplikacji i diagnozowanie potencjalnych problemów z wydajnością,
- monitorowanie aktywności użytkowników w aplikacji i na bazie danych w poszukiwaniu prób nieautoryzowanego dostępu do

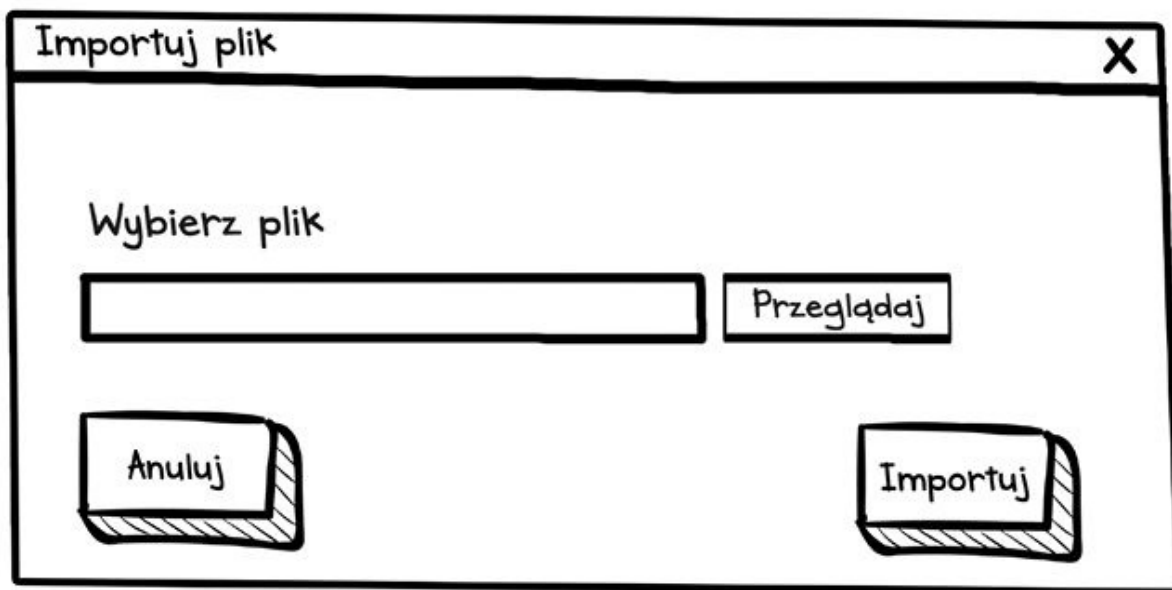


danych lub funkcji,

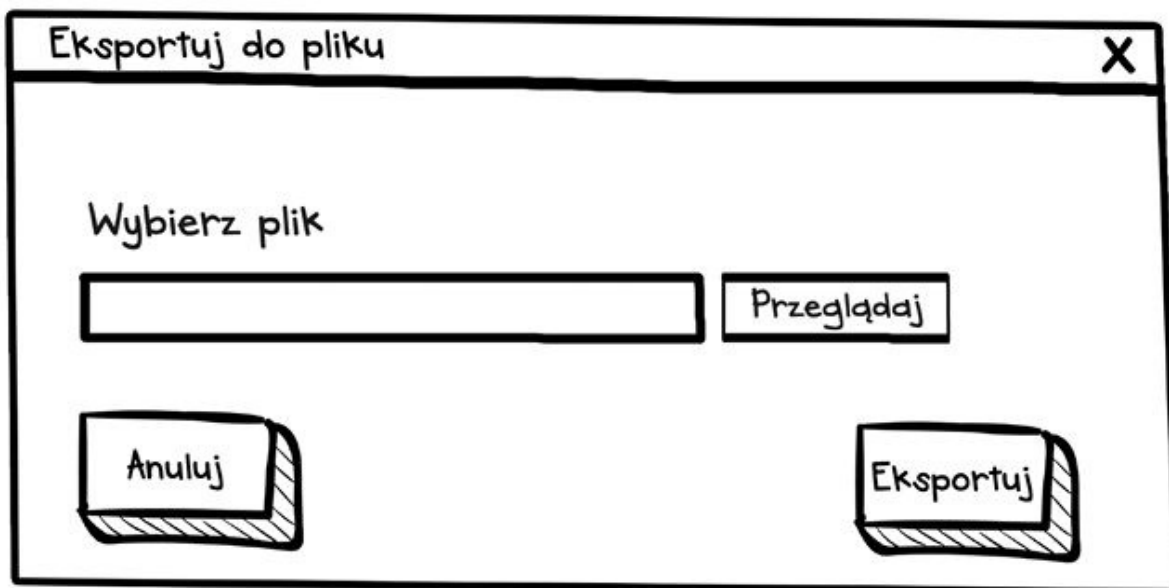
- konserwacja systemu polegająca na wgrywaniu aktualizacji składowych systemu,
- wprowadzanie zmian konfiguracyjnych składowych systemu,
- przeprowadzenie testów po zmianach konfiguracyjnych w systemie,
- wiele innych.

Działania te zależą ściśle od konstrukcji oprogramowania, dostępności interfejsów czy zastosowanego oprogramowania administracyjnego (w tym systemu operacyjnego) i trudno zdefiniować dla nich uniwersalne testy.

**Import/eksport do pliku.** Możliwość wykonywania kopii zapasowych, przenoszenia danych między programami to główne powody użycia funkcji importowania (rys. 7.21) i eksportowania (rys. 7.22) danych. Będzie to więc funkcja podstawowa dla wielu użytkowników.



Rys. 7.21. Wizualizacja dla importu plików



Rys. 7.22. Wizualizacja dla eksportu do pliku

### Przykładowe testy

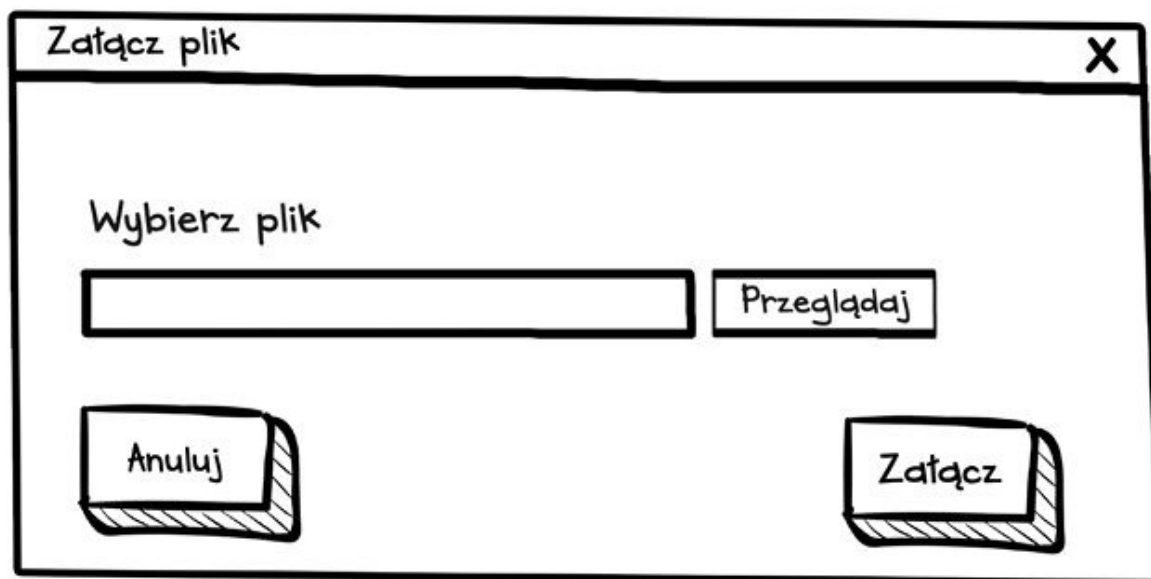
Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Próba importu poprawnego pliku	Poprawne importowanie danych z pliku oraz komunikat o importowaniu pliku	Reguły importowania pliku będą różne dla różnego typu aplikacji	Oprogramowanie z możliwością importu (zazwyczaj internetowe lub desktopowe)
2	Próba importu bez wskazania pliku (kliknięcie przycisku)	Komunikat o braku wskazania pliku do importu		Oprogramowanie z możliwością importu (zazwyczaj internetowe lub desktopowe)
3	Próba importu pliku o niewspieranym formacie	Komunikat o braku możliwości importowania danego formatu pliku	Konieczne podanie, dlaczego import zakończył się niepowodzeniem	Oprogramowanie z możliwością importu (zazwyczaj internetowe lub desktopowe)
4	Próba importu pliku	Komunikat o braku możliwości importu z powodu	Konieczne podanie, dlaczego import	Oprogramowanie z możliwością

	zawierającego nieprawidłowe dane (lub uszkodzonego)	niepoprawności danych (lub uszkodzenia pliku)	zakończył się niepowodzeniem	importu (zazwyczaj internetowe lub desktopowe)
5	Próba importu pliku o rozmiarze przekraczającym limity	Komunikat o braku możliwości importowania pliku o danym rozmiarze	Konieczne podanie, dlaczego import zakończył się niepowodzeniem	Oprogramowanie z możliwością importu (zazwyczaj internetowe lub desktopowe)
6	Eksport do pliku (np. do wybranego formatu)	Plik został prawidłowo eksportowany i zapisany	Brak	Oprogramowanie z możliwością importu (zazwyczaj internetowe lub desktopowe)
7	Sprawdzenie poprawności działania prawidłowo eksportowanego pliku	Plik po ponownym otwarciu powinien mieć zawartość zgodną z plikiem źródłowym	Mnogość metod weryfikacji w zależności od mechanizmu działania eksportu np. przez import lub też przez otwarcie w innej aplikacji	Oprogramowanie z możliwością importu (zazwyczaj internetowe lub desktopowe)
8	Sprawdzenie możliwości dokonania eksportu z przypisaniem nazwy już istniejącego pliku	Wyświetlenie komunikatu informującego o tym, że plik o podanej nazwie już istnieje w określonej lokalizacji i o konieczności wybrania innej nazwy; alternatywnie pytanie o możliwość nadpisania istniejącego pliku	Brak	Oprogramowanie z możliwością importu (zazwyczaj internetowe lub desktopowe)
9	Testowanie możliwości anulowania operacji importu (lub eksportu) w trakcie jej wykonywania	Operacja zostaje przerwana, żaden plik nie zostaje importowany (lub utworzony)	Tylko tam, gdzie taka funkcja jest dostępna	Oprogramowanie z możliwością importu (zazwyczaj internetowe lub desktopowe)

Niektóre inne ciekawe testy negatywne i dane testowe dla importu i eksportu plików:

- długa nazwa, ponad 255 znaków (dla importu i eksportu),
- znaki specjalne w nazwie pliku, np. spacja \* ? / \ | < > , . ( ) [ ] { } ; : ' “ ! @ # \$ % ^ & (dla importu i eksportu),
- próba podania ścieżki do nieistniejącego pliku (dla importu oraz eksportu),
- brak miejsca do zapisu na dysku (głównie dla eksportu),
- próba eksportu do pliku zabezpieczonego przed nadpisywaniem,
- próba importu pliku z serwera lub ze zdalnej maszyny.

**Dodawanie plików do aplikacji.** Od klientów pocztowych po systemy wymiany plików bardzo dużo aplikacji desktopowych i webowych operuje na plikach. Testy dodawania plików (rys. 7.23) są w pewnym zakresie zbliżone do testów importu, ale funkcjonalnie różnią się od nich znacząco.



Rys. 7.23. Wizualizacja dla załączania pliku

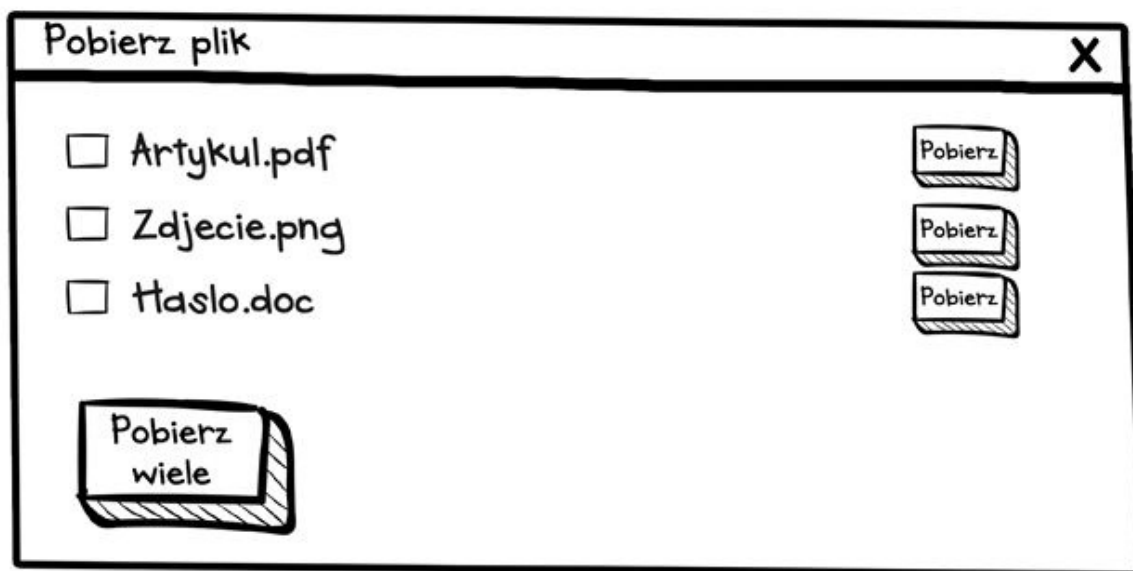
### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1				

	Poprawne dodanie z dysku pliku zgodnego z regułami walidacyjnymi	Dodanie pliku do aplikacji	Ogólnie ujęte reguły walidacyjne dotyczą szczególnych wytycznych odnośnie wagi czy formatu plików i zostały częściowo opisane przy funkcji importu i eksportu	Oprogramowanie z możliwością dodawania plików
2	Poprawne dodanie z adresu URL pliku zgodnego z regułami walidacyjnymi	Dodanie pliku do aplikacji	Brak	Oprogramowanie z możliwością dodawania plików (zazwyczaj internetowe)
3	Poprawne dodanie wielu plików zgodnych z regułami walidacyjnymi	Dodanie plików do aplikacji	Dla oprogramowania z opcją dodawania wielu plików za jednym razem	Oprogramowanie z możliwością dodawania plików
4	Próba dodania pliku niezgodnego z regułami walidacyjnymi	Nieudane dodanie pliku i wyświetlenie komunikatu o przyczynach problemu z dodaniem pliku	Reguły walidacyjne, np. rozmiar, format	Oprogramowanie z możliwością dodawania plików
5	Próba dodania pliku z niepoprawną ścieżką do pliku	Nieudane dodanie pliku i wyświetlenie komunikatu o niepoprawnej ścieżce lub braku wskazania pliku	Dotyczy zarówno dodawania plików z lokalizacji dyskowej jak i z lokalizacji zdalnej	Oprogramowanie z możliwością dodawania plików
6	Próba dodania poprawnego pliku i przerwanie połączenia z internetem	Plik nie zostaje dodany ( i jeśli możliwe, komunikat wskazujący na problemy z połączeniem)	Test ma na celu sprawdzenie zachowania oprogramowania w typowych warunkach funkcjonowania i problemów, jakie mogą wystąpić w środowisku	Oprogramowanie z możliwością dodawania plików (zazwyczaj internetowe)

**Dobieranie plików z aplikacji** Testy zbliżone do testów eksportu, ale

**Pobieranie plików z aplikacji.** Testy związane do testów eksportu, ale znacząco różne pod względem funkcjonalności. Pobranie plików (rys. 7.24) jest formą zapisu pliku w innej lokalizacji niż źródłowa. Dotyczy przede wszystkim oprogramowania internetowego (również intranetowego). Nie analizujemy tu jednak pobierania oprogramowania z wielu źródeł w tym samym czasie, a raczej pojedyncze źródło pliku.



**Rys. 7.24.** Wizualizacja dla pobierania pliku

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Poprawne pobranie pliku po wybraniu pojedynczego pliku z listy	Plik zostaje zapisany na dysku w ścieżce zdefiniowanej przez użytkownika	Zazwyczaj ścieżkę wskazuje ręcznie użytkownik (ale może być to również predefiniowany folder)	Oprogramowanie z możliwością pobierania plików (zazwyczaj internetowe)
2	Poprawne pobranie wielu plików z listy plików	Wszystkie pliki zostają zapisane na dysku w ścieżce zdefiniowanej przez użytkownika	Brak	Oprogramowanie z możliwością pobierania plików (zazwyczaj internetowe)
3				

	Przerwania połączenia podczas pobierania pliku	Plik nie zostaje pobrany (plik pobrany częściowo i uszkodzony nie jest zapisywany na dysku)	Czasami zapisany zostanie fragment pliku w oczekiwaniu na przywrócenie połączenia i pobranie reszty	Oprogramowanie z możliwością pobierania plików (zazwyczaj internetowe)
4	Sprawdzenie celowego wstrzymania pobierania pliku przez użytkownika	Pobieranie zostaje zatrzymane (w niektórych przypadkach użytkownik ma możliwość wznowienia wysyłania)	Brak	Oprogramowanie z możliwością pobierania plików (zazwyczaj internetowe)

**Dodawanie obrazków.** Choć dodawanie obrazków (rys. 7.25), np. jako załączników czy jako awatarów, należy do kategorii testów podobnej do CRUD czy do dodawania plików, to sam obrazek jest dość specyficzną daną testową. Należy mu więc poświęcić więcej uwagi.



