DO NOWEJ PODSTAWY PROGRAMOWEJ

Część 1

Inżynieria programowania – projektowanie oprogramowania, testowanie i dokumentowanie aplikacji

Kwalifikacja INF.04

Projektowanie, programowanie i testowanie aplikacji

Podręcznik do nauki zawodu
technik programista

Angelika Krupa, Weronika Kortas

Helion

Redaktor prowadzący: Joanna Zaręba

Projekt okładki: Jan Paluch

Ilustracja na okładce została wykorzystana za zgodą Shutterstock.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?inf041_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-7545-1

Spis treści

Wstęp

Programista, tester czy analityk to zawody cechujące się dynamicznym postępem technologicznym. Ważne jest, by cały czas się rozwijać i nie stać w miejscu. Warto poznawać nowe metodyki wytwarzania oprogramowania czy narzędzia wspierające procesy wytwórcze. W tym podręczniku znaleźć można opis najważniejszych etapów składających się na proces tworzenia aplikacji. Omówiono narzędzia niezbędne na każdym z nich (rozdział 1.). Zaprezentowano podstawy programowania w języku JavaScript (rozdział 2.). Przedstawiono zasady dobrego programowania (zasada czystego kodu, stosowanie algorytmów, wstęp do wzorców projektowych itp. — rozdział 3.). Opisano proces testowania powstającego i gotowego programu (rozdział 4.). Zaprezentowano przykłady dokumentacji wytwarzanej w procesie testowania (rozdział 5.). W rozdziale 6. omówiono metodologie prowadzenia projektu programistycznego. Zwieńczeniem podręcznika jest rozdział 7., który ma pokazać czytelnikom krok po kroku etapy tworzenia aplikacji z uwzględnieniem poznanych wcześniej wiadomości.

Treści zawarte w podręczniku są zgodne z podstawą programową dla zawodu **technik programista**. Są one powiązane merytorycznie, strukturalnie i metodycznie z ogólnymi *celami kształcenia* określonymi w kwalifikacji *INF.04. Projektowanie, programowanie i testowanie aplikacji*:

- **a)** projektowanie, programowanie i testowanie zaawansowanych aplikacji webowych;
- **b)** projektowanie, programowanie i testowanie aplikacji desktopowych;
- **c)** projektowanie, programowanie i testowanie aplikacji mobilnych.

W szczególności treści prezentowane w niniejszym podręczniku dotyczą *efektów kształcenia* wraz *kryteriami ich weryfikacji*, które zostały opisane w punktach *INF.04.3. Projektowanie oprogramowania* oraz *INF.04.8. Testowanie i dokumentowanie aplikacji* podstawy programowej.

Zgodnie z uwagami przedstawionymi powyżej niniejszy podręcznik powinien stanowić dla ucznia uzupełnienie i rozszerzenie materiału zawartego w podręcznikach do nauki programowania zaawansowanych aplikacji webowych, desktopowych i mobilnych. Przy tym dotyczy to zarówno (początkowego) etapu projektowania aplikacji, jak i (końcowych) etapów jej testowania i dokumentowania.

Przeczytanie (ze zrozumieniem) tego podręcznika jest bardzo czasochłonne i wymaga sporego zaangażowania. Mamy jednak nadzieję, że osiągnięty cel będzie wart wysiłku włożonego w lekturę.

Pisząc podręcznik, dążyłyśmy do tego, by zawrzeć w nim jak najwięcej praktycznych przykładów. Wyszłyśmy z założenia, że żadne słowa nie zastąpią godzin praktyki, dla których opanowanie teorii jest zaledwie niezbędnym wstępem.

Główne źródła wiedzy dotyczącej testowania i wytwarzania oprogramowania to:

1. Certyfikowany tester. Sylabus poziomu podstawowego ISTQB®. Wersja 2018 V 3.1. Prawa autorskie wersji polskiej zastrzeżone dla © Stowarzyszenie Jakości Systemów Informatycznych (SJSI).
2. Słownik terminów testowych®. Wersja 3.3.1. Prawa autorskie wersji polskiej zastrzeżone dla © Stowarzyszenie Jakości Systemów Informatycznych (SJSI).
3. Certyfikowany Tester. Sylabus rozszerzenia poziomu podstawowego. Tester zwinny. Wersja 2014.

Skuteczna nauka wymaga czasu. Wytwarzanie oprogramowania, ze względu na zespołowy charakter tego procesu, bazuje w dużej mierze na komunikacji. Z tego względu, aby ułatwić równoczesną pracę nad projektem, opiera się ją na wybranych metodykach, w szczególności zwinnych (Agile). Mówi się, że jeden dzień poświęcony na analizę problemu, z którym chcemy się zmierzyć, to dwa tygodnie tworzenia kodu (tak — dokładnie dwa tygodnie). Dlatego tak duży nacisk położono w podręczniku na planowanie i testowanie. Testy oprogramowania, przeprowadzane na różnych etapach pracy nad nim, są nieodzownym elementem tworzenia aplikacji. To od nich zależy, czy klient zlecający nam wykonanie programu zostanie z nami na dłużej i czy zamówi u nas kolejne projekty. Jeśli już na początku przytłoczy go liczba błędów w aplikacji, szanse na dalszą współpracę będą znikome.

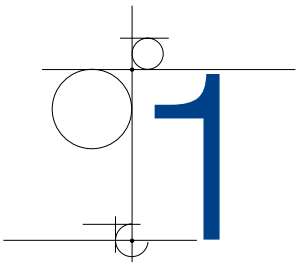
Podczas podróży po świecie wytwarzania oprogramowania warto pamiętać o tym, że najlepszym nauczycielem są właśnie błędy — zarówno własne, jak i te, które ktoś już popełnił przed nami. Warto się na nich uczyć i nieustannie dążyć do ich eliminowania. Mimo licznych wyrzeczeń i wielogodzinnej pracy pierwsze projekty mogą mieć wiele niedoskonałości. Jak można minimalizować liczbę błędów? Tylko przez praktykę. Im więcej projektów stworzymy, starając się stosować zasady dobrego programowania, tym mniej błędów będą zawierać nasze projekty.

Musimy jednak pamiętać, by uczyć się na błędach, a nie uczyć się błędów! Powielanie w kolejnych projektach błędu popełnionego w pierwszym programie, spowodowane brakiem wiedzy bądź niedbałością, na pewno nie jest dobrą praktyką.

Podczas realizacji projektu końcowego dobrze jest pracować w zespole. Samodzielna praca jest oczywiście możliwa, ale z reguły trwa dłużej i jest trudniejsza. Jest ponadto w pewnym sensie nienaturalna, bo obecnie niemal wszystkie firmy programistyczne opierają się na zespołach współpracujących programistów i testerów oraz analityków i Scrum Masterów. Zachęcamy do budowania zespołów nastawionych na wspólną pracę, której podstawą będzie dobra komunikacja. To ona jest kluczem do sukcesu w wielu wymiarach, także w przypadku tworzenia oprogramowania. Niezwykle ważne jest również to, żeby każdy członek zespołu znał dokładnie role i możliwości pozostałych — to na pewno usprawni komunikację.

Po tych wstępnych informacjach czas zacząć przygodę w świecie programowania. Mamy nadzieję, że niniejszy podręcznik będzie dla Ciebie najlepszym przewodnikiem.

Autorki



1 Przygotowanie środowiska pracy

Przygotowanie środowiska pracy to nie tylko instalacja i konfiguracja niezbędnych aplikacji i narzędzi, ale również nauka optymalnego ich wykorzystania. Obecnie wszelkiego rodzaju tutoriale i szkolenia e-learningowe niemal „wylewają się" z internetu, dlatego coraz częściej wymagana jest znajomość wielu narzędzi już na początku pracy — po to by proces tworzenia oprogramowania był jak najszybszy i jak najbardziej wydajny.

1.1. Narzędzia wykorzystywane w prowadzeniu projektu

W dzisiejszych czasach realizacja projektów jest tak różnorodna, że rynek nieustannie dostarcza nowych rozwiązań technologicznych, które mają za zadanie wspomagać procesy i zarządzanie projektami. W niniejszym rozdziale zostaną omówione narzędzia (zarówno płatne, jak i darmowe), które wspierają zarządzanie projektem, zarządzanie testami oraz wytwarzanie oprogramowania.

UWAGA

Rozdział ma charakter wprowadzający — warto mieć na uwadze, że niżej wymienione narzędzia cały czas są rozwijane i wzbogacane o kolejne funkcje.

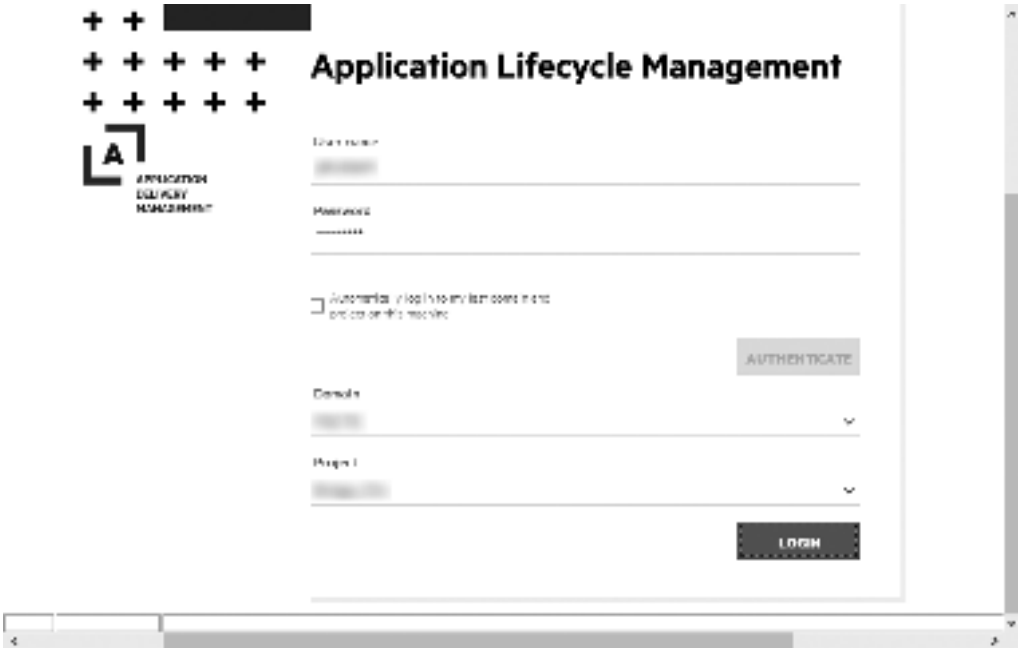
1.1.1. HP ALM

Jako pierwsze zostanie omówione narzędzie HP ALM. Jest to aplikacja do wspierania procesu testowego i zarządzania testami. Dostęp do narzędzia jest przyznawany na podstawie licencji (1 licencja = 1 użytkownik). Umożliwia ono planowanie oraz wykonanie testów manualnych i automatycznych, zarządzanie defektami (zgłaszanie, monitorowanie). HP ALM dzieli się na dwie części:

- administracyjną (ang. *Site Administration*);
- użytkownika (ang. *ALM Platform*) — cały moduł przeznaczony dla użytkowników.