**Rys. 7.25.** Wizualizacja dla przesyłania pliku

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1				

	Testowanie dodawania grafiki przez wskazanie lokalizacji pliku	Plik zostaje zapisany w zdalnej lokalizacji	Zazwyczaj podczas dodawania grafiki znaczenie ma nie to, w jakiej ścieżce się ona znajduje, ale co się w niej znajduje, dlatego użytkownik dostaje informację o wirtualnym miejscu umieszczenia, np. „Grafika została zapisana w folderze Twoje obrazy”	Oprogramowanie z możliwością dodawania grafik (zazwyczaj internetowe)
2	Testowanie dodawania grafiki przez alternatywne metody	Plik zostaje zapisany w zdalnej lokalizacji	Możliwość dodawania grafiki metodą przeciągania (przeciągnij z folderu i upuść w przeglądarce)	Oprogramowanie z możliwością dodawania grafik (zazwyczaj internetowe)
3	Testowanie poprawności wyświetlania domyślnej grafiki, kiedy nie dodano żadnej grafiki	Aplikacja prezentuje domyślną grafikę	Funkcja dostępna wszędzie tam, gdzie użytkownik może dodać wizualizację, np. do profilu awatar, ikonę itp.	Oprogramowanie z możliwością dodawania grafik (zazwyczaj internetowe)
4	Testowanie domyślnej grafiki, po dodaniu poprawnej grafiki i jej usunięciu	Aplikacja prezentuje domyślną grafikę	Jw.	Oprogramowanie z możliwością dodawania grafik (zazwyczaj internetowe)
5	Dodanie niepoprawnej grafiki	Wyświetlenie komunikatu o błędzie z informacją, dlaczego grafika nie została dodana	Walidacja, np. rozmiar pliku zbyt duży, nieakceptowalna rozdzielczość, nieobsługiwany format pliku itp.	Oprogramowanie z możliwością dodawania grafik (zazwyczaj internetowe)

**Komunikacja między użytkownikami jednej aplikacji.** Część aplikacji stworzona jest bezpośrednio do komunikacji między dwoma



użytkownikami rys. 7.26) lub większą ich liczbą, dla części z nich będzie to tylko jedna z dodatkowych funkcji.

**Rys. 7.26.** Wizualizacja dla funkcji komunikatora



Komunikacja jest procesem złożonym, z wieloma elementami użyteczności oraz dużą zależnością od połączenia. Łączność opisana tu dotyczy jedynie komunikacji tekstowej. Aplikacje mają jednak znacznie bardziej rozbudowane mechanizmy komunikacji, takie jak przesyłanie dźwięku w czasie rzeczywistym, plików do odsłuchania czy też wideopołączenia.

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Wysyłanie wiadomości do drugiego użytkownika aplikacji	Drugi użytkownik otrzymuje wiadomość	Brak	Oprogramowanie z wbudowaną komunikacją (zazwyczaj internetowe)
2	Próba wysyłania wiadomości do nieistniejącego (nieaktywnego,	Komunikat o braku możliwości dostarczenia wiadomości	Czasami komunikator może oferować opcję oczekiwania na	Oprogramowanie z wbudowaną komunikacją (zazwyczaj internetowe)

	zablokowanego itp.) użytkownika aplikacji		dostarczenie wiadomości	
3	Testowanie wysyłania wiadomości do grupy poprawnie zdefiniowanych użytkowników	Wszyscy użytkownicy otrzymują wiadomość	Brak	Oprogramowanie z wbudowaną komunikacją (zazwyczaj internetowe)
4	Logowanie wiadomości w systemie	Wiadomości są zapisywane w systemie na określony czas lub do ustalonej liczby wiadomości	Testy historii komunikatora	Oprogramowanie z wbudowaną komunikacją (zazwyczaj internetowe)
5	Blokowanie wiadomości od innego użytkownika	Użytkownik nie otrzymuje wiadomości od zablokowanego przez siebie użytkownika	Brak	Oprogramowanie z wbudowaną komunikacją (zazwyczaj internetowe)

**Koszyk (sklep internetowy).** Koszyk jest standardowym elementem sklepów. Na swój sposób stanowi pamięć podręczną użytkownika. Można powiedzieć, że jest to po prostu forma grupowania produktów, ze zliczaniem i podsumowaniem cen.

Koszyk	
 Produkt I	ilość: <input type="text" value="1"/> 1559 zł
 Produkt II	ilość: <input type="text" value="1"/> 46 zł
Łączna cena towarów	<input type="button" value="przelicz"/> 1605 zł
<input type="button" value="DALEJ"/>	

**Rys. 7.27.** Wizualizacja dla koszyka w sklepie internetowym

Musimy również rozdzielić funkcje, które udostępnia się użytkownikom w zależności od tego, czy są autoryzowani w systemie, czy też nie. Zalogowanie może pokazywać też szczególne uprawnienia użytkownika (np. do zniżek), co wpływa na kalkulację ceny w koszyku.

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Dodawanie produktu do koszyka	Element zostaje dodany do koszyka, następuje przeliczenie ceny	Należy rozważyć osobne testy dla użytkownika zalogowanego i niezalogowanego	Oprogramowanie z koszykiem (zazwyczaj internetowe)
2	Kontynuowanie zakupów po dodaniu elementów do koszyka	Użytkownik może kontynuować dodawanie przedmiotów do koszyka	Brak	Oprogramowanie z koszykiem (zazwyczaj internetowe)
3	Usuwanie produktów z koszyka	Z listy produktów w koszyku znika usunięty produkt, następuje przeliczenie ceny	Brak	Oprogramowanie z koszykiem (zazwyczaj internetowe)
4	Zmiana liczby przedmiotów w koszyku	Inkrementacja lub dekrementacja liczby przedmiotów w koszyku, następuje przeliczenie ceny	Brak	Oprogramowanie z koszykiem (zazwyczaj internetowe)
5	Usuwanie produktów z koszyka przez zmianę liczby przedmiotów z 1 na 0	„Wyzerowany” przedmiot nie znajduje się już w koszyku, następuje przeliczenie ceny	Gdy istnieje taka opcja	Oprogramowanie z koszykiem (zazwyczaj internetowe)
6	Określenie liczby przedmiotów przez podanie prawidłowych danych	Liczba przedmiotów zostaje określona, następuje przeliczenie ceny	Gdy istnieje taka opcja	Oprogramowanie z koszykiem (zazwyczaj internetowe)
7			Gdy istnieje taka opcja	

	Określanie liczby przedmiotów przez próbę podania nieprawidłowych danych	Komunikat o nieprawidłowej wartości w polu		Oprogramowanie z koszykiem (zazwyczaj internetowe)
8	Sprawdzanie możliwości zamawiania liczby produktów ponad liczbę dostępnych w magazynie	Informacja o przekroczeniu liczby produktów dostępnych w magazynie	Potrzebna wiedza o stanach magazynowych; test ściśle zależny od konstrukcji oprogramowania	Oprogramowanie z koszykiem (zazwyczaj internetowe)

**Czas** jest ważnym elementem wielu aplikacji. Aby dobrze przetestować działanie aplikacji w czasie i zależności czasowe, warto wiedzieć, skąd pobierana jest informacja o czasie dla aplikacji (np. z systemu operacyjnego). Testów tych użyjesz w aplikacjach z możliwością definiowania czasu (rys. 7.28) przez użytkownika lub też przez inne aplikacje. Można je również stosować w aplikacjach, których zachowanie będzie zależało od czasu wykonania operacji.

**Rys. 7.28.** Wizualizacja dla definiowania czasu w aplikacji

Podstawowe testy dla pojedynczego określania czasu to:

- czas z bardzo odległej przeszłości,

- czas współczesny, nieodległa przeszłość,
- teraz,
- nieodległa przyszłość,
- odległa przyszłość,
- testy formatu daty i innych ciągów w polu,
- testy zmodyfikowanego źródła określania czasu,

Taką koncepcję testowania danej testowej, jaką jest czas, możemy znaleźć między innymi w blogu TheTestEye.

Podstawowe testy dla dwóch zależnych danych typu czas wchodzących ze sobą w interakcje:

- dwie równe sobie dane typu czas (w przeszłości, teraźniejszości i przyszłości),
- czas z przeszłości/czas z przyszłości,
- czas z przeszłości/czas z teraźniejszości,
- czas z teraźniejszości/czas z przyszłości,
- czas z przyszłości/czas z odległej przyszłości,
- czas z odległej przeszłości/czas z przeszłości.

Ważną rolę w testowaniu dla danych typu czasu odegrają odpowiednio dobrane dane testowe, zwłaszcza w przypadku formularza wprowadzania danych. Podajemy tu, które testy warto przeprowadzić na polach z datą/kalendarzem.

## Lista testów

Lp.	Dane	Komentarz
1	Lata przestępne – sprawdzenie, czy są obsługiwane i nie powodują błędów w obliczeniach czasu	Brak
2	Miesiąc: 00 i 13 – sprawdzenie, czy są rozpoznawane jako błędy	Brak
3	Dzień: 00 i 32 dzień – sprawdzenie, czy są rozpoznawane jako błędy	Brak
4	Dzień 30 lutego – sprawdzenie, czy jest rozpoznawany jako błąd	Brak
5	Daty z innego wieku, np. 1890 – sprawdzenie, czy są obsługiwane i nie powodują błędów	Brak

	w obliczeniach czasu	
6	Różne formaty dat, np. amerykański – sprawdzenie, czy są obsługiwane i raportowane jako błąd lub poprawiane	Na przykład data 30 stycznia 2016 1/30/2015
7	Sprawdzenie, czy wybrana data z kalendarza jest poprawnie przekazywana do pól formularza	Brak
8	Sprawdzenie, czy walidacja poprawności danej typu czas działa z wyłączoną obsługą JavaScriptu	Jeśli aplikacja jest webowa i jeśli zakładamy, że aplikacja powinna działać w takiej konfiguracji
10	Sprawdzenie, czy po wprowadzeniu znaków niedozwolonych formularz zwróci błąd	Na przykład a0/1b/cc8c
11	Sprawdzenie, czy po wprowadzeniu dozwolonych znaków w złym formacie formularz zwróci błąd	Brak
12	Sprawdzenie, czy 1/1/11 jest akceptowane tak samo jak 01/01/2011	Ewentualnie poprawiany do odpowiedniego formatu
13	Sprawdzenie, czy nieznaczące spacje na początku i końcu daty są obcinane	Brak
14	Rok: 0001 oraz 0000 oraz 3000 – sprawdzenie, czy są raportowane jako błędy	Tam, gdzie nie są one obsługiwane
15	Sprawdzenie, czy domyślna wartość (jeśli jest) jest poprawna i nie powoduje błędów	Brak
16	Sprawdzenie poprawności działania dla danej testowej typu miesiąc wprowadzanych jako słowa, np. sierpień	Brak

Inne ciekawe testy pozytywne i negatywne oraz dane testowe dla czasu w systemach informatycznych.

### Lista testów

Lp.	Dane	Komentarz
1	Przekroczenie czasu	Zalogowania się w systemie
2	Różnica czasu między dwoma komunikującymi się maszynami	Brak
3	Zmiana daty	Przejsie z jednego dnia na następny

4	Przekraczanie strefy czasowej z automatycznym ustawianiem czasu w zależności od położenia	Na przykład w samolotach
5	Przesunięcia z czasu zimowego na letni i z letniego na zimowy	Brak
6	Resetowanie danej typu czas	Brak
7	Przesunięcie czasu do przodu lub do tyłu	W celu sprawdzenia zachowania się systemu

Istnieje wiele metod określania czasu w aplikacji. Najbardziej powszechne to wybór z listy lub z kalendarza (który jest bardziej złożoną listą). Istnieją rzadziej spotykane metody definiowania danej typu czas, np. pole tekstowe.

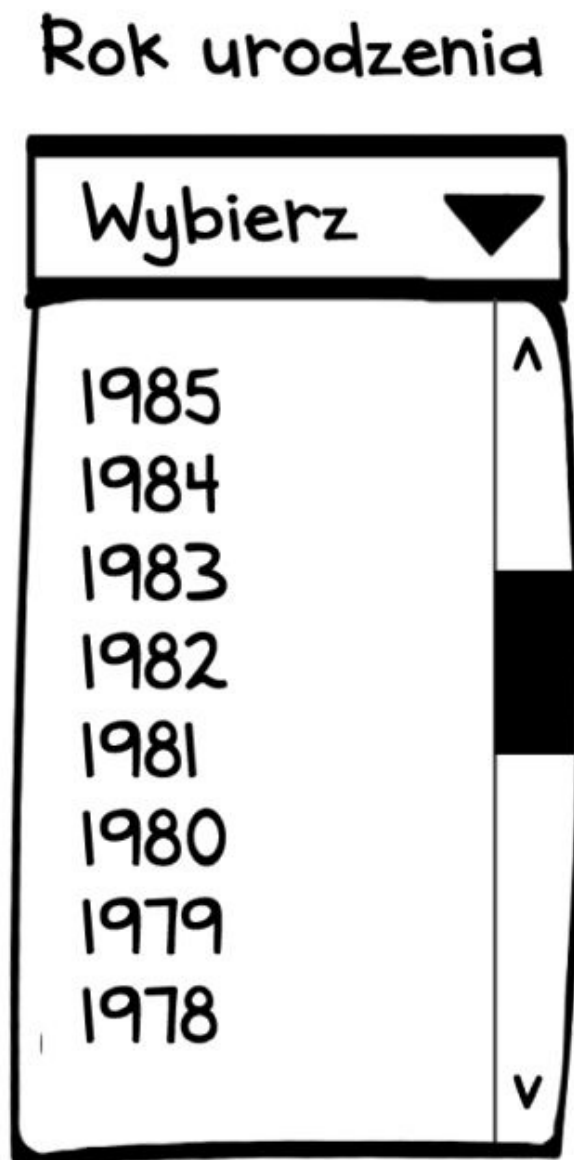
**Data/godzina wybierana z listy.** Projektowanie dobrych testów wymaga określenia warunków poprawności działania lub realistycznych założeń, np.:

- funkcja: określenie pojedynczej daty, np. roku urodzenia,
- wybór roku ze skończonej listy,
- wartości od 1913 do 2015; pierwsze pole: [Wybierz]
- pole nie jest wymagane.

Podstawowe testy, które należy przeprowadzić, to testy funkcji, sprawdzające jedynie poprawność działania elementu. Są to klasyczne testy tzw. klas równoważności, gdzie klasa jest zbiorem wartości wejściowych z przedziału obustronnie domkniętego  $\langle 1913; 2015 \rangle$ , do którego należy dołączyć dodatkowy element „wybierz”. Należy pamiętać o rozdzieleniu zbiorów wejściowych i wyjściowych. Zbiorem zdarzeń na wyjściu może być: wyświetlenie informacji o zapisaniu danych oraz wyświetlenie roku urodzenia w profilu, czy informacja o zapisaniu danych i niewyświetleniu roku w profilu, jeśli nie została wybrana żadna wartość ze zbioru dat. Zachowanie aplikacji będzie inne dla obu zbiorów, dlatego też należy uwzględnić przynajmniej dwa testy – po jednym na zbiór.

Rozszerzeniem zakresu testów będzie zweryfikowanie, czy aplikacja zachowuje się poprawnie na krawędziach danego przedziału, zwanych również granicami, gdzie 1913 i 2015 traktowane są jako wartości

graniczne. Testy te jednak w przypadku skończonej listy elementów (takiej jak podana lista lat) możemy potraktować jako niewiele wnoszące i mocno nadmiarowe.



**Rys. 7.29.** Wizualizacja dla wyboru danej typu czas z listy

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Przekazania daty ze środka	Data zapisuje się poprawnie	Brak	Oprogramowanie z wyborem daty



	listy, data z początku listy (1913) i data z końca listy (2015)	z odpowiednim komunikatem		z listy
2	Dokonanie wyboru wartości Wybierz i próba zapisania danych	Data zapisuje się poprawnie i nie wyświetla się już w profilu	Tylko po wcześniejszym założeniu, że data nie jest wymagana	Oprogramowanie z wyborem daty z listy
3	a. Wybranie daty z listy (dowolna), zapisanie, sprawdzenie, czy zapisała się poprawnie. b. Wybranie Wybierz, sprawdzenie poprawności zapisu. c. Ponowne wybranie daty z listy i zapisanie	Data wyświetlana ma być datą, która została wybrana w ostatnim kroku testu	Test kombinacji zdarzeń, w którym łączymy zdarzenie z testu 1 i testu 2, ma na celu ujawnienie błędów powiązanych z modyfikacją raz wprowadzonego roku oraz ze sprawdzeniem, czy aplikacja poprawnie przechowuje raz zdefiniowane dane	Brak

**Pojedyncza data/godzina wybierana z kalendarza.** Wybór pojedynczej wartości z kalendarza (rys. 7.30) jest standardem dla wielu aplikacji. Definiujemy datę urodzenia, wylotu, raportujemy czas pracy itd.

Założenia:

- funkcja: dokonanie wyboru daty z przyszłości,
- wybór daty tylko z kalendarza,
- daty z odległej przeszłości i przyszłości nie są akceptowane.



**Rys. 7.30.** Wizualizacja dla wyboru danej typu czas z kalendarza

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Wybór z kalendarza daty późniejszej od obecnej	Data zostaje zaakceptowana	Brak	Oprogramowanie z kalendarzem
2	Wybór dzisiejszej daty	Data nie zostaje zaakceptowana	Wybór dzisiejszej daty traktowany jest jako błąd	Oprogramowanie z kalendarzem
3	Wybór daty z przeszłości	Wybór niemożliwy lub data nie zostaje zaakceptowana	Wybór daty z przeszłości traktowany jest jako błąd	Oprogramowanie z kalendarzem
4	Wybór daty z odległej przeszłości	Wybór niemożliwy lub data nie zostaje zaakceptowana	Wybór takiej daty traktowany jest jako błąd	Oprogramowanie z kalendarzem
5	Wybór daty z odległej przyszłości	Wybór niemożliwy lub data nie zostaje zaakceptowana	Wybór takiej daty traktowany jest jako błąd	Oprogramowanie z kalendarzem

6	Próba wpisania daty z klawiatury i naciśnięcie Enter	Data nie zostaje zaakceptowana	Wprowadzanie daty z klawiatury traktowane jest jako błąd	Oprogramowanie z kalendarzem
---	--	--------------------------------	--	------------------------------

**Więcej niż jedna data/godzina wybierana z kalendarza.** W aplikacjach często występują zależności między dwiema datami (rys. 7.31). Rzadko, ale spotyka się również zależności między większą liczbą dat.

The image is a hand-drawn sketch of a software window titled "Wyszukaj według daty" (Search by date). The window has a standard title bar with a close button (X). Inside the window, there are two input fields. The first is labeled "Data od" (Date from) and the second is labeled "Data do" (Date to). Each input field is followed by a small calendar icon, indicating that dates should be selected from a calendar. At the bottom of the window, there are two buttons: "Anuluj" (Cancel) on the left and "Szukaj" (Search) on the right. The sketch is simple, using black lines on a white background.

**Rys. 7.31.** Wizualizacja dla wyboru dwóch danych typu czas z kalendarza

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Sprawdzenie poprawnych wartości z podaniem daty OD większej lub równej bieżącej dacie; podanie daty DO większej od OD	Wyszukanie informacji w zdefiniowanych datach	Brak	Oprogramowanie z relacją dwóch dat
2	Sprawdzenie rezerwacji niepoprawnej – podanie daty OD jako większej od daty DO	Informacja o braku możliwości wyszukania z odpowiednim komunikatem o błędzie	Brak	Oprogramowanie z relacją dwóch dat

3	Sprawdzenie rezerwacji poprawnej, gdzie data OD jest równa dacie DO	Wyszukanie informacji w określonej (pojedynczej) dacie	Brak	Oprogramowanie z relacją dwóch dat
---	---	--	------	------------------------------------

**Testowanie tekstowego pola czasu.** Podstawowym testem pola z możliwością podania daty będzie sprawdzenie, czy wiemy, w jakim formacie data ma być podana, a następnie czy data w poprawnym formacie jest akceptowana. Format daty może być następujący: RRRR/MM/DD, gdzie R – cyfry reprezentujące rok, M – cyfry reprezentujące miesiąc, D – cyfry reprezentujące dzień.

The image shows a hand-drawn sketch of a web form. The title bar of the form reads "Sprawdź, czy data istnieje" with a close button (X) on the right. Inside the form, there is a text input field labeled "Data" on the left. To the right of the input field is a small circle containing the letter "i", representing an information icon. At the bottom right of the form is a button labeled "Sprawdź".

**Rys. 7.32.** Wizualizacja dla definiowania danej typu czas jako tekstu

### Przykładowe testy

Lp.	Test	Oczekiwany rezultat	Komentarz	Cel
1	Sprawdzenie, czy podano datę w akceptowalnym formacie	Data zostaje zaakceptowana	Brak	Oprogramowanie z polem tekstowym do wprowadzania daty
2	Sprawdzenie, czy podano prawdziwą datę (z przyszłości)	Data nie powinna być przyjęta lub powinna zostać	Przykładowo: 2016-02-28, 2016.02.28, 20160228	Oprogramowanie z polem tekstowym do

	w nieakceptowalnym formacie	skonwertowana na format akceptowalny		wprowadzania daty
3	Podanie ciągu znaków niereprezentującego daty	Data nie powinna być przyjęta	Brak	Oprogramowanie z polem tekstowym do wprowadzania daty
4	Podanie daty w formacie zbliżonym do akceptowalnego	Data nie powinna być przyjęta lub powinna zostać skonwertowana na format akceptowalny	Przykładowo: rr/mm/dd	Oprogramowanie z polem tekstowym do wprowadzania daty
5	Podanie nieistniejącej daty w poprawnym formacie	Data nie powinna być przyjęta	Przykładowo: 2018/02/30	Oprogramowanie z polem tekstowym do wprowadzania daty
6	Próba wklejenia do pola niepoprawnej daty	Data nie powinna być przyjęta	Opcja wpisywania znaków innych niż cyfry jest niekiedy zablokowana, ale wklejenie może już nie być zablokowane. Przykładowe dane: bardzo długi string, 0, [spacja], znaki specjalne inne niż akceptowane	Oprogramowanie z polem tekstowym do wprowadzania daty

## Zadanie

Dla wszystkich wymienionych w tym rozdziale elementów, funkcji i funkcjonalności, procesów i logiki przeprowadź testy. W pierwszym kroku znajdź oprogramowanie zawierające dane funkcje (lub funkcję), a następnie określ, które z podanych testów będą adekwatne dla tej aplikacji. W przypadku wykluczenia testów opisz, dlaczego zostają pominięte.

- Spróbuj określić testy, których twoim zdaniem brakuje, a które mogłyby ujawnić kolejne defekty.

- Spróbuj za pomocą testów sprawdzić poprawność działania funkcji lub określić, że nie działa, przez ujawnienie występowania w niej poważnych defektów.
  - Sporządź raport z wynikami swojej pracy, określający obszary działające oraz informujący o znalezionych defektach.
- 

## 7.5. Raportowanie

Raportowanie jest podstawowym zadaniem każdego pracownika. Raport służy do kontroli jakości naszej pracy, a w przypadku testera oprogramowania zyskuje dodatkowy wymiar, ponieważ przede wszystkim ma pokazać jakość aplikacji. Jakość ta może być zaprezentowana wyłącznie w postaci raportów defektów, w których informacja o jakości ukryta jest w liczbie i krytyczności defektów. Jakość może być również raportowana w informacji przekazanej ustnie lub przyjmując postać dokumentu raportu.

### 7.5.1. Subiektywna ocena jakości oprogramowania

W pewnym momencie trwania projektu tester oprogramowania wie najwięcej o jakości oprogramowania. Chodzi o moment, gdy produkt jest zbyt duży, aby mógł go całościowo zrozumieć pojedynczy programista, a użytkownik nie rozpoczął jeszcze odbierać oprogramowania. Wszelkie informacje o jakości są wówczas bardzo pożądane. Mogą to być również informacje nie w pełni potwierdzone liczbami, a wynikające raczej z postrzegania jakości czy po prostu przeczucia.

Są firmy stawiające na otwartość w komunikacji, pytające testera o opinię na temat jakości oprogramowania i biorące jego zdanie pod uwagę. Pamiętając, że nasz głos ma zawsze mniejszą wagę niż głos sponsora projektu („Płacę, więc wymagam”) czy użytkownika („Wiem, czego potrzebuję”), nie powinniśmy go bronić niczym niepodległości. Czasami nawet, jeśli nie jesteśmy przekonani, że uczestnicy projektu mają rację, możemy powtórzyć za Sokratesem: „Wiem, że nic nie wiem”. Możemy uznać, że nie mamy wszystkich informacji pozwalających zdecydować, czy jakość oprogramowania jest wystarczająco dobra, by wypuścić je na rynek. Gdy przedstawiciele

projektu przyjdą do nas i zapytają o zdanie, powinniśmy je wyrazić z zastrzeżeniem, że to opinia subiektywna, która jedynie uzupełnia ocenę ilościową, wynikającą z pokrycia testami czy liczby defektów w oprogramowaniu.

Są również organizacje, zazwyczaj niedojrzałe, które będą nas rozliczać z każdej opinii o jakości oprogramowania. Musimy więc podchodzić do tematu bardzo delikatnie, żeby nie skończyć jako „ten, który mówił, że można wdrażać, i zobaczcie co się wydarzyło”. Nasze zdanie na temat jakości musi być połączeniem opinii obiektywnej (miar i metryk) i subiektywnej (wrażeń i odczuć).

### **7.5.2. Raport z testów**

Zakończenie testów wymaga właściwego podsumowania. Nie możemy się koncentrować na negatywnych aspektach, wskazując jedynie defekty. Musimy pokazać również to, co udało się zweryfikować i co działa w naszej opinii lub zgodnie z wymaganiami. Jeśli otrzymaliśmy zadanie do wykonania, to należy się z niego starannie rozliczyć. Co z tego, że naszym zdaniem wykonaliśmy je dobrze? Musimy pokazać efekt pracy i rozliczyć poświęcony na nią czas. Tak jak przy planowaniu pierwszy impuls przekazują kierownicy, tak przy raportowaniu główna informacja wychodzi od nas. Pojedyncze raporty są łączone w całość, by pokazać prawdziwy obraz oprogramowania.

Twoje raportowanie może być pasywne i ograniczać się do przedstawienia defektów i definiowania statusów dla uruchomienia testów (przeszły/nie przeszły/zablokowane/nieuruchomione). Narzędzi, które zasilasz takimi danymi, używa następnie kierownik do definiowania zweryfikowanej jakości i określania gotowości oprogramowania do wydania. Taki model jest właściwie zerojedynkowy i nie zostawia miejsca na subiektywną opinię czy głębszą analizę oprogramowania. Zupełnie inaczej wygląda to np. podczas testowania eksploracyjnego, gdzie raport daje większe pole do popisu. Oczywiście dodamy tam defekty i spełnienie kryterium powodzenia testów, ale i coś więcej. Wymienimy w raporcie miejsca, którym warto poświęcić dodatkową uwagę. Wskażemy podejrzane zachowania aplikacji – choć nie są to defekty, warto im się przyjrzeć dokładniej. Poinformujemy również, czego nie wiemy. Na przykład aplikacja zachowuje się w dany sposób, a wymagania nie są wystarczająco precyzyjne. Możemy

spokojnie uznać, że test kończy się powodzeniem, ale należy dopytać użytkownika, czy jego zdaniem aplikacja rzeczywiście funkcjonuje poprawnie. Oczywiście oba podejścia możemy łączyć i nie ograniczać się do przekazania suchych faktów, ale uzupełnić je w osobnym dokumencie czy narzędziu o informacje dodatkowe.

W raporcie zawsze ważne będą miary i metryki, m.in.:

1. Ile czasu zajęły ci testy?
2. Ile testów udało ci się wykonać?
3. Ile defektów zostało znalezionych i ile z nich jest krytycznych lub ważnych?
4. W ilu środowiskach udało się uruchomić testy?
5. Jakie było pokrycie testowe wymagań czy kryteriów akceptacji?
6. Jaki procent ważnych funkcji działa poprawnie, a jaki wykazuje defekty?

Z metrykami nie można przesadzać, ale nie można ich również pomijać. Wykresy, słupki, tabele łatwiej przemówią do wyobraźni naszych kierowników niż czysty tekst. Szybciej również przekażą informację o jakości testowanego oprogramowania i pracy, którą poświęciliśmy na jego weryfikację.

Raport w wielu przypadkach jest naszym pożegnaniem z projektem albo jego częścią. Chcemy zostawić po sobie dobre wrażenie, a raport pomaga nam to osiągnąć. O testowaniu mówimy jak o procesie, którego celem jest wytworzenie czegoś. Ten produkt końcowy to właśnie zebrany na kilku stronach nasz wysiłek w postaci poświęconego czasu i uwagi. Raport jest więc istotnym elementem całego testowania. I nie ma tu znaczenia, czy raportujemy małe zadanie z danego dnia, czy całą pracę w projekcie. Raport będzie miał w każdym przypadku taką samą strukturę.

1. Co należało zrobić?
2. Czy udało się to zrobić?
3. Jakie problemy się ujawniły?
4. Jaki jest efekt mojej pracy?
5. Co trzeba jeszcze zrobić?

I po raz kolejny zachęcam cię do rezygnacji ze standardów. Jeśli twoja organizacja nie narzuca sztywnych szablonów raportowania, nie



stosuj się od razu do IEEE 829 i nie próbuj uzupełnić wszystkich pól. Dowiedz się, jakie informacje są pożądane przez kierownika czy sponsora, i na nich się skoncentruj. Dodaj coś od siebie, chociażby subiektywną ocenę. Dobry raport nie musi mieć 50 stron. Dobry raport musi być kompletny. Musi dostarczyć wszystkie informacje wartościowe dla jego odbiorcy. Możesz w nim zawrzeć metryki typu litry wypitych kaw czy też godziny spędzone na konfiguracji środowiska, ale nie są to informacje o jakości oprogramowania. Możesz uwzględnić informacje o całościowym wysiłku, ale nie powinny być główną częścią raportu.

Moim zdaniem, raport zawsze powinien mieć formę pisemną. Jeśli coś zostaje jedynie w testerskiej głowie lub trzeba tę informację wydobyć z wielu źródeł, to nie jest raport. Informacja, którą chcesz przekazać, będzie żyła jedynie przez tę drobną chwilę, kiedy przekazujesz ją ustnie kierownikowi. Natomiast dokument jest trwałym śladem i dowodem wykonania pracy oraz jakości oprogramowania. Raport jest też podstawą do wyciągania wniosków i uczenia się na błędach. Łatwiej coś usprawniać, jeśli wnioski ma się spisane w jednym miejscu.

Elementy raportu z testów, które uważam za ważne:

1. Identyfikator dokumentu, służący również do wersjonowania. Może być zawarty w nazwie pliku.
2. Dane osoby lub osób odpowiedzialnych za testy. Ważne ze względu na możliwy kontakt przy próbie wyjaśnienia informacji z testów.
3. Elementy poddane testowaniu (w tym charakterystyki). Warto się odwołać do planu testów i wskazać, co należało zrobić, a co realnie zostało zrobione.
4. Zrealizowane zadania – po raz kolejny z odwołaniem do planu testów, by pokazać, co było planowane, a co zostało zrealizowane.
5. Uzyskane pokrycie – może być szeroko rozumiane, m.in. jako pokrycie funkcji, środowisk czy specyfikacji.
6. Metryki i miary pokazujące defekty, statusy testów itp. w sposób ilościowy.
7. Wyrocznie testowe, czyli wszystkie źródła, które pomogły określić, czy zachowanie aplikacji jest poprawne, czy nie.
8. Podsumowanie samego testowania
  1. ocena skuteczności zastosowanego podejścia do testów,

2. informacja o przeszkodach i problemach napotkanych w testowaniu,
3. skrótowy opis zmian wprowadzonych w trakcie realizowania projektu testowego i ich wpływ na zakres testów.
9. Subiektywna opinia o jakości oprogramowania wraz z rekomendacją co do wdrożenia oprogramowania.
10. Perspektywa, w której chcemy pokazać, co naszym zdaniem zostało jeszcze do zrobienia.
11. Wnioski na przyszłość (*lessons learned*).

Im mniejszego wycinka czasu lub zadań dotyczy nasz raport, tym mniej zawrzemy w nim treści. Możemy elastycznie dopasowywać raport do otoczenia i jeśli np. w projekcie brak planu testów, to nie będziemy się do niego odnosić.

Całą zawartość raportu testu można znaleźć w kompleksowym opracowaniu w normach lub na stronie [testerzy.pl](http://testerzy.pl).

### 7.5.3. Raporty o defektach

Raportowanie o defektach wchodzi w skład całego procesu testowania lub uruchamiania oprogramowania. Raport pokazuje postrzeganą przez testera różnicę między aktualnym zachowaniem aplikacji a tym, jak tester zakłada, że aplikacja powinna się zachowywać. Założenie co do zachowania aplikacji może być też oparte na mocnych podstawach, takich jak specyfikacja wymagań wobec oprogramowania czy też interpretacja przedstawiona przez istotnego użytkownika.

Kiedy podczas testowania oprogramowania znajdziesz potencjalny defekt, to wtedy zanim rozpoczniesz tworzenie raportu, czeka cię jeszcze kilka rzeczy do zrobienia. Warto rozważyć następujące kwestie:

1. Czy to, co znaleźliśmy, jest defektem oprogramowania? A może jest to zdarzenie, które nie wynika z pomyłki programistycznej? Może to nasza pomyłka albo pomyłka w teście, może to problem sprzętu? Warto sprawdzić potencjalne źródło przed napisaniem raportu, by nie zawiadamiać o zdarzeniach, które zaangażują ludzi do analizy, a której wynikiem mógłby być wniosek: „błąd testera”.
2. Czy to jest ważny defekt? Naszym celem jest raportowanie o defektach, które zostaną naprawione. Skoro więc mamy poczucie lub wiedzę, że zostaną zamknięte bez naprawy (przez swoją

trywialność lub niejednoznaczność wymagań), może nie warto tracić czasu na wypełnianie zgłoszenia? Teoretycy mówią: „Raportuj wszystko, może kiedyś ktoś to naprawi”. Testowanie jednak jest nieskończone. Po co marnować cenny czas na rzeczy błahe? W wielu narzędziach do raportowania defektów używa się statusu zgłoszenia „[naprawić] przy okazji”. Proponuję mało ważnym defektom nadać domyślny status „[zaraportować] przy okazji”, kiedy już nie będzie ważniejszych zadań. Gwarancją, że nie umknie, będzie krótka notatka o danym zachowaniu.

3. Czy ktoś już tego defektu nie znalazł wcześniej? Funkcją większości narzędzi do zarządzania defektami jest rozbudowane wyszukiwanie kontekstowe. Jego użycie pomaga wyeliminować zgłaszanie tego samego defektu przez różnych testerów. Należy unikać marnowania cudzego i własnego czasu i kilkakrotnie przeszukać bazę, zanim przypadkowo zgłosimy duplikat istniejącego już zgłoszenia. Po raz kolejny istnieje ryzyko, że ktoś uzna nasze zgłoszenie za marnowanie czasu na dodatkową, niepotrzebną analizę.
4. Czy można defekt zreprodukować? Nie na wiele zda się raport, jeśli opisanego defektu nikt nie będzie w stanie powtórzyć. Zawsze warto mieć dowód, że defekt wystąpił (log, zrzut ekranu itd.). Zawsze warto sprawdzić, czy i jak często defekt można odtworzyć.

Ale uwaga! Nie bój się raportować o defektach. Zwłaszcza, jeśli dopiero wchodzisz do projektu, zespół będzie dla ciebie bardziej wyrozumiały. Masz swój czas na wdrożenie, na naukę i raportowanie o awarii będzie formą docierania się. Szybko się dowiesz, co raportować, a co pozostawić bez zgłaszania. Dzięki raportom możesz również ujawnić dodatkowe źródła informacji o samym produkcie, wcześniej ci nieznanych.

Jeśli już zebraliśmy wszystkie niezbędne informacje, możemy przystąpić do profesjonalnego raportowania.

Oprócz tego, że defekty musimy znajdować, musimy o nich również optymalnie raportować. Chodzi o takie przekazanie informacji, by jej odbiorca zrozumiał nasz przekaz. Na zgłoszenie defektu (awarii) może się składać kilka opisanych tu elementów. W zależności od organizacji wymaganych i niewymaganych pól opisu defektów może być więcej. Te przedstawione są częścią wspólną większości raportów. Warto również

pamiętać, że nasze zgłoszenie defektów ma niewielkie szanse trafić na listę lektur obowiązkowych licealistów.