W dalszej części rozdziału skupimy się na części użytkownika, gdyż panel administracyjny jest dostępny — jak sama nazwa wskazuje — dla osób pełniących funkcję administratora. W panelu administracyjnym wykonuje się wszystkie czynności administracyjne, takie jak tworzenie domen, projektów, użytkowników oraz nadawanie uprawnień.

Po wybraniu adresu docelowego narzędzia w danej organizacji pojawia się ekran logowania (rysunek 1.1):



Rysunek 1.1. Ekran logowania w HP ALM

Jak można zauważyć, poza wymaganymi danymi do logowania aplikacja wymusza na nas wybór z list rozwijanych domeny (ang. *Domain*) i projektu. Użytkownik może się zalogować wyłącznie do projektów, do których nadano mu dostęp.

UWAGA

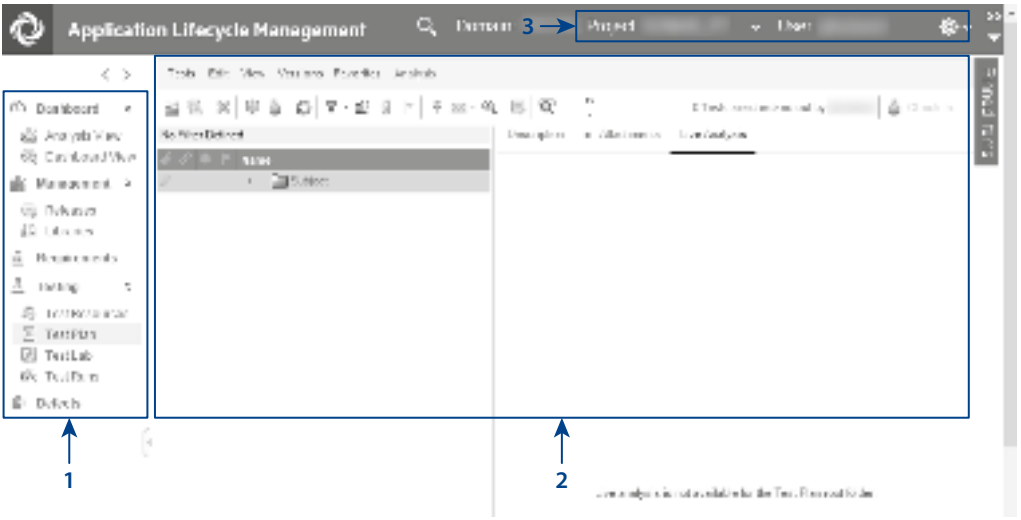
Po zalogowaniu widoczne są tylko te funkcje, do których mamy dostęp.

Panel użytkownika może się składać z następujących modułów (widoczność zależna od uprawnień):

- *Dashboard* — statystyki, raporty;
- *Management* — zarządzanie planem wersji;
- *Requirements* — definiowanie wymagań systemowych i zarządzanie nimi;
- *Testing* — tworzenie zestawów testów w projektach, uruchamianie zaplanowanych testów;
- *Defects* — tworzenie i śledzenie statusów defektów.

Poniżej przedstawiono podgląd widoku ekranu po zalogowaniu do aplikacji (rysunek 1.2). Widać na nim trzy grupy menu:

1. Menu wyboru modułów.
2. Menu operacji dostępnych w wybranym wcześniej module.
3. Menu użytkownika (wybór projektu, ustawienia konta).



Rysunek 1.2. Widok na menu i moduły w HP ALM

W dalszej części podrozdziału przedstawione zostaną moduły *Testing* i *Defects*.

1.1.1.1. Moduł Testing

Jest to moduł służący do organizowania procesu testowego. Składa się z następujących elementów (użytkownik widzi tylko te, do których ma nadany dostęp):

- *Test Resources* — zawiera wszystkie dodatkowe zasoby używane do testów manualnych oraz automatycznych, zarówno funkcjonalnych, jak i wydajnościowych.
- *Business Components* — moduł, który pozwala zarządzać komponentami biznesowymi.
- *Test Lab* — tu przenosimy testy, które są skonfigurowane i gotowe do uruchomienia.
- *Test Runs* — w module widoczne są wszystkie uruchomienia testów wraz z ich wynikami.
- *Test Plan* — moduł pozwalający na zorganizowanie całego procesu testowania poprzez określenie, jakie testy zostaną wykonane w celu zweryfikowania poprawności działania systemu.

DEFINICJA

Weryfikacja — egzaminowanie poprawności i dostarczenie obiektywnego dowodu na to, że produkt procesu wytwarzania oprogramowania spełnia zdefiniowane wymagania.

Walidacja — sprawdzenie poprawności i dostarczenie obiektywnego dowodu na to, że produkt procesu wytwarzania oprogramowania zaspokaja potrzeby użytkownika i spełnia jego wymagania.

1.1.1.2. Moduł Defects

Jest to moduł do zarządzania błędami, który daje możliwość ich rejestrowania, przypisywania do konkretnych osób i zmiany statusów zarejestrowanych błędów.

Możemy zgłosić tutaj wszystkie uwagi do zachowań (zarówno funkcjonalnych, jak i niefunkcjonalnych) systemu odbiegających od przyjętych wymagań.

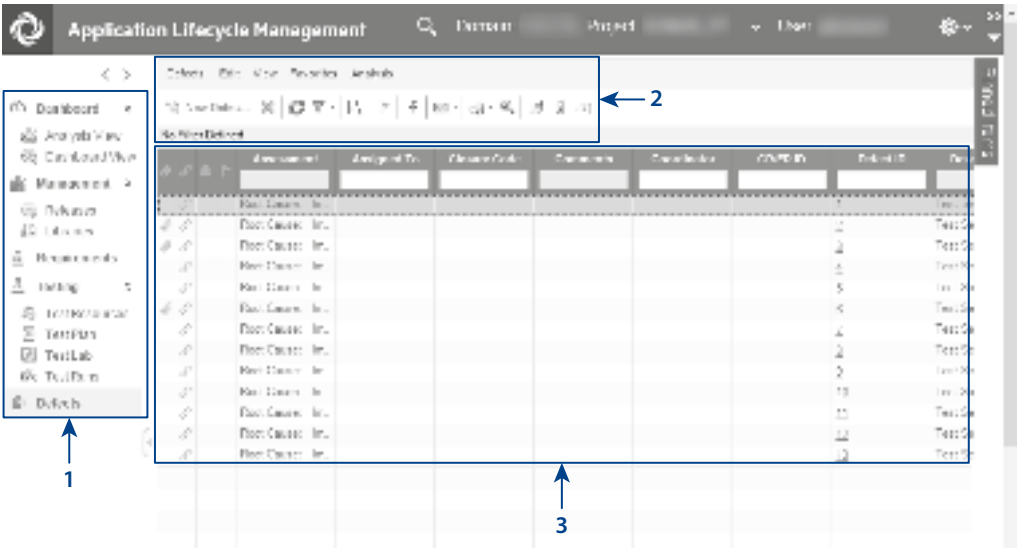
DEFINICJA

Wymaganie funkcjonalne dotyczy wyniku zachowania systemu, który powinien zostać dostarczony przez funkcję systemu.

Wymaganie niefunkcjonalne dotyczy jakości tworzonego oprogramowania, ale nie zostało określone przez wymagania funkcjonalne. Odnosi się m.in. do wydajności, dostępności, niezawodności, skalowalności i przenośności.

Poniżej przedstawiono podgląd widoku ekranu modułu *Defects* (rysunek 1.3). Można na nim zauważyć trzy grupy menu:

1. Menu wyboru modułów.
2. Menu górne.
3. Widok na wszystkie zgłoszenia w ramach projektu, do którego jesteśmy zalogowani.



Rysunek 1.3. Widok narzędzia w module Defects

Defekty możemy zgłosić zarówno z poziomu modułu *Test Lab*, jak i z poziomu *Defects*. W menu górnym (2) należy najpierw wybrać *New Defect*, a następnie wypełnić wszystkie wymagane pola w formularzu. Po dodaniu defektu jest on widoczny na liście (3).

HP ALM jest bardzo elastycznym narzędziem. Możemy je spersonalizować do każdego projektu. Co ważne — możemy sami definiować raporty czy widoki i udostępniać je innym osobom.

Jest to jedno z najdroższych narzędzi na rynku, jednak jest warte swojej ceny.

1.1.2. Jira

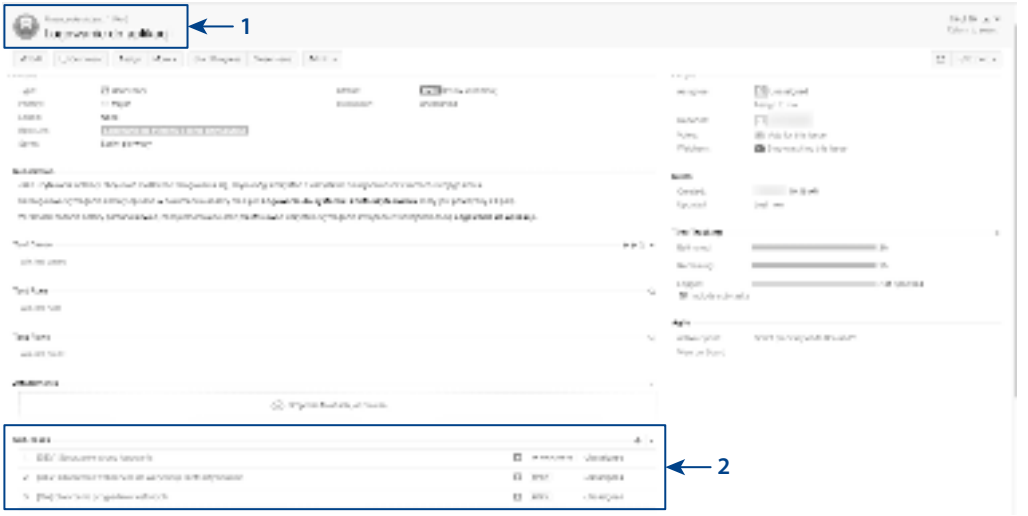
Jira jest komercyjnym narzędziem nie tylko wykorzystywanym w zarządzaniu projektami, ale również wspomagającym elektroniczny obieg dokumentacji w firmie.