W związku z tym:

- Rozmawiaj jak inżynier z inżynierem – prosto i konkretnie.
- Stawiaj na równoważniki zdań, zamiast na kwiecistą mowę.
- Unikaj skomplikowanych konstrukcji i zdań złożonych.
- Unikaj porównań, parabol i wszystkich innych rzeczy, które pojawiały się na języku polskim przy analizie wierszy.
- Poczucie humoru jest ważne, ale dowcip w zgłoszeniu defektów już nie.
- Posługuj się technicznym slangiem tylko wtedy, gdy wiesz, że osoba po drugiej stronie zrozumie, co do niej piszesz.
- Pamiętaj o aspektach miękkich, np., że raport o defekcie nie może nieść żadnych emocji, więc wykrzykniki i całe słowa pisane dużymi literami są nieakceptowalne. Jeśli nie wiesz, w jaki sposób tekst może przekazywać emocje i jak jest on odczytywany przez innych, zachęcam do zapoznania się z netykietą.

**Tytuł.** Musi informować o tym, co się wydarzyło, w możliwie najprostszych, żołnierskich słowach. Myślimy o nim w kategoriach późniejszego wyszukiwania defektów przez innych testerów, którzy znaleźli to samo co my. Myślimy również z perspektywy programisty, który po pierwszym przeczytaniu tytułu musi mieć już pojęcie, gdzie defekt wystąpił i czego może dotyczyć. Zastanawiamy się, czy jego wagę można ocenić już po samym tytule, by uprościć pracę kierownikowi projektu, który będzie definiował priorytety w działaniu.

Zły tytuł (przykłady z życia wzięte): „WTF!”, „Defekt”, „Pojawiła się awaria”, „Wyszukiwarka nie działa” itd.

Dobry tytuł: „Przy próbie wyszukania słowa «się» lub innego z polskim znakami wyszukiwarka nie zwraca żadnego rezultatu”.

**Kroki reprodukcji.** Zwięzły opis, jak odtworzyć dany problem. Im opis jest dłuższy, tym większa irytacja czytającego, jeśli na końcu się okaże, że można to było sformułować znacznie zwięźlej. Im bardziej zdawkowy opis, tym mniejsza szansa, że czytający będzie w stanie określić, co się niepokojącego wydarzyło. Po raz kolejny tester oprogramowania musi znaleźć złoty środek.

Często stosowaną metodą jest umieszczanie wypunktowania lub też prostego numerowania kroków do wykonania. Taki zapis szybciej się analizuje.

Złe kroki reprodukcji: „Jak odpalam szukanie to mi się nie szuka”.

Dobre kroki:

1. Przeglądarka Firefox wersja XY.
2. Adres XYZ.pl.
3. „W wyszukiwarce wprowadzam słowo «się» lub inne z polskimi znakami i dokonuję wyszukania, naciskając ikonę lupy”.

**Środowisko.** Wszystkie składowe środowiska mogą wpływać na poprawne i niepoprawne zachowanie aplikacji. Jeśli nie wiemy, jakie elementy wywołują awarię, po prostu piszemy to, co wiemy o środowisku.

Źle określone środowisko: „Komputer stoi na biurku, a defekt wyświetla się na monitorze”.

Dobrze określone środowisko: „Wersja przeglądarki Firefox XY – wersja językowa polska, system operacyjny: WIN8.1 – wersja polska”.

Nieznajomość powiązań między zgłoszeniem a sprzętem i oprogramowaniem może być traktowana jako niewiedza lub niekompetencja.

**Zachowanie oczekiwane a rzeczywiste.** To, co powinno się wydarzyć, i to, co naprawdę się wydarzyło. Jest to bardzo krytyczne miejsce raportu o defekcie, ponieważ to tutaj pokazujemy swój kunszt. Przede wszystkim udowadniamy, że znamy różnicę między działającym a niedziałającym oprogramowaniem i że rozumiemy, czego potrzebuje końcowy odbiorca. Zwłaszcza jeśli specyfikacja w projekcie jest szczątkowa, niezrozumiała lub nie istnieje, tester ma okazję wykazać się umiejętnością wpływania na osoby decydujące o tym, czy defekt zostanie naprawiony, czy nie. Jeśli naszym zdaniem defekt lub awaria powinny zostać naprawione, to w tym polu szablonu mamy możliwość przekazania swoich argumentów. Jeśli dodamy do tego obowiązkową zwięzłość i zakaz okazywania emocji, to zadanie okaże się jeszcze trudniejsze.

Źle opisane zachowanie: „Miało się wyszukać, a się nie wyszukało”.

Dobrze opisane zachowanie: „Oczekiwane zachowanie: na liście wyszukiwania pojawiają się wszystkie adresy stron zawierające słowo «się». Rzeczywiste zachowanie: mimo że słowo «się» jest w wielu miejscach na stronie, żaden wynik nie zostaje zwrócony”.

Tam, gdzie oczekiwane zachowanie jest „oczywistą oczywistością”, możemy ten element pominąć. Wymaga to jednak wiedzy innych członków zespołu, a to przychodzi wraz z doświadczeniem w projektach.

**Załączniki.** To wszystkie pliki, które mogą pokazać, że defekt wystąpił. Lubię załączniki nazwać „dowodem na defekt”. Oczywiście nie zawsze dysponujemy jednoznacznym dowodem, czasami jest to jedynie poszlaka, ale jeśli zależy nam na naprawieniu danego problemu, musimy korzystać z tego, co mamy.

Złe załączniki: zrzut z całego ekranu z wszystkimi elementami dodatkowymi, niepotrzebnymi do identyfikacji defektu, takimi jak: mrugający komunikator, ikona torrenta na pasku, pootwierane zakładki Facebooka oraz pasjans w tle.

Dobre załączniki: zrzut z okna przeglądarki, gdzie pojawił się defekt, z zaznaczonym miejscem awarii (jeśli to konieczne). Plik zapisany w skompresowanym formacie, nieutrudniającym odczytania informacji.

**Klasyfikacja defektów.** W niektórych narzędziach do raportowania i śledzenia defektów pojawiają się dwie dodatkowe opcje raportowania, związane z oceną samego defektu: „ważność” (*severity*), określana czasami jako „krytyczność”, oraz „priorytet”, nazywany również „pilnością”. Są one w wielu przypadkach mylone, co utrudnia odpowiednią klasyfikację zgłoszenia.

### Odpowiedzi programistów

Czasami, mimo najszczerzych intencji dobrego zgłoszenia defektu, odbiorca po prostu go nie rozumie. W takim wypadku zwykłym ludziom ręce opadają, a tester bierze się za poprawianie i uzupełnianie raportu. Oto przykłady – w kolejności od najmniej do najbardziej popularnych – odpowiedzi programistów na zgłoszenie (za publikacją na [testavimas.blogspot.com](http://testavimas.blogspot.com)).

24. Na moim komputerze działa.

23. Jako kto się zalogowałeś?
22. To jest funkcja!
21. Działa, jak powinno (*Working as designed*, WAD).
20. Dziwne...
19. Nigdy wcześniej tak się nie działo.
18. Wczoraj działało.
17. Jak to możliwe?
16. To musi być problem sprzętowy.
15. Co źle wpisałeś, że system padł?
14. Coś jest nie tak z twoimi danymi.
13. Nie ruszałem tego kodu od tygodni!
12. Musisz mieć starą wersję.
11. To musi być nieszczęśliwy zbieg okoliczności.
10. Nie mogę przetestować wszystkiego!
9. TO nie może być przyczyna TEGO.
8. Działa, ale nie było testowane.
7. Ktoś musiał zmienić mój kod.
6. Sprawdziłeś, czy nie masz wirusa?
5. Nie działa. I jak się z tym czujesz?
4. Nie możesz używać tej wersji na swoim systemie.
3. Po co to robisz w ten sposób?
2. Gdzie byłeś, kiedy pojawił się defekt?
1. Myślałem, że to naprawiłem.

Ważność powinna określać, jak duży wpływ na system będzie miał dany defekt i z jakimi konsekwencjami się wiąże. Defekt ze względu na wagę może być (podział wg ogólnie przyjętych zasad):

1. **Krytyczny** – awaria oprogramowania związana z ryzykiem utraty danych lub uszkodzeniem zdrowia czy pozbawieniem życia użytkownika. System z takim defektem nie może zostać wydany.
2. **Bardzo poważny** – uznajemy, że ważne wymaganie końcowego odbiorcy jest niespełnione lub zaimplementowane niepoprawnie. Oprogramowanie może być używane, ale przy zachowaniu pewnych reguł.
3. **Poważny** – czyli niezgodny z wymaganiami lub tylko częściowo zaimplementowany. Oprogramowanie może być użyte z pewnymi restrykcjami.

4. **Średni** – w rozumieniu znalezienia pomniejszych odchyżeń. Oprogramowanie może być używane bez restrykcji.
5. **Trywialny** – zgłoszenie typu literówka czy niepoprawne kolory na ekranie. Oprogramowanie może być spokojnie używane.

Z kolei priorytet defektu określa pilność jego naprawy, czyli to, jak szybko powinniśmy otrzymać poprawkę. Propozycje definicji poszczególnych priorytetów:

1. **Pilny** – zablokowany jest cały proces biznesowy. Taki defekt powoduje zazwyczaj zablokowanie testów, a uruchomienie kolejnych nie może być kontynuowane.
2. **Następne wydanie** – na poprawkę możemy poczekać do następnego wydania.
3. **Przy okazji** – poprawka nastąpi, kiedy zespół będzie poprawiał inne defekty w zainfekowanym obszarze.
4. **Otwarty** – nie ma aktualnie planu poprawy (i naprawa najprawdopodobniej nigdy nie zostanie przeprowadzona).

Warto odpowiednio klasyfikować defekty w raporcie. Oszczęda to naszym kolegom programistom i przełożonym czy kierownikom pracy przy definiowaniu priorytetów i ich wagi.

Musimy się też pogodzić z faktem, że część defektów nigdy nie zostanie naprawiona. W jednej ze swoich publikacji użyłem do ich określenia terminu z nowomowy testerskiej „incydentat”. W mojej opinii są to zgłoszenia zdarzeń, które pozostają w bazach defektów bez widocznego terminu ich rozwiązania (otwarty na zawsze). Incydentat wymaga szczególnej testerskiej uwagi, aby wyłowić te zgłoszenia, które muszą zostać obsłużone, a nie zamiecione pod dywan.

**Obrona defektów.** Czasami znalezienie defektu po prostu nie wystarcza. Czasami trzeba bronić swoich racji. Brak jednoznacznego źródła informującego o tym, co jest poprawnym zachowaniem oprogramowania, a co nie, prowadzi do różnego rodzaju negocjacji i przepychanek słownych, na temat tego, co jest, a co nie jest defektem.

Jak pisałem wcześniej, defekty możemy podzielić (w najprostszym ujęciu) na znalezione i niez znalezione. Znalezienie defektu nie oznacza jego naprawienia. Są dziesiątki powodów tego, że defekt nie zostanie naprawiony (część z nich zapewne znasz), m.in.:



- to nie jest (prawdziwy) defekt;
- nie da się odtworzyć defektu;
- defekt ze zbyt dużym wpływem na inne funkcjonalności – poprawka może zdestabilizować oprogramowanie;
- defekt zbyt kosztowny do naprawienia;
- trywialny, będzie naprawiony jeśli zespół programistyczny znajdzie wolny czas;
- to nie defekt, to funkcja.

W każdym z tych przypadków tester (autor raportu) musi bronić swojego stanowiska. Może uznać, że defektu rzeczywiście nie trzeba naprawiać, przyznając się w ten sposób do błędu w raporcie. Nie zawsze trzeba defektu. Wystarczy, że z tego „niedefektu” wyciągniemy naukę na przyszłość.

Można również próbować przekonać rozmówcę do swoich racji. Upór musi jednak mieć trwalsze podstawy niż zwykle „bo tak”. Jednym z podstawowych narzędzi będzie poddanie pojedynczego defektu analizie biznesowej. Co się wydarzy, jeśli defekt znajdzie użytkownik? Jakie są potencjalne konsekwencje finansowe w porównaniu z nakładami na poprawkę? Ostatecznie zawsze należy pamiętać o perspektywie użytkownika, który prędzej powie: „Nie działa”, „Utraciłem dane” niż: „Znalazłem defekt”.

**Narzędzia do raportowania o defektach.** Narzędzia do przechowywania zgłoszeń defektów są jednymi z najbardziej popularnych w środowisku wytwarzania oprogramowania. Wszystkie wyglądają mniej więcej tak samo i poznanie jednego daje podstawy do używania innych. Ich głównym celem jest komunikowanie wszelkiego typu zgłoszeń osobom, które mają dany problem przeanalizować, zinterpretować, rozwiązać albo podjąć inną decyzję.

Narzędzi używamy po to, aby ograniczyć konieczność pamiętania o zgłoszeniach defektów. Zakładamy, że jeśli defekt jest zgłoszony, to:

- nie zostanie zapomniany lub będzie o to trudniej,
- opierając się na zgłoszonych defektach, możemy wygenerować raport ilościowy,
- możemy odwoływać się do danych i statystyk przeszłych defektów.

Narzędzie to, co bardzo potrzebne i większość projektów nie będzie

Narzędzia te są bardzo potrzebne i większość projektów nie będzie negować ich używania, ale część organizacji ponad ich użycie przekłada bezpośrednią komunikację. Taką analizę i negację dogmatu przeprowadził w swojej publikacji Gojko Adzic. Doszedł on do wniosku, że okoliczności i otoczenie testowania oprogramowania znacząco się zmieniają i są miejsca i zdarzenia, które przewartościowują raportowanie defektów do narzędzia. Skoro w metodykach zwinnych stawiamy na mocną komunikację wewnątrz zespołów, warto się zastanowić nad raportowaniem defektów bezpośrednio (słownie) programiście. Dzięki temu nie będziemy tracili czasu na wpisywanie zgłoszenia. „Pokażemy” programiście, zamiast opisywać. Defekty i tak w ostateczności jakoś muszą zostać opisane, np. jako kartka na tablicy zadań w sekcji TO DO (do zrobienia).

Oczywistym jest, że zbieranie danych i ich analiza jest ważna, ale w przytłaczającym natłoku danych projektowych kierownik ma coraz mniej czasu na ich obróbkę i analizę.

Musimy pamiętać również o wadach takiego podejścia. Jeśli nie zaraportujemy defektów, to nie wiemy, ile ich całościowo mamy? Jak zmierzmy jakość produktu? Jak ocenimy postęp? Czy jednak liczba defektów rzeczywiście mówi coś o jakości? Mówi o wysiłku „wprowadzenia” i „wyprowadzenia” defektów do lub z oprogramowania, ale niewiele o samej jakości. Łatwo wyobrazić sobie produkt, który ma defekty i jest zaakceptowany przez klienta (*de facto* tak jest w większości przypadków) i sytuację, w której aplikacja nie ujawnia żadnych poważniejszych defektów, a klient kwestionuje jej przydatność.

W niektórych firmach stosuje się „zamienniki” do narzędzi zarządzania defektami. Są wartościowe, w postaci narzędzi pozwalających zgłoszeniem zarządzać tak jak zadaniem do wykonania, i są bezwartościowe, takie jak szablony Excela umieszczone na dyskach sieciowych. Podstawowa różnica między jednymi a drugimi to kontrolowanie praw dostępu i możliwości edytowania czy kasowania zgłoszeń. W prostym arkuszu Excela praktycznie każdy może zrobić wszystko i w ten sposób zamazać lub też zdeformować obraz jakości postrzeganej jako zgłoszone defekty. W narzędziach do zarządzania zadaniami czy też defektami takie działania są mocno utrudnione.

## Przykłady defektów i awarii

Nie tak nie ciekaw jak przynajmniej defekty. Oto przykłady z życia wzięte

nic tak nie cieszy jak prawdziwe defekty. Oto przykłady z życia wzięte. Czytając, warto zwrócić uwagę na ich konsekwencje dla tych, którzy błędy popełnili.

- Klienci PKO BP stracili pieniądze ze swoich lokat. Co prawda, na krótki czas i jedynie w serwisie iPKO, ale wielu przeżyło spore zdziwienie i zdenerwowanie. Rozwiązać problemu w sposób informatyczny się nie udało. Klienci musieli udać się do banku, zerwać lokaty i ponownie je założyć. Bank przeproszał za „ewentualne niedogodności”.
- Setki składających wnioski o fundusze europejskie z programu 8.1 zdziwiła informacja, że ich wniosek nie może być zarejestrowany w systemie, jeśli przypadkowo wybrali niewłaściwą wersję formularza. Niby nie defekt aplikacji, ale... raz zablokowany wniosek nie mógł być odblokowany do edycji bez złożenia. Nie mógł być jednak złożony, gdyż był w niepoprawnej wersji. Bez możliwości edycji jego kopiowanie do nowego formularza było utrudnione. Błąd?
- „Krajowy System Informacyjny Policji jest tak zawodny, że potrafi się zawiesić na kilka dni” – donosi prasa. System ten przechowuje dane ponad 100 tys. poszukiwanych osób, 2,6 mln skradzionych lub zgubionych rzeczy, punktów karnych itp. Rozwiązanie: powstanie nowy system. Czy będzie testowany? Nie wiadomo.
- Przez pięć lat pracownik, który dostał, co prawda, pracę w Avaya Inc., ale nigdy się do niej nie zgłosił, otrzymywał pensję. Łącznie dostał 470 tys. dolarów. Defekt systemu nie przewidywał, że użytkownika można dodać i usunąć ot tak.
- Generator Wniosków o Płatność (wersja 1.1.18) miał bardzo wiele defektów. Jeden przykładowy: tej konkretnej wersji aplikacji nie można było znaleźć. Można było ściągnąć jedynie starszą wersję i zaktualizować do najnowszej.
- Wersja popularnego Firefoksa – 3.5 – już po miesiącu doczekała się krytycznej poprawki (3.5.1), eliminującej niebezpieczeństwo włamania się do komputera użytkownika przeglądarki.
- Moskiewski Sberbank przez prawie 4 godziny był w jednej trzeciej sparaliżowany. Nie funkcjonowało ok. 30% oddziałów, gdyż w czasie weekendu zaimplementowano nowe oprogramowanie zawierające poważne defekty funkcjonalne.
- System rejestracji studentów na Politechnice Śląskiej zawiera defekty, które uniemożliwiają zakończenia tego procesu.