Tak wygląda ekran logowania dla Jira cloud (rysunek 1.4):

Rysunek 1.4. Ekran logowania w narzędziu Jira

Warto zaznaczyć, że ekran logowania do wersji cloud różni się wizualnie od wersji serwerowej. Jira dostosowuje się do projektu. Zależnie od metodyki, w jakiej projekt jest realizowany, narzędzie daje nam całą paletę różnorodnych funkcji. Poniżej (rysunek 1.5) zaprezentowano widok przykładowego zadania w systemie Jira wraz z podzadaniami (ang. *subtasks*). Można na nim zauważyć, że:

1. zadanie główne wyświetlane jest z nazwą projektu;
2. widoczne są wszystkie podzadania powiązane z zadaniem głównym.



Rysunek 1.5. Widok projektu w systemie Jira

W momencie gdy wszystkie podzadania wejdą w status świadczący o zakończonych pracach (może to być *Resolved* lub *Closed*), uznaje się, że zadanie główne może być wdrażane do środowiska (docelowego) produkcyjnego.

DEFINICJA

Środowisko produkcyjne — zainstalowane w siedzibie klienta, firmy tworzącej oprogramowanie lub w chmurze sprzęt i oprogramowanie, w których moduł (lub system) będzie używany. W skład oprogramowania mogą wchodzić systemy operacyjne, bazy danych i inne aplikacje.

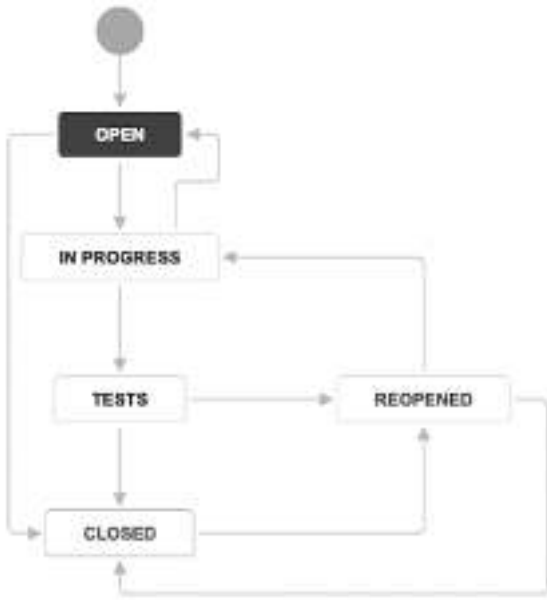
Dodatkowym atutem narzędzia Jira jest definiowanie cyklu życia (ang. *workflow*) błędu.

DEFINICJA

Cykl życia błędu to proces, przez który przechodzi usterka oprogramowania. Rozpoczyna się, gdy błąd zostanie wykryty, a kończy w chwili jego usunięcia, po wcześniejszym upewnieniu się, że nie został zduplikowany.

Na rysunku 1.6 przedstawiony jest diagram, który możemy dowolnie modyfikować. Każdy projekt może mieć odrębny *workflow*.

Rysunek 1.6.
Przykładowy cykl życia błędu (ang. workflow)



W *Panelu użytkownika* (rysunek 1.7) możemy zmienić preferencje co do wyświetlania stron oraz sprawdzić historię aktywności.

Rysunek 1.7. Konto użytkownika

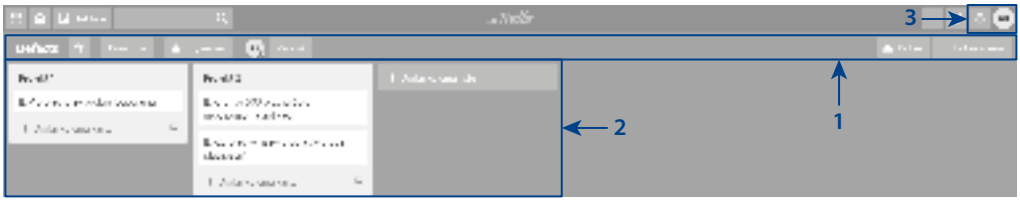
Dodatkowe informacje o narzędziu Jira zostaną podane w rozdziale 4.

1.1.3. Trello

Trello jest narzędziem bezpłatnym, które może być wykorzystywane zarówno w projektach komercyjnych, jak i w celach prywatnych (np. do planowania podróży, budżetu domowego, remontu).

Poniżej przedstawiono podgląd na widok tablicy Trello (rysunek 1.8). Można na nim zauważyć trzy grupy menu:

1. Menu operacji dotyczących udostępniania/prywatności tablicy.
2. Widok kart.
3. Konto użytkownika wraz z opcją powiadomień.



Rysunek 1.8. Widok na tablicę Trello

Dodatkowe informacje o Trello zostaną zaprezentowane w rozdziale 4.

W dalszej części podręcznika zostanie omówiony jeszcze TestLink — darmowe narzędzie typu *open source*.

1.1.4. Dobór narzędzi do zarządzania projektem

Poniżej przedstawiono krótkie zestawienie porównawcze narzędzi, które ułatwi Ci dopasowanie narzędzi do projektu. Zostały w nim uwzględnione ich kluczowe funkcje, dzięki którym wspierają prowadzenie projektu.

Tabela 1.1. Zestawienie porównawcze narzędzi

	HP ALM	Jira	Trello	TestLink
Płatna licencja	✓	✓		
Możliwość rozszerzania funkcji dzięki płatnym dodatkom	✓	✓	✓	
Personalizacja raportów	✓	✓		
Definiowanie cyklu życia błędu	✓	✓		
Wykonywanie przypadków testowych	✓			✓
Obsługa testów automatycznych	✓	✓		✓

1.2. Narzędzia wykorzystywane w wytwarzaniu oprogramowania

Jednym z elementów wytwarzania oprogramowania jest tworzenie aplikacji za pomocą wybranego języka programowania. W tym podręczniku skupiamy się na języku JavaScript. Zacznijmy od opisu elementów, na jakie należy zwrócić szczególną uwagę podczas instalacji oprogramowania.

Narzędzia pomocne do napisania aplikacji przeglądarkowej to Git i WebStorm.

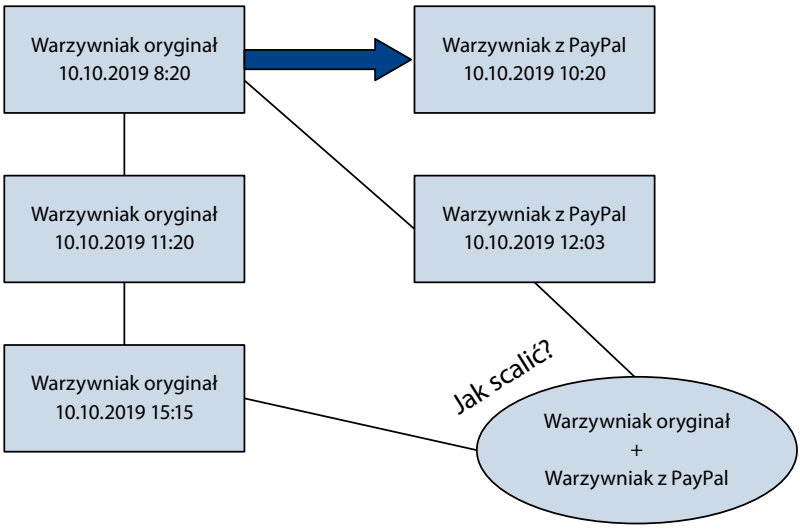
1.2.1. Instalacja programów WebStorm i Git

1.2.1.1. Git — system kontroli wersji

Wyobraźmy sobie wspólne pisanie fragmentów kodu przez wielu użytkowników i przenoszenie tych kodów za każdym razem do chmury. Jak zapewnić, by zmiany dodawane przez dowolnego programistę były widoczne dla pozostałych współpracowników?

Gdy kolega z zespołu pobierze plik, a następnie coś w nim zmieni, wówczas, by było wiadomo, że kod został zmieniony, można dodać katalog z datą i godziną zmiany i to w nim właśnie przechowywana będzie nowa wersja pliku, podczas gdy oryginalna pozostanie niezmieniona.

Załóżmy ponadto, że kolega pracuje nad nową funkcją. Skopiował więc cały katalog oryginalny i nanosi zmiany wewnątrz kopii folderu. Poprawki do bieżącej funkcji wciąż są natomiast rozwijane w oryginalnym folderze (rysunek 1.9).



Rysunek 1.9. Symulacja prowadzenia projektu zapisanego w folderach w chmurze

Pewnego dnia kolega skończył pisanie funkcji, nad którą pracował. Trzeba więc połączyć dwa różne katalogi. Gdyby nie istniały systemy kontroli wersji, trzeba byłoby porównać wszystkie pliki i nanosić ręcznie niezbędne zmiany. Przy dużej liczbie plików i zmian byłby to spory problem.

Na szczęście problem ten pozwalają łatwo rozwiązać wspomniane już narzędzia do kontroli wersji, a wśród nich to bodaj najpopularniejsze — Git.

DEFINICJA

Git to repozytorium kodu, czyli w uproszczeniu narzędzie do przenoszenia plików między komputerem, na którym jest tworzony kod, a innym serwerem, na którym przechowywana jest kopia. Zapewnia wersjonowanie plików (dlatego nazywane jest też systemem kontroli wersji), czyli przechowywanie w osobnych gałęziach (ang. *branch*), różnych wersji kodu pochodzących od jednej głównej wersji, nazywanej gałęzią główną (ang. *master*). Każda gałąź może być rozwijana niezależnie, a potem scalana do jednolitej wersji.

Przykład 1.1

Zobaczmy na przykładzie, jak rozwiązać problemy z integracją.