- Wart 15 mln dolarów system wczesnego ostrzegania o awariach w elektrowni jądrowej miał awarię. Bez przyczyny po okolicy rozniósł się wielokrotnie powtarzany komunikat: „Awaria!”. Znamienne.
- Defekt w systemie ustalania zajęć w szkole Prince George’s County spowodował, że 49 tys. studentów zaczęło rok szkolny z trzydniowym opóźnieniem.
- Adobe naprawił 12 niebezpiecznych defektów w FlashPlayer. Trzy z nich zostały odziedziczone z kodu wyprodukowanego przez Microsoft.
- iPhone zawierał defekt pozwalający zdalnie przejąć kontrolę nad urządzeniem.
- Kasyno w Nowym Meksyku odmówiło wypłaty wygranej w wysokości 2,6 mln dolarów, tłumacząc się defektem oprogramowania maszyny jednorękiego bandyty.
- Luka bezpieczeństwa w bankomatach w naszej części Europy spowodowała, że pojawił się w nich malware – złośliwe oprogramowanie, zdolne do zapisywania danych z pasków magnetycznych oraz kodów PIN. Wirus umożliwiał hakerom drukowanie zapisanych danych poprzez drukarkę w bankomacie. Wywołany potrafił również otwierać kasetką z pieniędzmi.
- 22–23 maja 2009. Milion klientów kawiarni Starbucks zapłaciło podwójną cenę za kawę, jeśli używało karty kredytowej. Ciekawostka: defektu oprogramowania nie zauważyli klienci, którzy myśleli, że cena po prostu poszła w górę.
- Z powodu defektu w oprogramowaniu obsługującym w 2009 roku wybory do Parlamentu Europejskiego 7 czerwca w Polsce – w dniu wyborów – aktualizowano oprogramowanie we wszystkich obwodowych komisjach wyborczych. O mało wyniki wyborów zostałyby podane z opóźnieniem. Co ciekawe, samo oprogramowanie do samego końca zwracało komunikaty błędów związane z niepoprawnie wprowadzonymi danymi, choć tych błędów nie było. Ta sama sytuacja wydarzyła się ponownie w 2014 roku przy nieudanym wdrożeniu systemu PKW.
- Użytkownicy telefonów komórkowych z Windows Mobile mieli problem z SMS-ami. Niektóre wiadomości tekstowe zamiast z datą „2010” wyświetlają się z datą „2016”.
- Osoby, które skusiły się na zakup gry „Ogniem i mieczem”, chciały pozwać firmę CD Projekt za defekty, które uniemożliwiały im

granie.

- 29 osób zginęło podczas katastrofy helikoptera RAF. Oficjalne źródła zrzucają winę na pilota. Nieoficjalnie mówi się, że przyczyna wypadku był defekt w oprogramowaniu. BBC dotarło do dokumentu, który system kontroli sterowania silnikiem określa jako „niebezpieczny”.
- Pierwszym komunikatem przesłanym przez sieć było „lo...” Błąd? 29 października 1969 roku wiadomość ta została przesłana między dwoma komputerami połączonymi siecią ArpaNET, która stała się internetem, jaki znamy dziś. Komunikat podróżował 643 km między Uniwersytetem w Kalifornii i Instytutem w Stanford. Elektroniczna przesyłka miała być słowem „login”, ale udało się przesłać jedynie pierwsze dwie litery, zanim system padł.
- Plus GSM mógł wypaczyć wynik w programie telewizyjnym „Mam Talent”. Podczas głosowania na talenty użytkownicy po wysłaniu jednego SMS-a dostawali komunikat o błędzie w dostarczeniu wiadomości. System „odrobineń” zwariował i próbował wysłać SMS do skutku, co powodowało wysłanie kolejnych SMS-ów. Niestety system bilingował każdy wysłany SMS osobno, aż do wyzerowania konta.

**Komunikaty o błędach.** Osobną kategorią defektów są komunikaty o błędach. Źle zaprojektowany komunikat o błędzie może być gorszy niż błąd w kodowaniu. Nie tylko nie pomaga użytkownikowi w rozwiązaniu problemu, ale jeszcze potrafi doprowadzić do furii. Klient zlecający wykonanie oprogramowania rzadko sam definiuje komunikaty o błędach. To wykonawca projektu robi to za niego. Tracą obie strony. Celem testowania komunikatów o błędach jest wychwytywanie elementów:

- wprowadzających w błąd,
- nieprawdziwych,
- wywołujących dezorientację,
- trudnych w użyciu czy nawet niemożliwych do użycia.

Oto kilka przykładów. Użytkownik popełniający błąd nie chce tego zrobić. Powtórzone wykrzykniki i duże litery to w internecie symbol krzyku. Przypominam, że opisane to jest w netykierce. Pytanie: po co krzyczeć na użytkownika, kiedy popełnia błąd? Jest to objaw frustracji

tworzącego oprogramowanie. Wystarczy napisać: „Podane hasło nie ochroni bezpieczeństwa danych. Musisz podać hasło zawierające...”



**Rys. 7.33..** Wizualizacja dla niepoprawnego komunikatu

Na rys. 7.34 zaprezentowano komunikat wyświetlany na stronie z problemami technicznymi. Widzimy, że nie tylko aplikacja ma problem. Również twórca ma problem z ortografią.

//  
brak polaczenia z baza danych,  
sprobuj odswiezyc  
//

**Rys. 7.34.** Tekst niepoprawnego komunikatu błędu

Jeżeli oszukujemy klienta, że może kupić coś, czego nie może kupić, to samo formatowanie komunikatu nie zdenerwuje go już tak bardzo jak treść (rys. 7.35). Użytkownik, dokonując zakupu, nie jest informowany, że należy wybrać produkty za minimum 99 zł aż do momentu płatności. Wydaje się, że twórcy aplikacji nie podoba się to, co użytkownik zrobił, i odsyła go na powrót do sklepu.



**Rys. 7.35.** Wizualizacja niepoprawnego komunikatu błędu

Wiadomo, że użytkownik jest narażony na defekty i błędy aplikacji. A jeśli aplikacja sama z siebie, celowo wprowadza użytkownika w błąd? Takie rzeczy wychytują i starają się usunąć zarówno testerzy funkcjonalni, jak i specjaliści od użyteczności.

## 7.6. Przykładowe projekty

Wcześniej opisałem praktykę testowania, ale wymaga ona przykładów. Oto zmodyfikowane przykłady projektów testerskich wzięte z mojej kariery testerskiej.

### Projekt 1 – strona internetowa

Testowanie stron internetowych to domena mniejszych firm testerskich, ludzi z internetowego tłumu (*crowd*), czyli opisywany crowdsourcing, osoby przyjmujące zlecenia itp. Jest to doskonały poligon dla każdego początkującego testera. Oto najprostszy z możliwych projektów z realnymi założeniami.

Zlecenia testowania przychodzą zazwyczaj z firm, które same nie zatrudniają testerów oprogramowania albo nie mają wystarczającej

wiedzy czy odpowiedniego środowiska do testów. Należą do nich: pojedynczy programiści freelancerzy, mniejsze firmy wytwarzające oprogramowanie na zlecenie, agencje interaktywne itp.

## **Zlecenie testów**

Klient przysłał następujące wymagania dotyczące strony, które najprawdopodobniej stanowiły również wytyczne do jej wytworzenia.

„Strona będzie miała 6 podstron: S1, S2, S3, S4, S5, KONTAKT. Dostępny będzie przycisk: KUP BILET, który będzie linkował do zewnętrznej strony sklepu internetowego (procesu zakupu nie testujemy). Na stronie głównej będzie odtwarzacz wideo z listą filmów. Na podstronie S2 będą galerie zdjęć. W nagłówku będzie zegar odliczający czas do wydarzenia. Podstrony będą zawierały teksty, przyciski i zdjęcia. Stronicowanie będzie ustawione na sztywno i będzie wynosiło 10 elementów na stronę. Elementy strony dodawane są przez programistów (brak systemu zarządzania zawartością).

Testowanie będzie obejmowało:

- stronę na najbardziej popularnych przeglądarkach w systemach Mac i PC,
- wersję mobilną na najbardziej popularnych telefonach oraz na tabletach i iPadzie.

Dodatkowo wiadomo, że projekt jest jednorazowy, czyli że strona powstaje na potrzeby wydarzenia i nie będzie modyfikowana. Takie produkty mają bardzo krótki czas życia i relatywnie niewielki zasięg. Ich głównym celem jest przekazanie informacji. Kiedy informacja się zdewaluuje, projekt zamiera albo umiera. Skreśla to automatyzację.

O stronie wiadomo, że nie ma tam żadnych formularzy z wyjątkiem KONTAKTU.

Ponieważ samo zlecenie nie zawiera wystarczająco dużo szczegółów, należy spróbować skompensować ich brak, analizując kontekst wynikający z naszej wiedzy i doświadczenia. Można to zrobić na wiele sposobów, np. przez plan testów. Dokument ten powinien w pierwszej części pokazywać wydatkowanie budżetu i co można za zdefiniowaną kwotę lub w zdefiniowanym czasie przetestować, np. ile środowisk i jakie. Po uszczegółowieniu zakresu można w planie testów zrezygnować z niektórych wariantów.



## Plan testów

Zaproponowany plan może być nadmiarowy, ale czasami lepiej napisać trzy słowa więcej, niż później ze zleceniodawcą drzeć koty o szczegóły. Jeśli taki plan zostanie zaakceptowany, znamy zakres naszych obowiązków. Możemy się do niego odnieść na koniec projektu.

### PLAN TESTÓW

ID: TP-Strona-12

Wprowadzenie: strona internetowa, będąca wizytówką strony wydarzenia. Funkcja raczej informacyjna, więc zakładamy jednorazowe użycie oprogramowania.

Strona znajduje się pod adresem: xyz.pl

Testowane elementy:

1. strona główna,
2. podstrony ze statyczną zawartością: S1, S2, S3, S4, S5,
3. podstrona KONTAKT z formularzem kontaktowym,
4. odtwarzacz wideo z listą filmów,
5. galeria zdjęć,
6. zegar odliczający czas do wydarzenia,
7. stronicowanie (10 elementów na stronę).

Ponieważ jedna technologia obsłuży wszystkie platformy, zakłada się, że będzie to rozwiązanie RWD (strona automatycznie dopasowująca się do rozdzielczości i wielkości ekranu).

Nietestowane elementy:

1. funkcja Kup bilet,
2. system zarządzania zawartością (administracja),
3. inne – wcześniej nieokreślone,
4. charakterystyki inne niż funkcjonalność.

Podejście do testów: testy oparte na doświadczeniu.

Zadania w testach: testy weryfikacji poprawności działania w wielu środowiskach, szukanie i zgłaszanie defektów oraz przygotowanie raportu.

Zdefiniowany czas na testy: 10 godz.

Środowisko (najpopularniejsze wersje przeglądarek zgodnie z ranking.pl w dniu rozpoczęcia testów): PC Firefox, PC Chrome, PC IE, PC Opera, MAC Safari, telefon Android z natywną przeglądarką, telefon iPhone z natywną przeglądarką, telefon Windows Phone z natywną

przeglądarką, tablet Android Chrome z natywną przeglądarką, iPad z natywną przeglądarką.

Metoda KISS (*Keep it simple stupid*) działa w stu procentach. Z jednej strony, powtarzamy klientowi to, co on nam powiedział, próbując to oczywiście parafrazować, z drugiej – dajemy mu dodatkowe informacje, których brakowało w pierwszym komunikacie.

## Uruchomienie testów

Po akceptacji planu możemy spokojnie rozpoczynać testy. Oczywiście nie ma sensu tworzyć przypadków testowych. Po pierwsze, klient ich nie potrzebuje, po drugie, projekt jest jednorazowy. Wystarczy prosta umiejętność posługiwania się stronami, diagnozowania funkcji w interfejsie i podane wcześniej przykłady testów. Naszym celem będzie też sprawdzenie spójności i poprawności wyświetlania różnych elementów w różnych przeglądarkach.

## Raport z testów

Ponieważ w projekcie mamy pojedyncze uruchomienie testów bez retestów, to zgłoszenia defektów mogą być przekazane w dokumencie (w przykładowym raporcie pokazano tylko kilka). Standardowo, jeśli pojawią się choć dwie iteracje uruchomienia testów, to warto wdrożyć narzędzie do raportowania defektów. Łatwiej będzie śledzić ich statusy.

## Przykładowy raport z testów.

<b>RAPORT Z TESTÓW</b> ID: RzT-Strona-12 <b>Podsumowanie:</b> w raporcie przedstawiono wyniki prac związanych testami strony xyz.pl. Testom zostały poddane wszystkie funkcje aplikacji. Testy przeprowadzono w dniach od 22.06.20XX r. do 24.06.20XX r. <b>Status testowania funkcji</b>	
Funkcja	Status
Strona główna Podstrony ze statyczna zawartością: S1, S2, S3, S4, S5	PROBLEMY

Podstrona KONTAKT z formularzem kontaktowym	DZIAŁA
Odtwarzacz wideo z listą filmów	NIE DZIAŁA
Galeria zdjęć	PROBLEMY
Zegar odliczający czas do wydarzenia	PROBLEMY
Stronicowanie (10 elementów na stronę)	[brak]
	DZIAŁA

Przy wypełnianiu tabeli zastosowano następujące oznaczenia: DZIAŁA – oznacza, że dana funkcja działa bez zarzutu i nie zgłoszono dla niej defektów; PROBLEMY – dla danej funkcji zgłoszono drobne defekty; NIE DZIAŁA – dla danej funkcji zgłoszono poważne defekty funkcjonalności; [brak] – funkcja nie została dostarczona.

### Środowiska testów

Lista urządzeń wykorzystanych do przeprowadzenia testów, wraz z informacją o systemie operacyjnym i użytej przeglądarce.

Sprzęt, system operacyjny, przeglądarka	Status
LG G2 D802 Android 5.0.4 wbudowana	DZIAŁA
LG G2 D802 Android 5.0.4 Google Chrome	PROBLEMY
Sony Z1 Compact Android 5.0.2 wbudowana	DZIAŁA
Sony Z1 Compact Android 5.0.2 Google Chrome	PROBLEMY
Samsung Galaxy Tab2 Android 4.2.2 wbudowana	PROBLEMY
Samsung Galaxy Tab2 Android 4.2.2 Google Chrome	PROBLEMY
iPhone 4 iOS 7.1.2 Safari	PROBLEMY
iPad iOS 8.3 Safari	NIE DZIAŁA
Nokia 930 Windows Phone 8.1 IE	PROBLEMY
PC Windows 8.1 Firefox	DZIAŁA
PC Windows 7 Opera	PROBLEMY
PC Windows 7 Chrome	DZIAŁA
PC Windows 7 Firefox	DZIAŁA
PC Windows 7 IE	DZIAŁA
PC Windows Vista Business Firefox	DZIAŁA
PC Windows Vista Business IE	DZIAŁA
PC Windows Vista Business Chrome	DZIAŁA

Przy wypełnianiu tabeli zastosowano następujące oznaczenia: DZIAŁA – oznacza, że nie zgłoszono defektów funkcjonalności danego środowiska wynikających z jego użycia; PROBLEMY – znaleziono pomniejsze defekty wynikające z posługiwania się danym środowiskiem; NIE DZIAŁA – zgłoszono poważne defekty funkcjonalności danego środowiska wynikające z jego użycia.

### Defekty

## Zgłoszono następujące defekty w aplikacji

ID	Tytuł	System	Przeglądarka	Kroki reprodukcyjne/obszar	Opis
1	Brak możliwości zmiany rozdzielczości w filmach	W7	Google Chrome Android Chrome Firefox	Odtwarzanie wideo	Jest dostępna funkcja zmiany rozdzielczości w filmach, ale nie działa
2	Nieprawidłowo działający przycisk pauzy podczas wyświetlania filmów	Android 5.0.2	Android Chrome	1. Uruchamiamy film 2. Klikamy przycisk Pauza 3. Ponownie klikamy przycisk Pauza	Przycisk nie działa po jednym kliknięciu. Aby wyświetlić dalszą część filmu, należy kliknąć obszar wyświetlania
3	Animacja ładowania nie znika po załadowaniu filmiku	Android 5.0.2 iOS 7	Android Chrome Safari	1. Włącz film na stronie głównej 2. Włącz powiązany film	Wyświetlana jest animacja ładowania zarówno aplikacji odtwarzacza, jak i aplikacji YouTube. Po załadowaniu odtwarzacza YouTube film zaczyna się odtwarzać przy ciągłym ładowaniu animacji aplikacji
4	Brak odnośnika do napisu: „Kliknij, aby zakupić”	W7	Google Chrome Firefox IE	Strona główna	Tekst nie jest podlinkowany do niczego
5	Nieprawidłowe działanie przycisku [x]	Android 5.0.2	Android Chrome		Przycisk ten występuje w każdej zakładce. Kliknięcie go nie powoduje wykonania akcji
6	Nieprawidłowe działanie zakładek na podstronie bilety	Android 5.0.2	Android Chrome (wertykalna)	S2	Po kliknięciu zakładki S2' a następnie Wstęp i ponownie Wstęp nie chowa się ona

7	Nieemożność wysłania prawidłowo skonstruowanej wiadomości w zakładce Kontakt	WPhone 8.1	IE	Kontakt	Z poprawnie wypełnionego formularza nie można wysłać wiadomości. Mimo kliknięcia w Wyślij wiadomość nie zostaje wysłana
8	Komunikat: „Unidentified browser” po przejściu na stronę ebilet.pl	WPhone 8.1	IE	Zakup biletu	Przeglądarka IE nie jest obsługiwana przez stronę, na której można kupić bilety
9	Nieprawidłowe wyświetlanie przycisków przy pełnym ekranie odtwarzania filmów	WPhone 8.1	IE	Odtwarzanie filmów	Przyciski są za małe i korzystanie z nich jest niemożliwe
...	...	...	...	...	..