Od czasu do czasu w lokalnej społeczności organizowane są wyprzedaże garażowe. Załóżmy, że rodzina Nowaków chciałaby się przyłączyć do tej wyprzedaży i opróżnić już pokoje dzieci — Antka i Marysi — bo zalegają w nich pluszaki i ubrania, z których dzieci już wyrosły.

W pokoju Antka odnaleziono m.in. garnitur komunijny, pluszowego koalę oraz o trzy rozmiary za małe buty.

W pokoju Marysi odnaleziono za małe biurko, za niskie krzesło obrotowe oraz mnóstwo szpargałów z wakacji w Egipcie.

Antek zdecydował, że oddaje wszystkie rzeczy wstępnie zakwalifikowane do wystawienia na wyprzedaż. Przeniósł je do salonu, gdzie były składowane przedmioty do sprzedania.

Marysia nie mogła pożegnać się z pamiątkami z Egiptu. „Oddaję biurko i krzesło, ale pamiątki zostają" — orzekła.

Załóżmy, że organizację wyprzedaży przeprowadzamy za pomocą Gita. Najpierw musimy wybrać rzeczy na wyprzedaż. Robimy to za pomocą polecenia w postaci `git add nazwa_rzeczy`.

Wybranie rzeczy (przeniesienie ich do salonu) za pomocą Gita wyglądałoby więc tak:

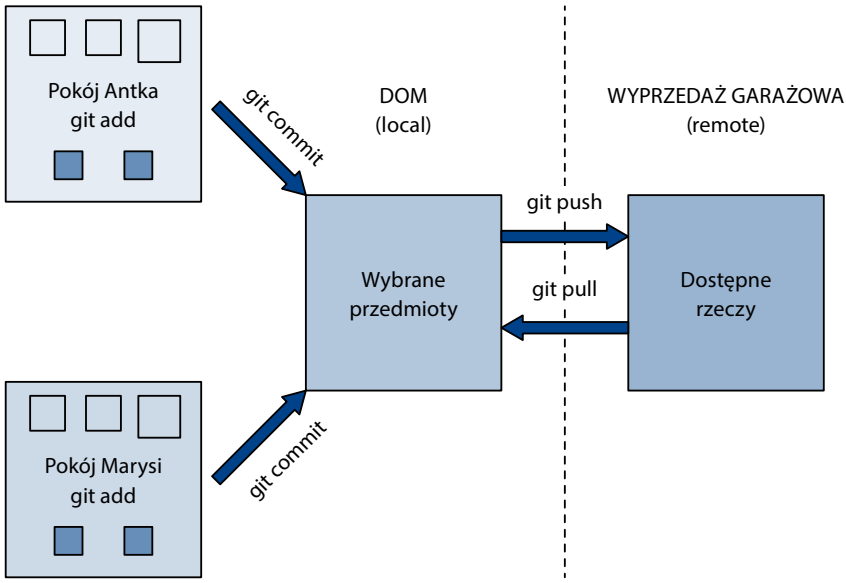
```
git add garnitur
git add pluszak
git add buty
git add biurko
git add krzeslo
```

Gdy wszystkie przedmioty zostały już wybrane (czyli przeniesione do salonu), podjęta została ostateczna decyzja zatwierdzająca wybór. Realizujemy ją za pomocą polecenia `git commit -m 'rzeczy na wyprzedaz'` (`-m` to parametr, który oznacza komentarz do commita, czyli wprowadzonych zmian). Nie znaczy to jednak, że rzeczy zostały już przeniesione do miejsca, gdzie będą sprzedawane, czyli do garażu sąsiada (to będzie nasze repozytorium zdalne — `remote`). Wciąż jeszcze znajdują się w salonie (czyli repozytorium lokalnym — `local`).

Następnie tata wyniósł rzeczy do garażu sąsiada. W tym momencie przedmioty zostały więc przeniesione z repozytorium lokalnego do zdalnego. Odpowiednia komenda to `git push`.

Wyprzedaże organizowane są po to, by rzeczy, które nam już nie są potrzebne, mogły zostać zakupione przez innych. My je „wypychamy" (`push`), dobrze byłoby jednak, by inni mogli je pobrać (`pull`). Aby to było możliwe, dla wybranych przedmiotów należy

wykonać polecenie `git pull`. Nie będziemy jednak przeprowadzać transakcji w garażu. Wybrane przez kupującego rzeczy ułożymy więc znów w naszym repozytorium lokalnym — w salonie (rysunek 1.10).



Rysunek 1.10. Ilustracja przepływu zadań (ang. workflow) w Gicie na przykładzie wyprzedaży garażowej

Podobnie będzie wyglądała praca w Gicie z kodem. Dopóki pliki są u nas na dysku, to są lokalne (ang. *local*); gdy już przerzucimy je do repozytorium, to oprócz plików lokalnych mamy też ich zdalne (ang. *remote*) kopie.

Przykład 1.2

Wróćmy do problemu z kodem źródłowym warzywniaka. Za pomocą systemu kontroli wersji można go rozwiązać w opisany tutaj sposób. Na początku kolega pracujący nad integracją sklepu z systemem płatności PayPal mógł zrobić kopię kodu z dnia 10.10.2019 i pracować właśnie na niej. Tak zrobioną kopię nazywamy gałęzią. Komenda, która służy do tworzenia nowej gałęzi, to `git checkout -b paypal` (`checkout` utworzy lokalnie gałąź o nazwie `paypal`; nazwa będzie zatem odpowiadać rozwijanej funkcji). Na końcu prac trzeba scalić wszystkie aktywne gałęzie z oryginalnym kodem źródłowym, przechowywanym w głównej (ang. *master*) gałęzi. Uzyskuje się to za pomocą komendy `git merge paypal` wykonywanej, gdy jesteśmy w gałęzi *master*.

Prace nad projektem rozpoczynamy od utworzenia projektu w dowolnym portalu, który obsługuje zarządzanie wersjami projektu (np. *github.com*, *bitbucket.org* — my będziemy pracowali w pierwszym z nich). Po założeniu konta w serwisie *github.com* (czyli serwerze z kodami źródłowymi) i utworzeniu projektu o nazwie *firstProject* uzyskamy możliwość

pobrania projektu poprzez bezpośredni link, który otrzymaliśmy. Będzie on wyglądać podobnie do tego: *https://github.com/nazwauzytkownika/firstProject.git*.

URL jest dostępny od razu po utworzeniu projektu, dzięki czemu możemy go udostępniać współpracownikom.

1.2.1.2. WebStorm — IDE

W repozytorium zdalnym Gita będziemy przechowywać nasz kod. Kod możemy pisać w dowolnym programie (może to być nawet notatnik), ale zrobimy to o wiele sprawniej, jeśli użyjemy IDE (akronim od ang. *Integrated Development Environment* — zintegrowane środowisko programistyczne), przede wszystkim ze względu na oferowane przezeń funkcje, jak uzupełnianie składni czy kontrola błędów. Możemy użyć np. programu WebStorm, który można pobrać ze strony producenta (wybieramy plik *.exe*):

https://www.jetbrains.com/webstorm/download/download-thanks.html?platform=windows

Dobrze jest zapoznać się z przykładowym uruchomieniem pierwszego projektu; opis tego procesu można znaleźć na stronie producenta:

https://www.jetbrains.com/help/webstorm/getting-started-with-webstorm.html

UWAGA

Jakie środowisko wybrać? Trudno odpowiedzieć na to pytanie. Wybór jest w tym przypadku (jak i w wielu innych) arbitralny, zależy głównie od indywidualnych preferencji użytkownika. Trzeba jednak zdawać sobie sprawę, że funkcjonalność środowisk programistycznych jest bardzo podobna (podstawowe funkcje to: kolorowanie kodu, autouzupełnianie poleceń, sygnalizowanie błędów składniowych, tryb debugowania kodu). Jeżeli opanujesz jedno, prawdopodobnie poradzisz sobie z pozostałymi. Zazwyczaj różnią się od siebie wyglądem, skrótami klawiszowymi i liczbą zintegrowanych narzędzi.

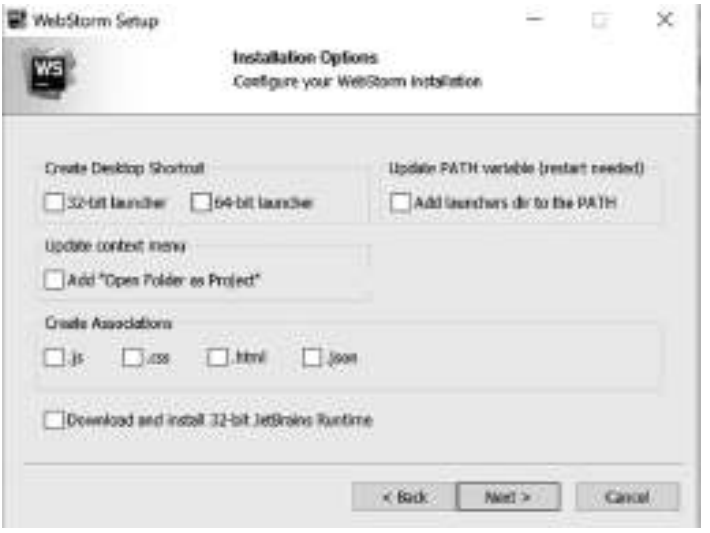
Przykłady prezentowane w tym podręczniku wykorzystują wspomniany już WebStorm. Jest to rozwiązanie komercyjne, jego autorzy jednak udostępniają darmową licencję do celów edukacyjnych lub też oferują zniżki dla start-upów. Do wyboru tego narzędzia skłania m.in. jego dobra integracja z narzędziem Jira.

Można jednak skorzystać z innych środowisk. Wiele z nich jest — jak WebStorm — komercyjnych, ich producenci udostępniają jednak darmowe wersje z nieco ograniczoną funkcjonalnością (przy czym dla początkującego programisty będzie ona zupełnie wystarczająca). Tak jest choćby w przypadku IntelliJ IDEA (*https://www.jetbrains.com/idea/*). Obok płatnej wersji Ultimate funkcjonuje darmowa, bez ograniczeń czasowych, wersja Community. Niestety ta ostatnia nie obsługuje języka JavaScript, dostępnego w wersji płatnej, jeśli jednak chcesz programować w innych językach (przede wszystkim w Javie), warto ją wypróbować. Visual Studio, program Microsoftu obsługujący m.in. JavaScript, przygotowano w aż trzech wersjach: komercyjnych Professional i Enterprise (obie posiadają bezpłatne wersje próbne) oraz darmowej Community. Wszystkie można pobrać ze strony *https://visualstudio.microsoft.com/pl/*.