## Podsumowanie

W przeciągu przepracowanego czasu zostały przetestowane wszystkie z wyznaczonych obszarów. Podczas testów testowane obszary zostały pokryte w różnym stopniu, w zależności od ich złożoności i skomplikowania.

Wśród 69 zaraportowanych zgłoszeń są błędy funkcjonalności i użyteczności o różnych priorytetach.

Nie stwierdzono defektów krytycznie wpływających na działanie aplikacji, ale nie wszystkie zadeklarowane środowiska są obsługiwane.

Rekomendujemy poprawki zaraportowane dla krytycznych dla klienta środowisk.

## Projekt 2 – strona internetowa z projektem

Na testerskim warsztacie może pojawić się zadanie testowania oparte na zdefiniowanym wyglądzie strony wraz z testowaniem funkcji oprogramowania.

## Zlecenie testów

Klient zlecający firmie wytwórczej zaprojektował wraz z analitykiem i grafikami poszczególne podstrony i chciałby mieć pewność, że otrzymał to, co zamawiał i za co zapłacił. Dotyczy to zarówno wyglądu aplikacji, jak i poszczególnych funkcji. Ponieważ chce mieć pewność profesjonalnego wykonania zadania, zleca to na zewnątrz. Z racji tego, że może się pojawić konieczność efektywnego komunikowania i dyskusowania defektów na linii klient–wytwórca–testerzy, udostępnia się narzędzie do raportowania o defektach.

Przekazane zostają projekty ekranów aplikacji, na podstawie których programiści mieli zbudować aplikację.

Brak specyfikacji funkcjonalnej wymusza przeprowadzenie testów eksploracyjnych.

## Plan testów

Plan testów powstaje jedynie na potrzeby testerów jako definicja celów do ociążnięcia.

<p>PLAN TESTÓW</p> <p>ID: TP-KLIENT-XY</p> <p>Wprowadzenie: strona internetowa, będąca wyszukiwarką produktów, bez możliwości ich zakupu.</p> <p>Strona znajduje się pod adresem: zyx.pl.</p> <p>Testowane elementy:</p> <ul style="list-style-type: none"><li>• wygląd oparty na layoutach,</li><li>• podstrony ze statyczną zawartością,</li><li>• galeria zdjęć,</li><li>• wyszukiwanie i stronicowanie.</li></ul> <p>Nietestowane elementy:</p> <ul style="list-style-type: none"><li>• testom nie podlega logowanie się do aplikacji za pomocą konta superużytkownika.</li></ul> <p>Podejście do testów: testy eksploracyjne i testy oparte na specyfikacji.</p> <p>Zdefiniowany czas na testy: 24 godz., podzielone na sesje testowe.</p> <p>Środowisko (najpopularniejsze wersje przeglądarek zgodnie z ranking.pl w dniu rozpoczęcia testów): PC Firefox, PC Chrome, PC IE, PC Opera.</p>
---

## Uruchomienie testów

Uruchomienie testów opierało się na sesjach testowych. Pierwsza, godzinna analiza pozwoliła na wskazanie 16 sesji trwających średnio 1 godz. każda.

### Przykładowe cele sesji testowych

Lp.	Cel sesji	Czas	Komentarz
1	Zweryfikuj layouty względem wyglądu strony	3 godz.	Podzielono na trzy sesje po 1 godz.
2	Przetestuj wyszukiwanie produktów wraz ze stronicowaniem i z sortowaniem	1,5 godz.	Sesja może zostać wydłużona, jeśli pojawi się wiele defektów
3	Sprawdź statyczne podstrony	0,5 godz.	Proste przejście nawigacji i pobieżne sprawdzenie treści
4	Przetestuj galerię zdjęć	1 godz.	Pełne zweryfikowanie uruchamiania, zamykania i nawigowania po zdjęciach

Dodatkowy czas zarezerwowano na zgłaszanie defektów i przygotowanie raportu końcowego.

### Raport z testów

Szablon raportu został zaproponowany przez firmę testującą, ale uzupełniony o wymagania klienta. Umieszczono w nim dodatkowo defekty (oprócz raportowania o nich w narzędziu), żeby były zebrane w tym samym dokumencie. Nie jest to standardowo wykonywane, gdyż wymaga powielania pracy. W raporcie nie wymagano określenia, co działa poprawnie, a jedynie zgłaszania defektów. W raporcie pokazano kilka przykładowych defektów. Usunięto dane mogące zidentyfikować produkt.

#### RAPORT Z TESTÓW

ID: RzT-Strona-12

#### Informacje wstępne

#### Wprowadzenie

Sposób testowania:

1. Testy eksploracyjne (funkcjonalne) sprawdzające zgodność aplikacji w różnych przeglądarkach internetowych.

2. Testy odwołujące się do dostarczonych layoutów frontendu aplikacji.

### **Przedmiot testów**

XYZ wersja z dnia DD.MM.RRRR.

### **Cechy obiektów nie podlegające testowaniu**

Nie testowano następujących layoutów ze względu na zmianę założeń projektowych: 1.jpg, 2.jpg, 7.jpg.

Nie testowano elementów wymagających logowania się do aplikacji za pomocą konta superużytkownika.

### **Cechy obiektów podlegające testowaniu**

Testowaniu podlegają dostępne dla użytkownika elementy aplikacji od strony funkcjonalnej, użytkowej i estetycznej.

### **Podmiot testujący**

Firma testerska.

### **Środowisko testowe**

Środowisko testowe: Google Chrome X, Mozilla Firefox Y, Opera Z, IE W.

Nazwa systemu operacyjnego: Microsoft Windows 7 Home Premium 64bit Service Pack 1.

### **Czynności, techniki i narzędzia**

Testy eksploracyjne (testy funkcjonalności).

### **Kryteria zaliczenia/niezaliczenia testu**

Dla testów funkcjonalnych przypadek jest niezaliczony, gdy wystąpił przynajmniej jeden z wymienionych efektów:

- niezgodność funkcjonalności aplikacji ze spodziewanym efektem,
- zawieszenie lub upadek aplikacji,
- niepoprawny komunikat o błędzie,
- nieoczekiwane działanie aplikacji,
- komunikat wprowadzający użytkownika w błąd,
- komunikat nieadekwatny do zdarzenia.

### **Zadania do wykonania**

1. Zapoznanie się z dostarczonymi informacjami na temat testowanej aplikacji.
2. Określenie testowalności aplikacji.
3. Przeprowadzenie testów eksploracyjnych (funkcjonalnych).
4. Weryfikacja layoutów.
5. Przygotowanie raportu.

### **Testy funkcjonalne i ich wyniki**



Testy polegały na jednoczesnym poznawaniu aplikacji, wykonywaniu testów eksploracyjnych i tworzeniu dokumentacji testowej. Tester przystępował do pracy ze ściśle określonymi celami, takimi jak sprawdzenie funkcjonalności aplikacji oparte na najlepszych praktykach.

### **Defekty zgłoszone w systemie raportowania.**

Decyzję o poprawieniu znalezionych defektów powinna zapaść po stronie zamawiającego testy aplikacji.

#### **Wykryte defekty**

#1 Internal Server Error 500 przy próbie wyświetlenia zakładki Z1.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: funkcjonalność.

Priorytet: krytyczny.

Po kliknięciu zakładki Z1 z górnego poziomego menu otrzymujemy komunikat o wyjątku 500 Internal Server Error. Log z wyświetlonymi błędami znajduje się w załączniku do raportu o nazwie 1-1.txt.

Ekran defektu: [załącznik, grafika: Z1.png]

<....>

#3 Pusta zakładka OLEJE SILNIKOWE w menu BAZA WIEDZY.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: użyteczność.

Priorytet: niski.

Po kliknięciu zakładki OLEJE SILNIKOWE nie wyświetla się żadna treść. Brak wiedzy, czy to efekt zamierzony i baza wiedzy jest obecnie pusta, czy na stronie brakuje elementów, które powinny być wyświetlane.

<...>

#5 Brak stosownego komunikatu w przypadku braku wyników wyszukiwania produktów.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: estetyczny.

Priorytet: niski.

W zakładce PRODUKTY w zależności od wybranego rodzaju produktów użytkownik może sortować produkty wg ich parametrów.

Jeśli po wybraniu przez użytkownika parametrów nie ma żadnych wyników sortowania, otrzymujemy pustą stronę bez żadnego komunikatu. Zgodnie z regułami wspierania użytkownika należy poinformować o braku produktów dla wybranych przez niego kryteriów.

#6 Brak ograniczenia liczby znaków w wyszukiwarce, w zakładce WYBIERZ PRODUKT.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: użyteczność.

Priorytet: niski.

W polu tekstowym Wyszukiwanie można wpisać nieograniczoną liczbę znaków, co powoduje efekt „rozjechania” widoczny na screenshocie.

Ekran z wynikami wyszukiwania po słowach kluczowych: [załącznik, grafika: wyszukiwanie.png]

#7 Brak obsługi języków innych niż polski.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: funkcjonalny.

Priorytet: wysoki.

Po zmianie języka na angielski lub niemiecki brakuje opisów produktów, artykułów w bazie wiedzy itd.

#8 Komunikat pojawiający się podczas wyszukiwania na stronie.

Przeglądarka: Chrome, Opera, IE.

Typ błędu: estetyczny/użyteczność.

Priorytet: niski.

Komunikat informujący o minimalnej liczbie znaków wpisanych w wyszukiwarce wygląda jak zwykły alert w JavaScriptcie (w Firefoksie komunikat ten wygląda poprawnie).

<...>

#11 Przycisk Zaloguj się w dolnym menu.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: funkcjonalność.

Priorytet: niski.

Przycisk Zaloguj się odsyła użytkownika na stronę <http://www.tutajdajlogowanie.pl/>.

#12 Pobieranie pliku PDF w zakładce Z2.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: użyteczność/funkcjonalność.

Priorytet: niski.

Nie da się pobrać pliku PDF WNIOSEK na dole strony. Plik nie jest podlinkowany.

<...>

#14 Hasła, które wydają się ważne dla serwisu, nie zwracają rezultatów.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: funkcjonalność.

Priorytet: wysoki.

Wyszukiwarka nie wyświetla żadnych wyników wyszukiwania potencjalnie ważnych haseł, takich jak: „książka kucharska”, „produkt 1”, „kubek z nadrukiem”, „e-book”, mimo że hasła te występują w treści podstron.

#15 Wielokrotne wyświetlanie tych samych podstron w wynikach wyszukiwania.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: użyteczność/funkcjonalność.

Priorytet: średni.

Wyszukiwanie haseł jest losowe. Po wpisaniu hasła „firma1” ten sam wynik wyszukiwania występuje kilka razy w wynikach.

#16 Wyszukiwarka zwraca błąd przy wyszukiwaniu frazy „\*”.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: funkcjonalność.

Priorytet: krytyczny.

Przy próbie wyszukania fraz takich jak „\*” , „+” , „{” , „ ?{ „ pojawia się wyjątek widoczny w załączniku.

Ekran defektu: [załącznik, grafika: wyjatek.png]

#18 Wyszukiwarka przy wyszukaniu niektórych długich ciągów znaków zgodnie z informacją osiąga szybkość poniżej zera sekund.

Przeglądarka: wszystkie przeglądarki.

Typ błędu: funkcjonalność.

Priorytet: niski.

Przy wyszukiwaniu bardzo długich ciągów znaków wyświetlany czas wyszukiwania jest ujemny, np. dla „Książka, książka, książka, książka, książka, książka, książka, książka, książka, książka, książka, książka” jest to informacja „Wyniki: 0 (w -0,5 s)”

<...>

#22 Brak opcji sortowania w bazie wiedzy w Firefoksie

Przeglądarka: Firefox

Typ błędu: funkcjonalność.

Priorytet: krytyczny.

Opcja sortowania artykułów na stronie S3 nie jest wyświetlana.

#23 Niezgodność wyników wyszukiwania produktu z wybranymi kryteriami.

Przeglądarka: wszystkie przeglądarki

Typ błędu: funkcjonalność.

Priorytet: wysoki.

Po wyborze w zaawansowanym wyszukiwaniu filtra „meble biurowe” w wynikach otrzymujemy produkty ze wszystkich kategorii.

<...>

### **Podsumowanie**

Aplikacja jest przejrzysta wizualnie, niestety czasami dość ciężka ze względu na liczbę i wielkość wyświetlanych obrazków.

## **Projekt 3 – aplikacja internetowa z procesem wspierającym wytwarzanie i utrzymanie oprogramowania**

To już prawdziwy projekt narzędzia, które nie tylko mogłoby być testowane, ale mogłoby również być używane przez testerów. Jest to system zgłoszeń defektów przez użytkowników.

### **Zlecenie testów**

Klient zlecający przeprowadzenie testów udostępnił instrukcję użytkownika oraz instrukcję instalacji produktu. Na tej podstawie została przygotowana oferta przeprowadzenia testów funkcjonalnych z ewentualnym sprawdzeniem wydajności statycznej i użyteczności.

### **Plan testów**

Plan testów był wymagany, ale jego konstrukcja była mocno powiązana z projektem i aby nie ujawnić danych klienta, nie będzie tu pokazany.

### **Uruchomienie testów**

Uruchomienie testów opierało się na weryfikacji przekazanej specyfikacji z oprogramowaniem oraz wysokopoziomowych przypadkach testowych. Podczas testów przekazywano również uwagi dodatkowe.

### **Raport z testów**

Szablon raportu po raz kolejny został uzgodniony z klientem. Umieszczono w nim spis defektów. W zaprezentowanym tu raporcie

pokazano kilka przykładowych defektów. Usunięto dane mogące zidentyfikować produkt. Są to jedynie fragmenty całego raportu.

## **RAPORT Z TESTÓW**

ID: NrD-Aplikacja-1

### **Sposób testowania**

1. Testy funkcjonalne sprawdzające zgodność aplikacji z dostarczoną dokumentacją.
2. Testy eksploracyjne obejmujące przede wszystkim funkcjonalność oraz użyteczność.

### **Poziom testowania**

1. Poziom testów eksploracyjnych obejmuje wszystkie miejsca, do których użytkownik ma dostęp.
2. Poziom testów funkcjonalnych obejmuje opisane w dokumentacji działania, których wykonanie jest możliwe z nadanymi uprawnieniami.
3. Testy użyteczności oparte na wiedzy eksperckiej oraz standardach jakości przekazywania informacji w internecie.

### **Cechy obiektów niepodlegające testowaniu**

1. Komunikaty wysyłane z systemu.
2. Dynamiczna wydajność systemu – system testowany lokalnie.

### **Cechy obiektów podlegające testowaniu**

Testowaniu podlegają wszystkie funkcje realizowane za pomocą serwisu.

### **Środowisko testowe**

Środowisko testowe: Mozilla FF, Microsoft Internet Explorer.

Nazwa systemu operacyjnego: Microsoft Windows 7 Home Premium Service Pack 1.

### **Kryteria zaliczenia/niezaliczenia testu**

Dla testów funkcjonalności przypadek jest niezaliczony, gdy wystąpił przynajmniej jeden z wymienionych efektów:

1. zawieszenie lub upadek aplikacji,
2. brak funkcjonalności,
3. nieoczekiwane działanie aplikacji.

Dla testów funkcjonalności przypadek jest zaliczony, mimo że wystąpił przynajmniej jeden z wymienionych efektów:

1. błąd użyteczności,
2. niepoprawny komunikat o błędzie,
3. komunikat wprowadzający użytkownika w błąd,

4. komunikat nieadekwatny do zdarzenia.

### **Zadania do wykonania**

1. Instalacja aplikacji.
2. Zapoznanie się z dostarczoną dokumentacją.
3. Określenie testowalności aplikacji i wytypowanie wrażliwych obszarów.
4. Zdefiniowanie wysokopoziomowych przypadków testowych.
5. Przeprowadzenie testów eksploracyjnych.
6. Przygotowanie raportu.

### **Testy funkcjonalne i ich wyniki**

Testy sprawdzające poprawność działania aplikacji w odniesieniu do „Instrukcja użytkownika. XXX. Wersja 1.00 z dnia DD.MM.RRRR”.

Wyniki testów:

ID – identyfikator, gdzie X.Y numeracja rozdziału pokrewna z zawartością instrukcji, a PX – numeracja wysokopoziomowego przypadku np. P1.

Działanie – opis wykonanego przypadku.

Dział – TAK/NIE znaleziono błędy funkcjonalności.