UWAGA cd.

Istnieją także środowiska zupełnie darmowe, jak choćby te dostępne na licencji *open source* (tzn. że aplikacja jest tworzona przez programistów w ramach otwartego projektu i każdy może pobrać kod oraz zmodyfikować go na własny użytek, może też dopisać nowe funkcje i udostępnić je innym), takie jak Visual Studio Code (najczęściej używany edytor kodu w przypadku języków opartych na JavaScripcie, dostępny pod adresem *https://github.com/microsoft/vscode*), Netbeans (*https://netbeans.org/index_pl.html*) czy Eclipse (*https://www.eclipse.org/*). Wartym polecenia środowiskiem jest także Atom (*https://atom.io/*).

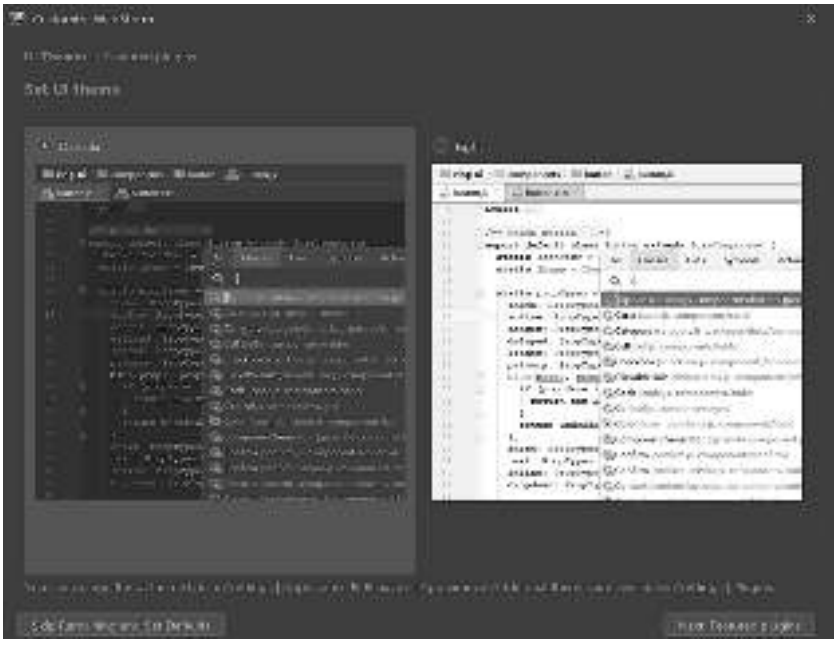
Podczas instalacji warto zwrócić uwagę na okno dotyczące opcji w sekcji *Create Associations* (rysunek 1.11) . Mamy do wyboru rozszerzenia plików, które mają być skojarzone z programem — to znaczy, że będą domyślnie otwierane przez WebStorm. Zalecamy wybranie plików z rozszerzeniami *.js* oraz *.html*.



Rysunek 1.11. Wybór opcji dostępnych podczas instalacji programu WebStorm

Przy pierwszym uruchomieniu pojawi się okno wyboru stylu programu. Spora grupa programistów ze względu na minimalizowanie odbicia światła wybiera ciemny styl *Darcula* (rysunek 1.12), ale są dostępne także inne style. Ich wybór jest zależny od indywidualnych preferencji użytkownika.

Nie trzeba instalować dodatkowych wtyczek, można więc pominąć kolejne kroki i pozostawić domyślne ustawienia (w razie potrzeby wtyczki mogą być doinstalowane później).



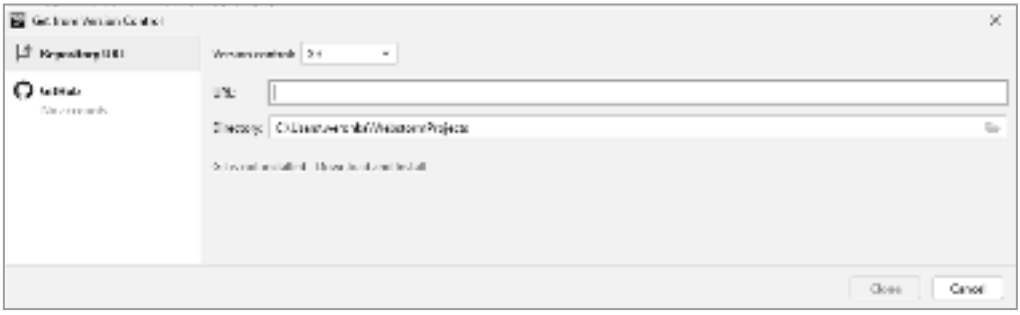
Rysunek 1.12. Wybór stylu dla programu WebStorm

Po zainstalowaniu oprogramowania pojawi się okno (rysunek 1.13) pozwalające wybrać, czy chcemy utworzyć nowy projekt (*Create New Project*), czy otworzyć i edytować już istniejący (*Open*). Opcja *Get from Version Control* pozwala na pobranie kodu z repozytorium zdalnego.

Rysunek 1.13. Okno, w którym wskazujemy źródło kodu

1.2.1.3. Integracja IDE z Gitem

Przy pierwszym uruchomieniu programu WebStorm i wybraniu opcji pobrania kodu z repozytorium uzyskamy widok podobny do pokazanego na rysunku 1.14:



Rysunek 1.14. Okno dialogowe ze wskazaniem łącza do repozytorium

Oczywiście istnieją inne repozytoria projektowe, ale na początku warto się skupić na najbardziej popularnym narzędziu.

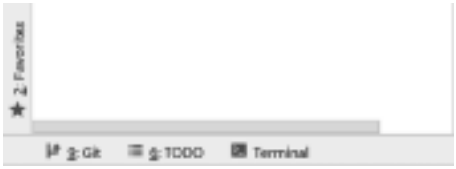
Jeśli do tej pory nie zainstalowaliśmy Gita, to wyświetlony zostanie link pozwalający na doinstalowanie brakujących aplikacji (*Download and Install* na rysunku 1.14). Po poprawnym zainstalowaniu pojawi się okno podobne do tego z rysunku 1.15:

Rysunek 1.15. Okno dialogowe pozwalające na pobranie pustego projektu z repozytorium zdalnego

Adres URL określa, w jakiej zdalnej lokalizacji mamy nasz projekt (czyli w polu URL trzeba wkleić link do projektu utworzonego na repozytorium zdalnym). Dynamicznie dodał się również katalog na podstawie wybranego projektu. Po kliknięciu przycisku *Clone* w tle będzie się wykonywać komenda:

```
git clone https://github.com/nazwa_uzytkownika/firstProject.git
```

Po otwarciu pustego projektu w WebStorm można zobaczyć w lewym dolnym rogu opcję wyboru repozytorium Gita (rysunek 1.16). Możemy ją wywołać za pomocą skrótu klawiszowego *Alt+9*.

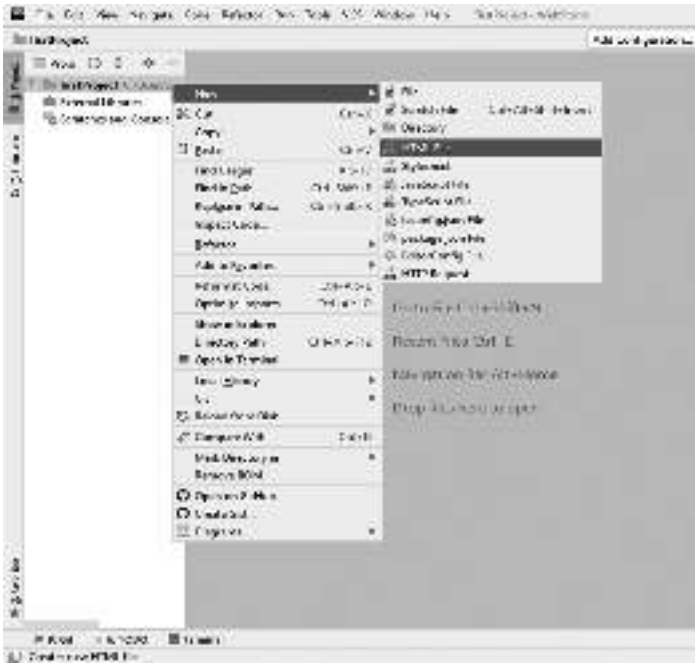


Rysunek 1.16. Lewy dolny róg z opcją wyboru okna dialogowego prowadzącego do repozytorium Gita

Kiedy uruchomimy to okno, zobaczymy, że obecnie nie mamy żadnych zmian (czyli nic nie stworzyliśmy — *No changes committed*), ale w lewej części okna widać, że jesteśmy na lokalnej gałęzi *master* (rysunek 1.17).

Rysunek 1.17. Widok okna Git w WebStorm

Teraz dodamy nowy plik, aby został umieszczony w repozytorium. W tym celu kliknij na nazwie projektu prawym przyciskiem myszy i wybierz opcje zgodnie z rysunkiem 1.18. Po uzupełnieniu nazwy pliku i wybraniu dowolnej wersji HTML pojawi się okno dialogowe (rysunek 1.19).



Rysunek 1.18. Widok dodania nowego pliku do istniejącego projektu

Jeśli się zgodzimy, to zobaczymy nasz plik w oknie lokalnych zmian (*Local Changes — Commit*).

Rysunek 1.19. Okno dialogowe z zapytaniem, czy nowo powstały plik dodać do repozytorium Gita

Jeśli domyślnie się nie otworzyło, mamy możliwość jego wyboru (zakładka *Commit* z lewej strony na rysunku 1.20; ten sam efekt uzyskamy, wciskając skrót *Alt+F1*).

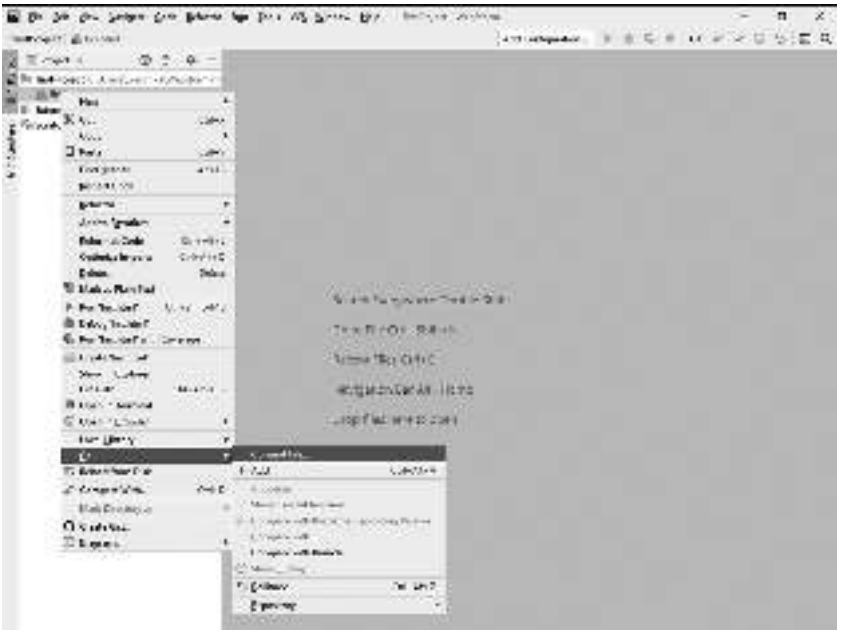
Rysunek 1.20. Wybór dostępnych widoków w WebStorm (zakładki z lewej strony)

Po wciśnięciu wspomnianej zakładki (lub skrótu) uzyskamy możliwość wyboru widoku (rysunek 1.21). Pod opcją z numerem 6 znajdziemy skrót prowadzący do lokalnych zmian (*Local Changes*).

Rysunek 1.21. Okno dialogowe wyboru dostępnych widoków

Po wybraniu tej opcji zobaczymy, że na liście lokalnych zmian nazwa dodanego pliku jest wypisana kolorem zielonym (gdybyśmy w oknie pokazanym na rysunku 1.19 wybrali przycisk *Cancel*, wyświetlałaby się na czerwono).

W oknie lokalnych zmian po ich wybraniu możemy z menu kontekstowego wybrać opcję *Commit File*, tym samym jawnie określając, że te zmiany chcemy przenieść do repozytorium zdalnego (rysunek 1.22).



Rysunek 1.22. Widok menu kontekstowego z oznaczonych wyborem Commit File

W oknie dialogowym, które się teraz ukaże, możemy jeszcze odznaczyć pliki wcześniej zaznaczone lub też wybrać pliki nieoznaczone, znajdujące się w katalogu *Unversioned Files*. Wymagane jest wprowadzenie opisu zatwierdzanych zmian (*Commit Message*).

UWAGA

Zmiany zatwierdzane w repozytorium zdalnym nazywane są często **commitem** (rysunek 1.23). Tak też będą określane w dalszej części tego rozdziału.

Rysunek 1.23. Okno dialogowe Commit

UWAGA

Im dokładniejsze są opisy zatwierdzanych zmian, tym łatwiej jest później nimi zarządzać.

W menu kontekstowym commita (rysunek 1.24) (po uzupełnieniu wymaganych pól — czyli wybraniu pliku/plików oraz uzupełnieniu pola z komentarzem) mamy do wyboru tylko przeniesienie zmian do tymczasowego katalogu (*Commit*). W rzeczywistości wszystkie pliki są oznaczone jako do przeniesienia.

Rysunek 1.24. Menu kontekstowe Commit

Kolejnym krokiem jest wysłanie zmian do repozytorium zdalnego. Za pierwszym razem pojawi się okno powiadamiające o tym, że nie mamy ustawionych niezbędnych informacji wskazujących, kto wykonał zmiany oraz jaki jest adres e-mail użytkownika (rysunek 1.25).

Rysunek 1.25. Okno dialogowe do uzupełnienia danych wymaganych przy komunikacji z repozytorium zdalnym

Jeśli wybraliśmy opcję z *Commit and Push*, to od razu zostaliśmy przekierowani do okna dialogowego z prośbą o podanie informacji niezbędnych do uwierzytelnienia w repozytorium zdalnym (rysunek 1.26).

Rysunek 1.26. Okno dialogowe z prośbą o dane dostępowe do wskazanego wcześniej repozytorium zdalnego

Jeśli poprawnie wpisaliśmy dane i operacja zakończyła się sukcesem, to w prawym dolnym rogu programu WebStorm pojawi się komunikat (rysunek 1.27):

Rysunek 1.27. Komunikat o poprawnym wysłaniu commita do repozytorium zdalnego

Jeśli nie wybraliśmy *Commit and Push*, to po wybraniu widoku *Git* (dostępnego również pod skrótem klawiaturowym *Alt+9*) i po wskazaniu gałęzi *master* w menu kontekstowym można wybrać *Push* i postępować zgodnie z wcześniej opisanymi krokami (rysunek 1.28).

Rysunek 1.28. Menu kontekstowe w widoku Git

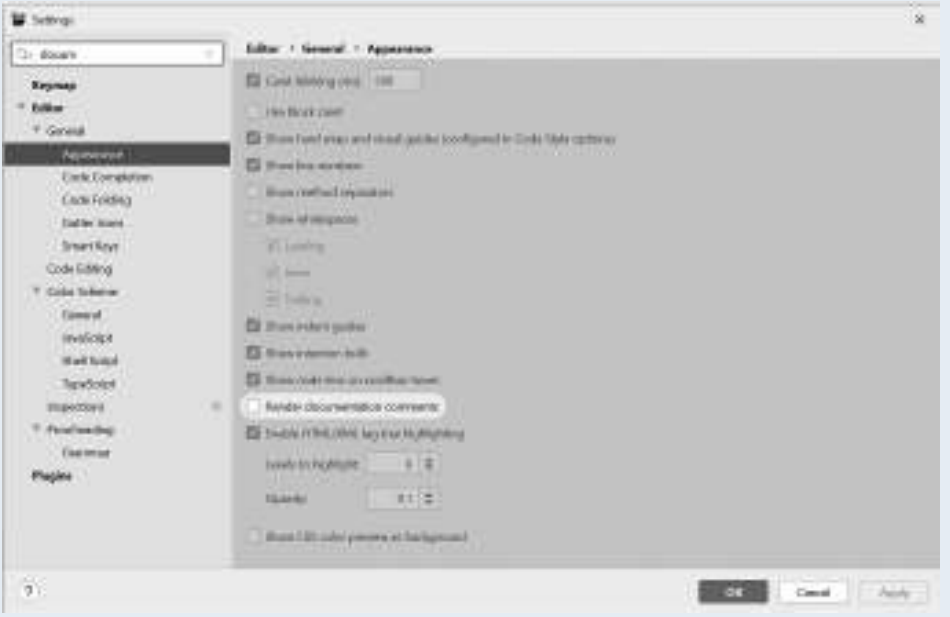
Po poprawnym przejściu całego procesu pod wskazanym linkiem projektu pojawi się wpis na stronie wybranego przez nas repozytorium zdalnego, wskazujący na to, że commit został zakończony sukcesem (rysunek 1.29).

Rysunek 1.29. Widok na stronie repozytorium zdalnego z pierwszym commitem

UWAGA

Podczas pisania aplikacji możemy nakazać środowisku WebStorm, by automatycznie generowało dokumentację.

Wywołamy tę opcję poprzez przejście do ustawień (*File/Settings*) albo poprzez skrót klawiszowy *Ctrl+Alt+S* (rysunek 1.30).



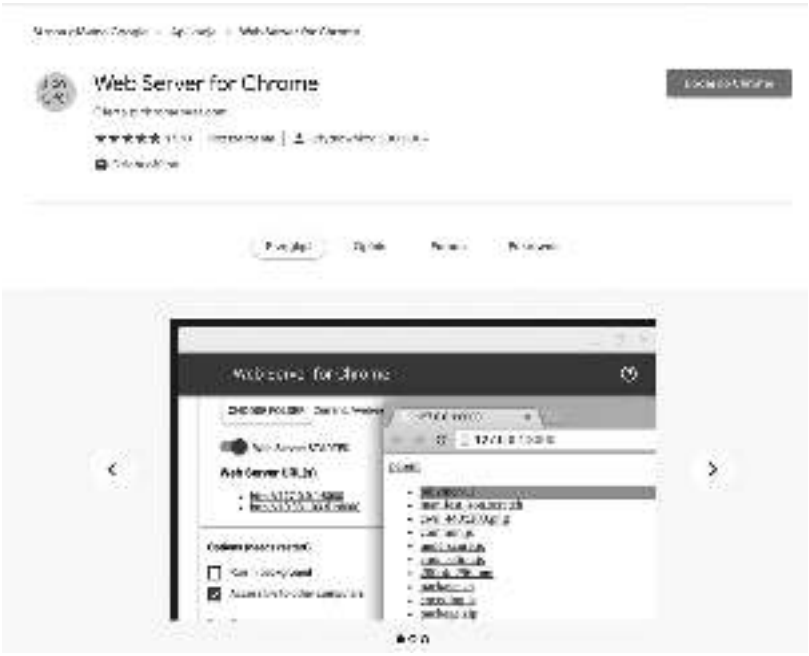
Rysunek 1.30. Wybierając wyróżnione pole, nakazujemy IDE generowanie dokumentacji

1.2.2. Serwer aplikacji

Oprócz plików HTML i CSS większość dostępnych stron internetowych ma również fragmenty pozwalające na interakcję z użytkownikiem, napisane w języku JavaScript. W celu poprawnego uruchomienia wszystkich plików składających się na aplikację należy je udostępnić na serwerze i wskazać adres URL, pod którym jest dostępny nasz program.

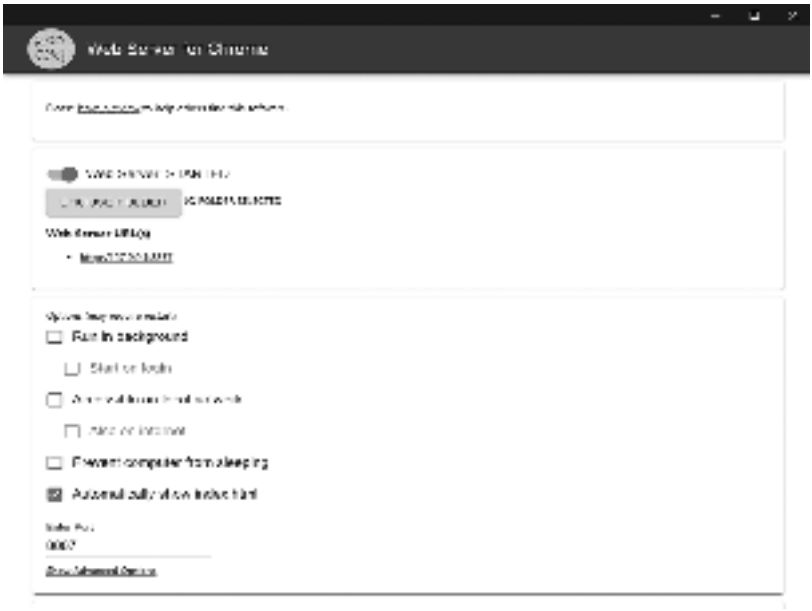
Przeglądarka Chrome ma wtyczkę, która jest właśnie serwerem.