ID	Działanie	Dział?
1.1P1	Logowanie ważnym loginem i hasłem	TAK
1.1P2	Wylogowanie	TAK
1.2P1	Rozmieszczenie elementów aplikacji wg rys. 1.2	TAK
1.2P2	Zmiana hasła	TAK
1.2P3	Zmiana wersji językowych	NIE
1.2P4	Nawigacja po menu głównym	TAK
1.3P1	Sprawdzenie, czy „Dostęp do wybranych funkcji systemu mają tylko użytkownicy z odpowiednimi uprawnieniami”	TAK
1.3P2	Wywołanie komunikatu: „Komunikat o braku uprawnień użytkownika do funkcji systemu”	TAK
1.4P1	Sprawdzenie poprawności działania opisanego ikonami	TAK
1.4P2	Sprawdzenie oznaczeń dla pól wymaganych i niewymaganych	TAK
1.5P1	Sprawdzenie prezentacji listy	TAK
1.5P2	Otwarcie szczegółu obiektu na liście	TAK
1.5P3	Użycie pola filtru (poprawność filtrowania)	TAK
1.5P4	Użycie sposobu filtrowania	TAK

1.5P5	Użycie nawigacji	TAK
1.5P6	Testy zapisu ustawień	TAK
1.5P7	Testy okna ustawień	TAK
2.1P1	Sprawdzenie czy profil użytkownika wygląda jak na rys. 2.1	TAK
2.2P1	Sprawdzenie możliwości edycji profilu	TAK
2.2P2	Sprawdzenie możliwości zmiany ustawień profilu	TAK
2.3P1	Sprawdzenie zakładek na dole: Konto użytkownika i Uprawnienia do projektu	TAK
3.1P1	W oknie Lista prezentowane są wszystkie dostępne aplikacje	TAK
3.2P1	Użycie przycisku Przejście do szczegółów	TAK
3.2P2	Sprawdzenie możliwości edycji aplikacji	TAK
3.2P2	Użycie Powrotu do listy	TAK
3.2P3	Sprawdzenie zakładek na dole: Podsumowanie, Historia zgłoszeń, Wersje aplikacji	TAK
3.3P1	Dodanie nowej aplikacji z pozycji menu Aplikacje → Nowa aplikacja	TAK
3.3P2	Dodanie nowej aplikacji z pozycji paska skrótów	TAK
3.3P3	Poprawne zapisanie nowej aplikacji	TAK
4.1P1	Sprawdzenie poprawności wyświetlania się Listy	TAK
4.2P1	Poprawne wyświetlanie szczegółów systemu	TAK
4.2P2	Sprawdzenie możliwości edycji systemu	TAK
4.2P3	Działanie Powrotu do listy	TAK
4.2P4	Sprawdzenie działania zakładek Podsumowanie i Historia zgłoszeń	TAK
4.3P1	Dodanie nowego systemu z pozycji menu Systemy → Nowy system oraz z pozycji paska skrótów	TAK
4.3P2	Zapis nowo dodanego systemu	TAK
5.1P1	Poprawne wyświetlanie listy kontrahentów	TAK
5.2P1	Przejście do szczegółów kontrahenta	TAK
5.2P2	Sprawdzenie możliwości edycji i powrotu do listy	TAK
5.2P3	Sprawdzenie działania zakładek: Podsumowanie, Umowy, Historia zgłoszeń, Osoby kontaktowe	TAK
5.3P1	Dodanie nowego kontrahenta z pozycji menu Kontrahenci → Nowy kontrahent oraz z paska skrótów	TAK
5.3P2	Sprawdzenie poprawności zapisu nowo dodanego kontrahenta	TAK

6.1P1	Sprawdzenie poprawności wyświetlania się listy umów	TAK
6.2P1	Przejsie do szczegółów Umowy	TAK
6.2P2	Sprawdzenie poprawności działania przycisków: Edytuj, Zakończ i Powrót do listy	TAK
6.2P3	Sprawdzenie zakładek: Podsumowanie, Historia zgłoszeń	TAK
6.3P1	Sprawdzenie poprawności działania dodania nowej umowy z pozycji menu Umowy → Nowa umowa i z paska skrótów	TAK
6.3P2	Sprawdzenie poprawności zapisu nowej Umowy	TAK
7.1P1	Sprawdzenie poprawności wyświetlania się Listy zgłoszeń	TAK
7.2P1	Przejsie do szczegółów zgłoszenia	TAK
7.2P2	Sprawdzenie działania funkcji: Akceptacja	TAK
7.2P3	Sprawdzenie działania funkcji: Odrzucenie	TAK
7.2P4	Sprawdzenie działania funkcji: Zamknij zgłoszenie	TAK
7.2P5	Sprawdzenie działania funkcji: Rozpoczęcie realizacji	TAK
7.2P6	Sprawdzenie działania funkcji: Anulowanie zgłoszenia	TAK
7.2P7	Sprawdzenie działania funkcji: Komentarz	TAK
7.2P8	Sprawdzenie działania funkcji: Przywrócenie zgłoszenia	TAK
7.3P1	Dodanie nowego zgłoszenia z poziomu menu i paska skrótów	TAK
8.1P1	Przejsie do listy personelu	TAK
8.1P2	Przejsie do szczegółów personelu	TAK
8.1P3	Sprawdzenie działania funkcji: Edytuj, Usuń, Przywróć i Powrót do listy	TAK
8.1P4	Sprawdzenie działania zakładek: Konto użytkownika, Uprawnienia i Uprawnienia do projektu	TAK
8.2P1	Przejsie do słownika stanowisk personelu	TAK
8.2P2	Przejsie do szczegółów stanowiska	TAK
8.2P3	Sprawdzenie działania funkcji Edytuj i Powrót do listy	TAK
8.3P1	Dodanie nowego stanowiska	TAK
9.1P1	Przejsie do menu Raporty, Zgłoszenia serwisowe	TAK
9.1P2	Przejsie do menu Raporty, Zgłoszenia serwisowe z czasem obsługi	TAK
9.1P3	Przejsie do menu Raporty, Sumaryczny czas obsługi zgłoszeń	TAK

Spośród wszystkich funkcjonalności nie działa jedynie zmiana wersji językowej. Pozostałe działania systemu są pokrewne z instrukcją.



## Znalezione defekty

ID	Waga	Prio	OS	Status	Podsumowanie
5826	Powa	P3	Wind	NEW	Dodanie zgłoszenia dla nowego kontrahenta
5782	Norm	P3	Wind	NEW	Użytkownik o zerowym poziomie dostępu może przejść do dodawania nowego zgłoszenia
5788	Norm	P3	Wind	NEW	Przy wyłączonej obsłudze JavaScriptu <i>błędy w formularzu przy polu Województwo</i>
5771	Norm	P3	Wind	NEW	Wprowadzenie do pola Login i hasło długiego ciągu znaków daje nieprawidłowy komunikat o błędzie
5774	Norm	P3	Wind	NEW	W szczegółach zgłoszenia po zmianie jego statusu dostajemy komunikaty o błędach
5777	Norm	P3	Wind	NEW	Możliwość przypisania zgłoszenia do wygasłej umowy
5783	Norm	P3	Wind	NEW	Niepoprawne wyświetlanie numeru telefonu komórkowego w szczegółach kontrahenta
5786	Norm	P3	Wind	NEW	Wprowadzenie modyfikacji w formularzu zgłoszenia i kliknięcie Powrót nie wywołuje komunikatu „Czy na pewno chcesz opuścić edycję bez zapisania zmian?”
5789	Norm	P3	Wind	NEW	Włączenie braku zgody na zapisywanie plików cookies uniemożliwia zalogowanie się do systemu
5827	Norm	P3	Wind	NEW	Nieprawidłowe wyświetlanie się podglądu wydruku
5778	Norm	P3	Wind	NEW	Możliwość dania dostępu do zarządzania aplikacją personelowi, bez możliwości jej przeglądania
5781	Norm	P3	Wind	NEW	Użytkownik o zerowym poziomie dostępu może przejść do zakładki Zgłoszenia i raporty
5776	Norm	P3	Wind	NEW	Dodawanie osoby kontaktowej do zgłoszenia zwraca pole z kodem
5779	Drob	P3	Wind	NEW	Ustawienia profilu, pola wyboru z prawej strony
5785	Drob	P3	Wind	NEW	Angielski nagłówek w edycji osoby kontaktowej
5780	Drob	P3	Wind	NEW	Niepoprawna walidacja pola adresu e-mail
5769	Drob	P3	Wind	NEW	Przycięcie nazwy stanowiska na liście Osoba odpowiedzialna za aplikację
5772	Drob	P3	Wind	NEW	Walidacja pola Kod pocztowy przy dodawaniu nowego kontrahenta
5775	Drob	P3	Wind	NEW	Wpisanie do pola Telefon niepoprawnych danych zwraca dziwny komunikat o błędzie

5784	Drob	P3	Wind	NEW	Po dodaniu niepoprawnego adresu e-mail pojawia się czerwona ikona zamiast komunikatu
5787	Drob	P3	Wind	NEW	Wpisanie długiego ciągu znaków do pola System przy dodawaniu systemu uniemożliwia dodanie go
5790	Drob	P3	Wind	NEW	Angielskie nazwy po filtrowaniu wg daty na liście zgłoszeń
5770	Drob	P3	Wind	NEW	Walidacja pola NIP przy dodawaniu nowego kontrahenta
5773	Drob	P3	Wind	NEW	Wyświetlanie daty w formularzu dodawania kontraktu po błędnym wpisaniu

### **Testy wydajności statycznej**

Do testów użyto benchmarków wydajności dla stron internetowych. Aplikacja nie ma żadnych poważnych problemów z wydajnością w swojej konstrukcji. Znaleziono niewielkie błędy, które mogą przekładać się na wolniejszy czas odpowiedzi użytkownika w trakcie użytkowania aplikacji:

1. Strona jest relatywnie ciężka – ma prawie 600 kB bez cache'owania zawartości, co może powodować jej wolne ładowanie przy pierwszych odwiedzinach.
2. Ciężar strony wynika przede wszystkim z dużej ilości z kodu JavaScriptu oraz CSS-ów. Przy większej liczbie zapytań do serwera (mało prawdopodobne) może to powodować poważne obciążenia po stronie serwera.

### **Testy użyteczności (dodatkowe)**

Badaniu poddano dodatkowo wszystkie elementy strony, łącznie ze szczególnym naciskiem na:

1. proces zgłaszania,
2. proces definiowania opcji,
3. proces administracji.

Defekty użyteczności nie są zgłaszane do systemu zarządzania defektami, ponieważ użyteczność nie jest zdefiniowana wymaganiami. Są to opinie i analiza osoby doświadczonej w badaniu interfejsów.

Poniżej znajduje się lista uwag, które mogą być uznane za błędy trywialne lub też błędy użyteczności.

1. Aplikacja na ekranie głównym ma zdublowane przyciski Wyloguj.
2. Przycisk okna ustawień – pokazuje/ukrywa okno służące do usuwania/dodawania kolumn do listy. Kolumny usuwamy/dodajemy do listy przez przeciąganie ich z/do okna ustawień – funkcja ta jest nieintuicyjna i trudna w użyciu.

3. Niepotrzebnie klikalne nagłówki po Przejściu do szczegółów obiektu dowolnej opcji.
4. Złe rozmieszczenie przycisków w Szczegółach kontrahenta. Rys. 5.3 z instrukcji obsługi.
5. Rys. 6.4 – niezrozumiałe pole obowiązkowe Umowa.
6. <...>
7. Ustawienia profilu to w rzeczywistości ustawienia powiadomień.
8. Listwa obszarów, np. Akceptacja zgłoszenia, na ekranie jest aktywna i sprawia wrażenie możliwej do kliknięcia przez pojawienia się rączki po najechaniu.
9. Użytkownik o zerowym poziomie dostępu po kliknięciu w zakładkę np. KONTRAHENCI znajduje się na stronie BRAK DOSTĘPU bez informacji, do czego nie ma dostępu.
10. W szczegółach KONTRAHENTA przycisk DODAJ SYSTEM dodaje elementy do górnej tabelki w obszarze system, a przycisk DODAJ APLIKACJĘ do dolnej.
11. W szczegóły osoby kontaktowej przyciski EDYTUJ NOWY USUŃ nie wyglądają jak klawisze do wykonania akcji, a ich lokalizacja jest nieintuicyjna.
12. Można przypisać odpowiedzialność za umowy do osób, które nie mają uprawnień, aby nimi zarządzać.
13. Trudno określić, po ile stronicowane są elementy. Na liście zgłoszeń wyświetlanych jest 20 elementów, a w sekcji zgłoszeń serwisowych jest to 10 elementów.
14. Można dodawać systemy i kody systemów o takiej samej nazwie.

### **Podsumowanie**

Aplikacja działa poprawnie w odniesieniu do instrukcji, ale instrukcja użytkownika nie jest dokumentem, który będzie często czytany przez użytkowników. Znalezione błędy nie mają wagi krytycznej. Aplikacja, choć przejrzysta wizualnie, nie jest jasna, jeśli chodzi o możliwe w niej działania. Szczególnie poważny jest brak informacji o właściwych przyczynach i konsekwencjach problemu. Przykładowo brak poprawnej instalacji X powoduje pojawienie się informacji o niemożliwości przetworzenia zgłoszenia, podczas gdy w rzeczywistości jest ono dodane do systemu. Brakuje również obsługi zdarzeń po stronie przeglądarki. Przy braku zgody na zapis pliku cookie lub też przy braku obsługi JavaScriptu aplikacja nie wykrywa i nie informuje, że użytkownik nie będzie w stanie zakończyć pewnych procesów.

Sam proces zgłaszania jest relatywnie intuicyjny, ale administracja nie będzie możliwa bez dodatkowych szkoleń dla administratorów.

## **Projekt 4 – testy edukacyjnej aplikacji desktopowej**

Testom podlegają nie tylko aplikacje internetowe, ale również desktopowe. Dla dużej firmy zajmującej się tworzeniem oprogramowania edukacyjnego przeprowadziliśmy testy funkcjonalne i wydajności, ze wskazaniem optymalnej konfiguracji sprzętowej dla danego oprogramowania.

### **Zlecenie testów**

Firma zlecająca oprócz oprogramowania udostępniła również specyfikację wymagań wobec aplikacji.

### **Plan testów**

Plan testów nie był wymagany, a jego skrócowa wersja nawiązywała do wcześniej pokazanych.

### **Uruchomienie testów**

Uruchomienie testów opierało się na wielu konfiguracjach sprzętowych, aby wskazać te, na których aplikacja działa w sposób optymalnie szybki albo akceptowalnie szybki. Weryfikacja specyfikacji oprogramowania bazowała na eksploracji przy użyciu listy kontrolnej opisującej zdefiniowane funkcje.

### **Raport z testów**

Szablon raportu został uzgodniony z klientem. Umieszczono w nim spis defektów. W zapezentowanym tu raporcie pokazano kilka przykładowych. Usunięto dane mogące zidentyfikować produkt. Są to jedynie fragmenty całego raportu.

RAPORT Z TESTÓW

ID: AE-desktop-3

## **Wprowadzenie**

Testy miały na celu sprawdzenie, czy aplikacja poprawnie uruchamia się na systemach, gdzie domeną zarządza serwer z systemem Small Business Server 2003 (SBS), a komputery klienckie pracują na systemach Windows XP SP2 i XP SP3, Windows Vista oraz Windows 7.

## **Podsumowanie jakości**

Nie zauważono żadnego problemu związanego z konfiguracją serwerową.

Podczas testów nie zauważono różnic (nie wykryto defektów) w użytkowaniu aplikacji dla kont użytkownika standardowego i administratora.

Konfiguracja sprzętowa nie miała żadnego wpływu na pojawienie się błędów.

## **Zadania wykonane**

Przeprowadzono testy funkcjonalne i wydajności całej aplikacji.

## **Wymagania sprzętowe**

Ustalono minimalne wymagania dla sprzętu to: Celeron Dual-Core T3500 @ 2,10 GHz, 2GB RAM

Rekomendowane wymagania dla sprzętu: Celeron Dual-Core T3500 @ 2,40 GHz, 3GB RAM i wyższe

Lista defektów, wyniki testów funkcjonalnych i wydajnościowych wraz ze zrzutami ekranu znajduje się w załączniku.

## **Rekomendacja**

Brak krytycznych błędów kwalifikuje oprogramowanie do wdrożenia.

1. W załączonych dokumentach z wynikami testów przyjęto oznaczenia: Jeśli dana funkcjonalność działa poprawnie, to została oznaczona kolorem zielonym.
2. Jeśli testowana funkcjonalność wymagała komentarza, to została oznaczona kolorem żółtym (brak dokładnej specyfikacji oprogramowania).
3. Jeśli podczas wykonywania testu danej funkcji znaleziono defekt lub defekty, to oznacza się go kolorem czerwonym jako niedziałającą.

## **Załączniki**

Lista kontrolna funkcji znajduje się w pliku funkcje\_listy\_kontrolne.xls

Lista defektów wraz z określonymi konfiguracjami sprzętowymi i systemowymi dostępna jest w narzędziu do zarządzania zgłoszeniami.

Wykaz z testów wydajności poszczególnych konfiguracji w pliku wydajność.xls.

Załącznik funkcje\_listy\_kontrolne.xls, zawierający wyniki uruchomienia testów dla jednej z testowanych konfiguracji sprzętowej.

ID	Komponent/funkcja	Podfunkcja	Działa/nie działa	Uwagi
1	Autoodtwarzanie		Działa	
2	Menu główne		Działa	
3	Uruchomienie lekcji		Wątpliwość	Niesprecyzowane wymagania: wymagany Adobe Flash Player Active X
4	Nawigacja		Działa	
5		Przeciąganie strony	Działa	
6		Przewracanie strony	Działa	
7		Poruszanie się między stronami	Działa	
8		Powiększanie strony	Działa	
9		Powrót do pierwszej strony	Działa	
10		Poprzednia strona	Działa	
11		Numeracja stron	Działa	
12		Następna strona	Działa	
13		Przejsięcie do ostatniej strony	Działa	
14		Szybkie wyszukiwanie	Działa	
15	Toolbar		Nie działa	Defekt nr 1 i 2
16		Przegląd miniatur	Działa	
17		Drukowanie	Działa	
18		Ustawienia	Działa	
19		O nas	Działa	
20		Pomoc	Działa	

21		Tworzenie zakładki	Nie działa	Defekt nr 3
22		Sporządzenie notatki	Nie działa	Defekt nr 4 i 5
23		Zakreślacz	Nie działa	Defekt nr 6
24		Pełny ekran	Nie działa	Defekt nr 7
25	Odtwarzanie audio/wideo		Działa	
26	Słowniczek		Działa	
27	Pokaz slajdów/powiększanie obrazków		Działa	
28	Ćwiczenia		Działa	

## Projekt 5 – testy aplikacji mobilnej

Testowanie aplikacji mobilnych ma również swoją specyfikę i należy ją uwzględnić podczas przeprowadzania testów.

### Zlecenie testów

Firma zlecająca oprócz oprogramowania udostępniła również uproszczoną specyfikację testów aplikacji.

### Uruchomienie testów

Testowanie opierało się na wskazanej puli obsługiwanych urządzeń. Część testów została dostarczona przez firmę zlecającą, a reszta obszaru została pokryta przez testy eksploracyjne.

### Raport z testów

Szablon raportu został uzgodniony z klientem. Defekty były zgłaszane w zewnętrznym projekcie. W tym przykładzie można zaobserwować odniesienie się do wcześniej przeprowadzanych testów. W zaprezentowanym tu raporcie pokazano kilka przykładowych defektów. Usunięto dane mogące zidentyfikować produkt. Są to jedynie fragmenty całego raportu.