Aby korzystać z tej funkcji, należy wyszukać i zainstalować wtyczkę *Web Server for Chrome* (rysunek 1.31).



Rysunek 1.31. Pobranie wtyczki Web Server for Chrome

W celu uruchomienia serwera należy wybrać w sekcji *Rozszerzenia* wtyczkę *Web Server for Chrome*; pojawi się wówczas okno dialogowe z ustawieniami wtyczki (rysunek 1.32):



Rysunek 1.32. Okno ustawień wtyczki Web Server for Chrome

W opcji *Choose folder* należy wskazać folder, gdzie jest zlokalizowana aplikacja (czyli folder nadrzędny dla pliku *index.html*).

Po kliknięciu URL ujrzymy uruchomioną aplikację.

1.3. Podsumowanie

Rozpoczynając pracę w zespole projektowym, na instalację narzędzi oraz zapoznanie się z nimi mamy od kilku godzin do kilku dni, w zależności od liczby narzędzi i ilości czasu, jaki został oszacowany przez przełożonego. Zapoznaj się dobrze z opisanymi tu programami, bo w dalszej części książki wszystkie będą wielokrotnie używane. Oprócz przedstawionych tutaj wybranych opcji warto przyswoić sobie dokumentację udostępnioną dla wyżej wymienionych aplikacji. Czas poświęcony na poznawanie narzędzi zwróci się w dwójnasób podczas korzystania z nich.

1.4. Zadania

Zadanie 1.1

Jaka jest różnica pomiędzy walidacją a weryfikacją?

Zadanie 1.2

Czy GitHub to jedyna platforma do repozytorium zdalnego? Jeśli uważasz, że nie, wymień inne.

2

Elementy programowania na przykładzie języka JavaScript

Zasadniczym etapem wytwarzania oprogramowania jest pisanie kodu aplikacji. W tym rozdziale omówimy elementy składające się na ten etap. Pokażemy, jak programować w języku JavaScript, choć w gruncie rzeczy moglibyśmy wybrać dowolny inny język programowania. Pokrótce omówimy podstawowe niezbędne umiejętności, także te niezwiązane bezpośrednio z JavaScriptem, jak pisanie kodu HTML czy stosowanie arkuszy CSS. Jeśli umiejętnie połączymy ze sobą wszystkie poznane tu elementy, będziemy w stanie stworzyć prostą aplikację webową.

Na początku JavaScript był prostym językiem skryptowym, mającym na celu dodanie prostych interaktywnych funkcjonalności do stron internetowych. Z czasem odkryto potęgę tego języka i obecnie jest używany w kodzie prawie każdej strony internetowej. Efekty, które można uzyskać, opierając się na JavaScripcie, to np. galerie zdjęć, popularne, choć nie zawsze lubiane „wyskakujące" okienka modalne czy możliwość wysyłania wiadomości e-mail z poziomu strony, a także niekiedy bardzo rozbudowane gry sieciowe.

Za wyborem JavaScriptu (w dalszej części tego rozdziału będziemy nazywać go skrótowo JS) przemawia także to, że wiele nowych bibliotek/frameworków oparto właśnie na tym języku. Innym atutem może być to, że pracując nad kodem, rezultat naszych prac możemy śledzić w dowolnie wybranej przeglądarce. W tym podręczniku przyjmiemy właśnie taką strategię, to znaczy będziemy używać prostego kodu JS bezpośrednio w przeglądarce. Trzeba jednak mieć świadomość, że kod JS może być wykorzystywany także w skomplikowanych i bardzo rozbudowanych projektach, np. webserwisach

(na Node.js, czyli rozbudowanym środowisku JS, umożliwiającym m.in. asynchroniczną — różne wątki są wykonywane równolegle — pracę na tych samych zasobach wielu użytkownikom, oparty jest choćby Netflix). Istnieją również desktopowe aplikacje oparte wyłącznie na JS, np. Slack (jeden z popularniejszych komunikatorów używanych w firmach).

Tę część podręcznika kierujemy do przyszłych programistów. Jeśli chcesz dołączyć do tego grona, musisz poznać pewne zasady pisania programów (niektóre z nich są stosowane nie tylko w JS).

Nauka programowania jest jak nauka języka obcego, np. hiszpańskiego. Na początku należy przyswoić charakterystyczne dla niego słownictwo oraz zasady gramatyczne. W przypadku języka programowania tę część nazywamy **składnią języka** (ang. *syntax*). W językach naturalnych istotna jest także interpunkcja. To, gdzie postawisz przecinek, kropkę lub średnik, ma niekiedy kluczowe znaczenie dla zrozumienia komunikatu. Jak zobaczysz, nie inaczej jest w przypadku języków programowania.

2.1. JS — i co dalej?

Prawie każda strona internetowa składa się z trzech składników: HTML, CSS i JS.

DEFINICJA

Kod HTML (ang. *HyperText Markup Language*) — zawiera treść, która zostaje wyświetlona na stronie. Zbudowany jest on z tagów, które mogą posiadać różne funkcje, np. są to nagłówki (ang. *header*), ciało strony (ang. *body*) i stopka (ang. *footer*).

CSS (ang. *Cascading Style Sheets*), czyli **kaskadowe arkusze stylów** — zadaniem tego narzędzia jest przypisanie stylów elementom HTML, definiujących np. wygląd czcionki, kolory, wygląd akapitów, rozmieszczenie elementów na stronie. Stosowanie CSS pozwala uniknąć powielenia kodu.

Silnik JavaScript (JS) — kod JS odpowiada za zachowanie dynamicznych elementów strony, umożliwia też interaktywność witryn. Kod ten jest interpretowany i wykonywany przez wbudowany w przeglądarkę silnik JavaScript lub inny przeznaczony do tego program — tzw. interpreter.

Aby lepiej wyjaśnić te różnice, uruchomimy przeglądarkę Chrome i wyświetlimy w niej stronę internetową. W każdej przeglądarce możemy dokładnie prześledzić, jak jest zbudowana strona. Po kliknięciu prawym przyciskiem myszki i wybraniu z menu kontekstowego pozycji *Zbadaj* (lub wciśnięciu skrótu klawiszowego *Ctrl+Shift+I*) ujrzymy po prawej stronie przeglądarki następujący widok (rysunek 2.1):

Rysunek 2.1. Konsola przeglądarki Chrome

W oknie konsoli można także wykonywać dowolny program napisany w języku JS. Program to zestaw komend, które uruchamiają się jedna po drugiej tylko wtedy, gdy wykona się poprzednia komenda.

Dzięki lekturze tego i kolejnych rozdziałów tej książki będziesz coraz lepiej rozumieć, jak uruchamiają się poszczególne komendy oraz cała aplikacja.

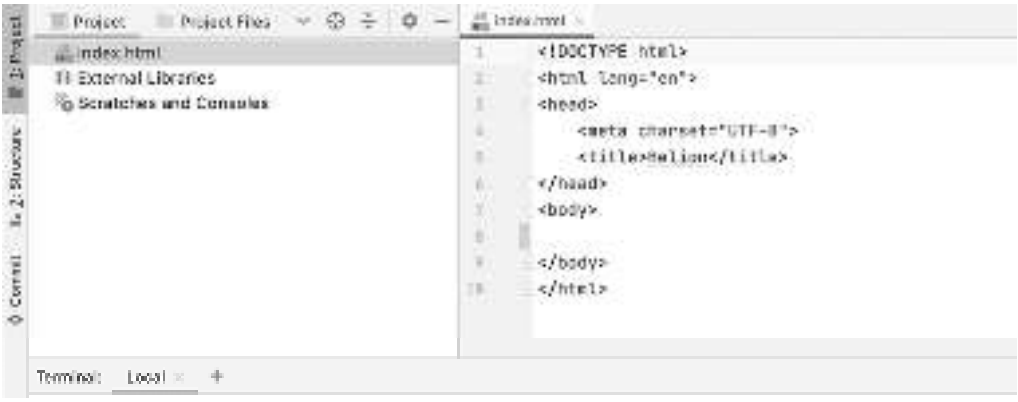
2.2. Składnia

2.2.1. Tagi

Elementem, który jest nam potrzebny do zrozumienia istoty języka JavaScript (a w gruncie rzeczy także HTML), jest **tag**. To pewien znaczący ciąg znaków rozpoczynający się otwierającym nawiasem ostrokątnym (<). Każdy tag musi być ponadto zamknięty. Służy do tego sekwencja znaków ukośnika i zamykającego nawiasu ostrokątnego (/>).

Tagi `<body></body>`, odpowiednio, otwierają i zamykają tzw. ciało strony internetowej. Tagi `<script></script>` wydzielają w obrębie strony skrypt (program) napisany w JS. Tagi mogą być zagnieżdżone. Jak pokazano na poniższym rysunku, tagi `<head></head>` oraz `<body></body>` zawierają się wewnątrz `<html></html>`.

Ponadto, jak widać na rysunku 2.2, w tagu *head* zawarliśmy również dodatkowe informacje o tytule strony i jej kodowaniu.