## RAPORT Z TESTÓW

ID: AM-mobilna-5

### Wstęp

W niniejszym raporcie przedstawione są wyniki prac nad drugą, kolejną iteracją przeprowadzenia testów w aplikacji XMobile. Prace zrealizowane zostały na podstawie wysokopoziomowych przypadków testowych (podanych w pierwszej iteracji testów przez klienta) oraz własnych – stworzonych na potrzeby dokładniejszego pokrycia funkcjonalności aplikacji. Odniesiono się również do raportu z pierwszej tury testów.

### Wyniki testów

Raporty o defektach znajdują się w narzędziu do raportowania defektów.

W raporcie przyjęto następujące oznaczenia dla wyników testów:

1. Pozytywny – wynik uruchomionego testu na podstawie jest pozytywny.
2. Negatywny – wynik uruchomionego testu jest negatywny.
3. Dopuszczony – oznacza, że działanie aplikacji jest zgodne z opisem przypadku testowego, ale są drobne zastrzeżenia.

ID	Kroki	Oczekiwany wynik	Wynik testu
1	1 Pierwsze urządzenie się łączy. Drugie czeka	Drugi użytkownik dostaje komunikat o oczekiwaniu na połączenie	Dopuszczony
2	1 Pierwsze urządzenie oczekuje na połączenie (operator nie odbiera)	Użytkownik i operator dostają odpowiednie informacje	Negatywny
	2 Drugie urządzenie dzwoni	Drugie urządzenie przejmuje pierwszeństwo w kolejce	
	3 Operator odbiera połączenie	Łączy się drugie urządzenie, pierwsze oczekuje dalej i może się połączyć dopiero po zakończeniu połączenia z drugim urządzenie	
3	1 Pierwsze urządzenie się łączy	Trwa transmisja z pierwszym urządzeniem	Negatywny
	2 Drugie i trzecie urządzenie dzwoni	Urządzenia oczekują na połączenie, pojawiają się odpowiednie komunikaty dla operatora i użytkownika	
	3		



		Urządzenie pierwsze kończy rozmowę	Urządzenie drugie jest przełączane do operatora	
	4	Urządzenie drugie kończy rozmowę	Urządzenie trzecie jest przełączane do operatora	
4	1	Pierwsze urządzenie się łączy	Trwa transmisja z pierwszym urządzeniem	Dopuszczony
	2	Dzwoni drugie i trzecie urządzenie	Urządzenia oczekują na połączenie, pojawiają się odpowiednie komunikaty dla operatora i użytkownika	
	3	Urządzenie drugie się rozłącza (z kolejki)	Użytkownik nr 3 dostaje informację, że teraz jest następny w kolejce	
	4	Urządzenie pierwsze się rozłącza	Urządzenie trzecie jest przełączane do przewodnika	
5	1	Urządzenie pierwsze dzwoni Urządzenie drugie dzwoni Urządzenie trzecie dzwoni	Dopóki nie zostanie odebrane połączenie, to osoba ostatnio dzwoniąca ma pierwszeństwo. W tej sytuacji kolejność jest więc 3, 2, 1. Przewodnik i użytkownicy mają odpowiednie komunikaty	Negatywny
	2	Operator odbiera rozmowę	Pierwsze łączy się urządzenie trzecie	
5	3	Urządzenie trzecie kończy rozmowę	Następne jest urządzenie drugie	Negatywny
	4	Urządzenie drugie kończy rozmowę	Łączy się urządzenie pierwsze	
6	1	Pierwsze urządzenie się łączy (operator 1)	Trwa transmisja z pierwszym urządzeniem	Pozytywny
	2	Drugie urządzenie się łączy	Ponieważ trwa transmisja z op. 1, drugie urządzenie jest przekierowywane do op. 2. Trwa równoległa transmisja do obu operatorów	
7	1	Pierwsze urządzenie się łączy (op. 1)	Trwa transmisja z pierwszym urządzeniem	Dopuszczony
	2	Drugie urządzenie się łączy (op. 2)	Trwa transmisja z drugim urządzeniem	
	3			

		Dzwoni urządzenie trzecie	Ze względu na transmisję u obu operatorów użytkownik trzeci oczekuje na połączenie. Są wysyłane odpowiednie komunikaty	
	4	Urządzenie drugie kończy połączenie	Urządzenie trzecie jest przełączane do operatora 2	
8	1	Pierwsze urządzenie się łączy (op. 1)	Trwa transmisja z pierwszym urządzeniem	Dopuszczony
	2	Drugie urządzenie się łączy (op. 2)	Trwa transmisja z drugim urządzeniem	
	3	Dzwoni urządzenie trzecie	Ze względu na transmisję u obu operatorów, użytkownik trzeci oczekuje na połączenie. Są wysyłane odpowiednie komunikaty	
	4	Urządzenie pierwsze kończy połączenie	Urządzenie trzecie jest przełączane do operatora 1	
9	1	Urządzenie pierwsze i drugie się łączy (odpowiednio 1. i 2. operator)	Trwa transmisja obu urządzeń	Negatywny
	2	Dzwoni urządzenie trzecie i czwarte	Urządzenia oczekują na połączenia. Wysyłane są odpowiednie komunikaty	
9	3	Urządzenie pierwsze się rozłącza	Pierwsze z urządzeń w kolejce (4te) łączy się z operatorem 1	Negatywny
	4	Urządzenie drugie się rozłącza	Urządzenie trzecie (jedyne w kolejce) łączy się z operatorem 2	
10	1	Użytkownik 1 dzwoni	Operator odbiera – transmisja trwa	Dopuszczony
	2	Użytkownik 2 dzwoni	Dostaje informację, że musi czekać (że jest 2. w kolejce)	
	3	Użytkownik 2 się rozłącza		
	4	Użytkownik 2 stara się połączyć ponownie (połączenie z 1 dalej trwa)	Dostaje informację, że musi czekać	
11	1	Użytkownik 1 dzwoni	Operator nie odbiera	Negatywny
	2	Użytkownik 2 dzwoni	Operator nie odbiera	
	3	Operator odrzuca użytkownika 1	Kolejny użytkownik w kolejce dzwoni do operatora	

12	1	Użytkownik dzwoni	Operator odbiera	Negatywny
	2	Operator rozłącza użytkownika	Użytkownik dostaje komunikat głosowy o rozłączeniu	
13	1	Użytkownik 1 dzwoni	Operator 1 i operator 2 nie odbierają	Pozytywny
	2	Użytkownik 2 dzwoni	Operator 1 i operator 2 nie odbierają	
	3	Operator 1 odbiera użytkownika pierwszego w kolejce	Użytkownik pierwszy w kolejce jest łączony do operatora 1	
	4	Operator 2 odbiera użytkownika następnego w kolejce	Następny użytkownik w kolejce jest łączony do operatora 2	
14	1	Operator włącza pełen ekran	Pełen ekran jest włączony, wszystko jest widoczne	Negatywny
	2	Operator wyłącza pełen ekran	Pełen ekran jest wyłączony, wszystko jest widoczne	
15	1	Operator 1 jest dostępny	Operator 1 jest pierwszy w kolejce	Pozytywny
	2	Operator 2 jest dostępny	Operator 2 jest drugi w kolejce	
	3	Operator 1 ma awarię	Operator 1 jest usunięty z kolejki, Operator 2 wskakuje na pierwsze miejsce w kolejce	
	4	Użytkownik 1 dzwoni	Operator 2 odbiera połączenie od użytkownika 1	
16	1	Użytkownik 1 jest połączony	Połączenie z operatorem 1	Pozytywny
	2	Użytkownik 1 ma awarię – utrata łączności	Połączenie jest przerywane, operator 1 ma możliwość odbierania połączeń	
	3	Użytkownik 2 dzwoni	Operator 1 odbiera połączenie od użytkownika 2	
17	1	Operator 1 i 2 są dostępni	Odpowiednie miejsce w kolejce	Negatywny
	2	Dzwoni użytkownik	Użytkownik jest kierowany do operatora 1	
	3	Operator 1 ma	Operator 1 jest usunięty z kolejki, a operator 2 ma pierwsze miejsce w kolejce i odbiera	

	awarię w trakcie dzwonienia użytkownika 1	połączenie od użytkownika	
--	---	------------------------------	--

### **Opis jakości**

Wszystkie zgłoszenia z raportu z 1 iteracji testów zostały zweryfikowane pod względem ich dalszego występowania. Nie zaobserwowano już następujących defektów (numery ID incydentów z poprzednich testów): ID 8, ID 11, ID 13, ID 14, <...>.

W aplikacji brakuje odpowiednich komunikatów dla operatorów i użytkowników. Brakuje informacji o liczbie osób czekających w kolejce na połączenie oraz informacji dla użytkownika o jego pozycji w kolejce. Obserwuje się nadal łączenie użytkowników w nieprawidłowej kolejności – jednak jest to zależne od tego, w jakim odstępie czasowym dany użytkownik wysłał prośbę o połączenie w stosunku do poprzedniego użytkownika wysyłającego prośbę.

W aplikacji w dalszym ciągu stwierdzono:

- niepoprawne działanie połączenia ze znajomym z listy znajomych,
- niepoprawne działanie zamknięcia aplikacji,
- brak transmisji obrazu w przypadku, gdy ekran zostaje zablokowany.

Podczas przeprowadzonych testów zostało zgłoszone 20 incydentów.

### **Podsumowanie**

Testy miały na celu sprawdzenie poprawności działania aplikacji. Spełnione zostały wszystkie założenia przyjęte przed rozpoczęciem prac. Zrealizowano wszystkie określone w poprzednich testach przypadki testowe. Zweryfikowano raport z pierwszej tury testów.

# Bibliografia\*

- 610-1990 – *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, <https://standards.ieee.org/findstds/standard/610n-1990.html> (dostęp: 12.10.2015).
- 830-1998 – *IEEE Recommended Practice for Software Requirements Specifications*, <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=720574&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel4%2F5841%2F15571%2F1%2F0%2F0%2F0.pdf> (dostęp: 12.10.2015).
- 1233-1998 – *IEEE Guide for Developing System Requirements Specifications*, <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=741940&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel4%2F5982%2F16016%2F1%2F0%2F0.pdf> (dostęp: 12.10.2015).
- Bach J., *A Tester's Commitments*, <http://www.satisfice.com/blog/archives/652> (dostęp: 12.10.2015).
- Bach J., *Exploratory Testing Explained*, <http://www.satisfice.com/articles/et-article.pdf> (dostęp: 12.10.2015).
- Bach J., *Heuristics of Software Testability*, <http://www.satisfice.com/tools/testable.pdf> (dostęp: 12.10.2015).
- Bach J., *Heuristics Test Strategy Model*, <http://www.satisfice.com/tools/htsm.pdf> (dostęp: 12.10.2015).
- Bach J., Bolton M., *Rapid Software Testing*, <http://www.satisfice.com/rst.pdf> (dostęp: 12.10.2015).
- Bach J., Bolton M., *Testing and Checking Refined*, <http://www.satisfice.com/blog/archives/856> (dostęp: 12.10.2015).
- Black R., *What IT Managers Should Know about Testing: How to Analyze the Return on the Testing Investment*, <http://www.rbcs-us.com/images/documents/What-IT-Managers-Should-Know-about-Testing-ROI.pdf> (dostęp: 12.10.2015).
- BS 7925 - 2. *Standard for Software Component Testing*, <http://edu.ittraining.pl/material/Norma-BS-7925-2-Component-Testing> (dostęp: 12.10.2015).
- Certyfikowany Tester, Sylabus dla Poziomu Zawansowanego Kierownika Testów*, [http://sjsi.org/wp-content/uploads/2014/04/SylabusPoziomuZawansowanegoTM\\_2012\\_WersjaBETA.pdf](http://sjsi.org/wp-content/uploads/2014/04/SylabusPoziomuZawansowanegoTM_2012_WersjaBETA.pdf) (dostęp: 12.10.2015).
- Certyfikowany tester. Plan poziomu podstawowego*, [http://sjsi.org/wp-content/uploads/2013/08/sylabus-poziomu-podstawowego-wersja-2011.1.1\\_20120925.pdf](http://sjsi.org/wp-content/uploads/2013/08/sylabus-poziomu-podstawowego-wersja-2011.1.1_20120925.pdf) (dostęp: 12.10.2015).

Cherry C., *25 Mantras That Will Make You A Happier Tester*, <http://itesting.com.au/2013/03/13/25-mantras-that-will-make-you-a-happier-tester> (dostęp: 12.10.2015).

Crispin L., Gregory J., *Agile Testing: A Practical Guide for Testers and Agile Teams*, Addison-Wesley, 2009.

Edgren R., Emilsson H., Jansson M., *Charakterystyki jakości oprogramowania*, [http://testerzy.pl/materialy/TheTestEye\\_SoftwareQualityCharacteristicsV1\\_1\\_PL.pdf](http://testerzy.pl/materialy/TheTestEye_SoftwareQualityCharacteristicsV1_1_PL.pdf) (dostęp: 12.10.2015).

Edgren R., *Many models – Better test ideas*, <http://thetesteye.com/blog/2012/02/many-models-better-test-ideas> (dostęp: 12.10.2015).

Gerrard P., Windsor S., *Specification by Example. How Validating Requirements Helps Users Developers and Testers*, [http://www.es.sogeti.com/PageFiles/173/Specification by Example\\_How Validating Requirements Helps Users Developers and Testers\\_Paul Gerrard and Susan Windsor.pdf](http://www.es.sogeti.com/PageFiles/173/Specification%20by%20Example_How%20Validating%20Requirements%20Helps%20Users%20Developers%20and%20Testers_Paul%20Gerrard%20and%20Susan%20Windsor.pdf) (dostęp: 30.08.2015).

Heusser M., *Classic Testing Mistakes*, <http://docslide.us/documents/classic-mistakes-in-software-testing.html> (dostęp: 12.10.2015).

IEEE Standard for Software Reviews and Audits, <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4601582> (dostęp: 12.10.2015).

IEEE Standard for Software Test Documentation, <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5976> (dostęp: 12.10.2015).

ISO 9000 – Quality management, [http://www.iso.org/iso/iso\\_9000](http://www.iso.org/iso/iso_9000) (dostęp: 12.10.2015).

ISO 9241-210:2010, Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems ISO 13407: Human-centred design processes for interactive systems, [http://www.iso.org/iso/home/store/catalogue\\_ics/catalogue\\_detail\\_ics.htm?csnumber=52075](http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=52075) (dostęp: 12.10.2015).

ISO/IEC 25010:2011, Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, [http://www.iso.org/iso/catalogue\\_detail.htm?csnumber=35733](http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733) (dostęp: 12.10.2015).

ISO/IEC 9126-1:2001, [http://www.iso.org/iso/catalogue\\_detail.htm?csnumber=22749](http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749) (dostęp: 12.10.2015).

ISO/IEC/IEEE 29119 Software Testing Standard, <http://www.softwaretestingstandard.org> (dostęp: 12.10.2015).

Jacobsen E., „Escape” Review Meeting Model, <http://www.testthisblog.com/2012/06/escape-review-meeting-model.html> (dostęp: 12.10.2015).

Kaner C., *A Tutorial in Exploratory Testing*, <http://www.kaner.com/pdfs/QAExploring.pdf> (dostęp: 12.10.2015).

Kelly M., *Touring Heuristic*, <http://michaeldkelly.com/blog/2005/9/20/touring-heuristic.html> (dostęp: 12.10.2015).

Kohl J., *Test Mobile Applications with I SLICED UP FUN!*, <http://www.kohl.ca/articles/ISLICEDUPFUN.pdf>, (dostęp: 12.10.2015).

Kruchten P., *The (missing) value of software architecture*, <http://philippe.kruchten.com/2013/12/11/the-missing-value-of-software-architecture> (dostęp: 12.10.2015).

McConnell S., *Kod doskonały*, Helion, 2010.

Meurs M., *10 Signs that you're not cut to be a tester*, <http://commontestsense.blogspot.com/2012/09/10-signs-that-youre-not-cut-to-be-tester.html> (dostęp: 12.10.2015).

Miller S., *The Seven Habits of Highly Effective Testers*, <http://blog.smartbear.com/software-quality/the-seven-habits-of-highly-effective-testers> (dostęp: 12.10.2015).

Page A., *Musings on Test Design*, <http://angryweasel.com/blog/?p=502> (dostęp: 12.10.2015).

Patton R., *Software Testing (2nd Edition)*, Sams Publishing, 2005.

Perry W., *Effective Methods for Software Testing*, Wiley Publishing, 2006.

*Project management triangle*, [https://en.wikipedia.org/wiki/Project\\_management\\_triangle](https://en.wikipedia.org/wiki/Project_management_triangle) (dostęp: 12.10.2015).

*Rozporządzenie Ministra Pracy i Polityki Społecznej z dnia 27 kwietnia 2010 r. w sprawie klasyfikacji zawodów i specjalności na potrzeby rynku pracy oraz jej stosowania*. Dz.U. 2010, nr 82, poz. 537, <http://isap.sejm.gov.pl/DetailsServlet?id=WDU20100820537> (dostęp: 12.10.2015).

*Słownik wyrażeń związanych z testowaniem (v.2.3)*, <http://sjsi.org/wp-content/uploads/2014/10/slownik-termin%C3%B3w-testowych-ver-2.3-PL.pdf> (dostęp: 12.10.2015).

*Test Heuristics Cheat Sheet Data Type Attacks & Web Tests*, <http://testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf>, (dostęp: 12.10.2015).

*The Economic Impacts of Inadequate Infrastructure for Software Testing*, <http://www.nist.gov/director/planning/upload/report02-3.pdf> (dostęp: 12.10.2015).

Whittaker J., *How to break software*, Addison-Wesley, 2002.

Woolf M., *Big List of Naughty Strings*, <https://github.com/minimaxir/big-list-of-naughty-strings> (dostęp: 12.10.2015).

Zangrando L., *Pretotyping*, <http://www.pretotyping.org> (dostęp: 12.10.2015).

# Przypisy

\* Część materiałów pochodzi z nieistniejącego już forum [testy.org.pl](http://testy.org.pl). Duża część materiału pochodzi z moich publikacji na portalu [testerzy.pl](http://testerzy.pl).