Rysunek 2.2. Kod domyślnie wygenerowanej strony index.html

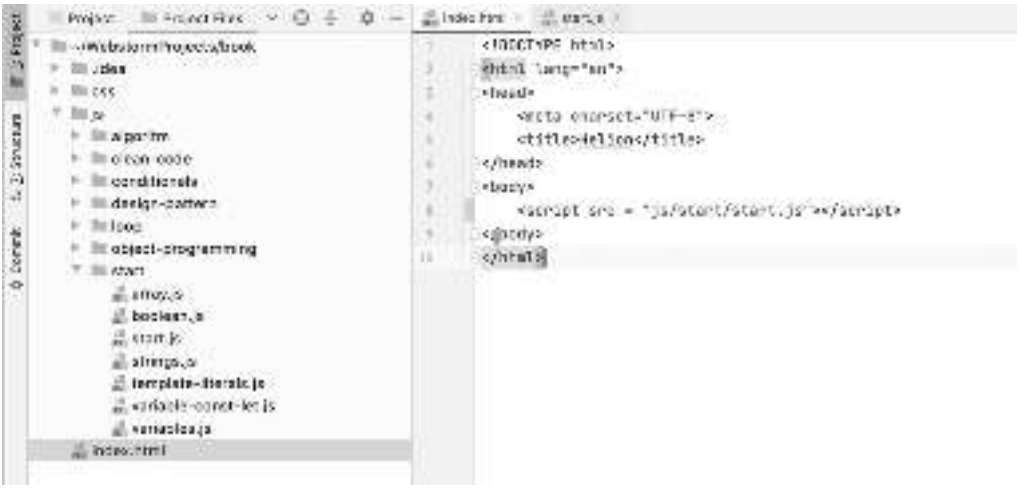
UWAGA

Nie musisz znać dokładnie właściwości wszystkich tagów. Jeśli chcesz się z nimi zapoznać, skorzystaj z ich zestawienia, które możesz znaleźć np. pod adresem *https://www.w3schools.com/tags/tag_html.asp*.

Twoja znajomość tagów będzie się pogłębiać z każdą napisaną przez Ciebie stroną. Im więcej ich napiszesz, tym łatwiejsze będzie kodowanie kolejnych.

2.2.2. Skrypt JS a plik HTML

Teraz do domyślnie wygenerowanej strony wstawimy nasz kod javascriptowy. Na razie zamieścimy pusty plik *start.js* (rysunek 2.3).

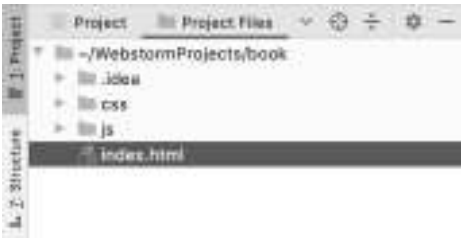


Rysunek 2.3. Skrypt start.js zamieszczony wewnątrz kodu strony index.html

UWAGA

Kody — zarówno HTML, jak i CSS, a przede wszystkim skryptów JS — znaleźć można w serwisie GitHub, pod adresem *https://github.com/weronikakortas/book*.

Aplikacje webowe, jak już wspomniano, składają się z trzech elementów. Każdy z nich dla większej przejrzystości kodu najlepiej jest oddzielać od pozostałych. W projekcie aplikacji będziemy mieli zatem trzy pliki (rysunek 2.4).



Rysunek 2.4. Struktura aplikacji webowej w widoku projektu w WebStorm

W tej chwili skupimy się wyłącznie na *start.js* i *index.html*, pomijając folder *css*, zawierający kaskadowe arkusze stylów.

To kod HTML odpowiada za to, co znajduje się na stronie. W pliku HTML można umieścić jej treść, np. nagłówki, akapity tekstu, listy wypunktowane, obrazki czy odsyłacze do innych stron. JavaScript odpowiedzialny jest za wykonywanie akcji, np. obliczeń czy złożonych interakcji z użytkownikiem. Aby poinstruować przeglądarkę internetową, że ma wykonać napisany przez nas kod JavaScript, w pliku HTML trzeba zawrzeć informację o tym, że na naszej stronie będziemy używać skryptów, które znajdują się w pliku *start.js*. Czynimy to, zamykając ścieżkę do pliku w znacznikach `<script></script>`, jak na rysunku 2.3.

UWAGA

Jak już wspomniano, plik *index.html* to domyślny nowy plik w formacie HTML, generowany za pomocą środowiska WebStorm. Zwróć jednak uwagę, że stanowi on prostą, ale kompletną i działającą aplikację. Wystarczy kliknąć w prawym górnym rogu przeglądarki, by od razu zobaczyć rezultat. Można też znaleźć plik na dysku i uruchomić go za pomocą Chrome (zgodnie z instrukcją zamieszczoną w pierwszym rozdziale).

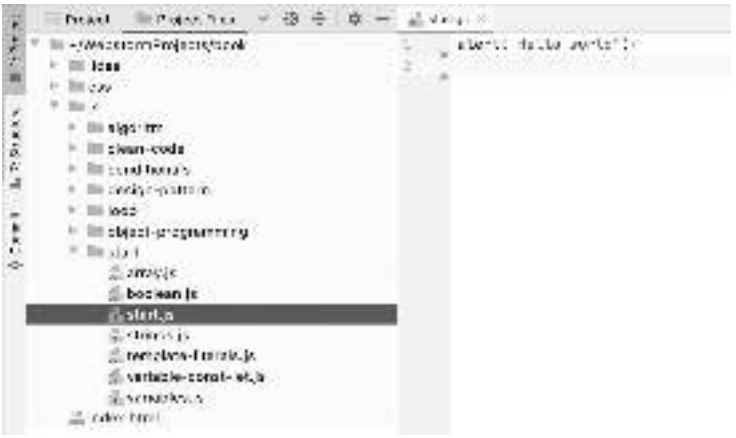
CIEKAWOSTKA

Współpraca między kodem HTML a JavaScriptem nie ogranicza się jedynie do poinstruowania przeglądarki internetowej znacznikiem `<script>` o konieczności wykonania skryptów ze wskazanych plików. Z poziomu skryptów JS możliwe jest odczytywanie treści strony internetowej, wpływanie na jej zawartość i wygląd, jak również reagowanie na działania użytkownika i zdarzenia — np. kliknięcie przycisku lub przewinięcie strony do określonego miejsca. Aby taka współpraca była możliwa, przeglądarki internetowe udostępniają dla JavaScriptu model obiektowy dokumentu (DOM, ang. *Document Object Model*) i model obiektowy przeglądarki (BOM, ang. *Browser Object Model*) — zespoły klas i interfejsów pozwalających na dostęp do zawartości strony internetowej i interakcję z przeglądarką i użytkownikiem z poziomu kodu skryptów.

W tym rozdziale poznasz podstawy składni języka JavaScript, jak również jego bardziej złożone elementy, takie jak obiekty i kolekcje, które ułatwią Ci zaznajomienie się z tymi interfejsami. Jeśli chcesz się z nimi zapoznać, zajrzyj np. pod adres *https://www.w3schools.com/js/js_htmldom.asp*, gdzie znajdziesz obszerne informacje na ten temat.

2.2.3. Wyświetlanie komunikatów

Rozbudujmy teraz nasz skrypt. Na początek niech ma w sobie wyłącznie komunikat informujący o tym, że użytkownik odwiedził naszą stronę. Do wyświetlania takich komunikatów służy polecenie `alert` (rysunek 2.5).

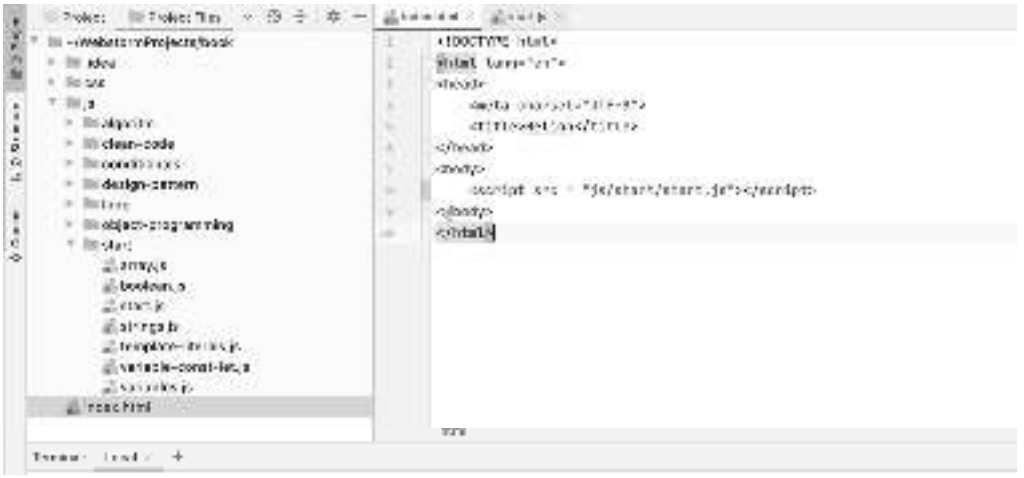


Rysunek 2.5. Instrukcja w skrypcie start.js wyświetlająca powitanie

Niemal każdy początkujący programista zaczyna od komunikatu „Hello world". Pójdźmy tym tropem i my.

Teraz czas dodać nasz alert do już wcześniej działającej aplikacji (rezultatem ma być wykonanie kodu strony i wyświetlenie zapisanego w skrypcie komunikatu).

Aby uzyskać ten efekt, należy nasz skrypt podlinkować do strony *index.html*. Pokazywaliśmy to już wcześniej (na rysunku 2.3); tym razem nasz skrypt nie jest jednak pusty, inna jest też jego lokalizacja (rysunek 2.6).



Rysunek 2.6. Plik index.html z „podpiętym" skryptem start.js

Do dodania skryptu znajdującego się w katalogu *js* należy użyć tagu *script* i dodać do niego właściwość informującą o tym, gdzie ten skrypt się znajduje. Musimy zatem użyć składni `<script src = "nazwa_katalogu/nazwa_skryptu.js"></script>`; w naszym przypadku użyliśmy jej w 8. linii domyślnego pliku HTML.

Po uruchomieniu pliku uzyskamy następujący efekt (rysunek 2.7):

Rysunek 2.7. Okno wyświetlone po wykonaniu skryptu start.js

Komunikat został wyświetlony w oknie alertu, który jest widoczny dla użytkownika odwiedzającego naszą stronę. A co, jeśli chcemy uzyskać informację widoczną także dla nas?

W takim przypadku należy dodać do skryptu wiersz powodujący wysłanie komunikatu do okna konsoli, używając polecenia pokazanego w drugiej linijce na listingu 2.1:

Listing 2.1

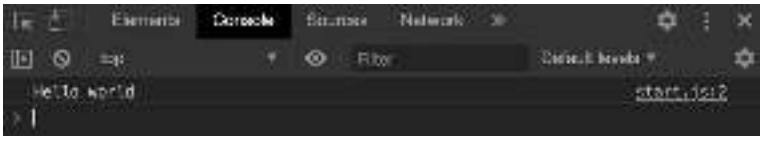
Przekierowanie komunikatu do okna konsoli

```
1. alert("Hello world");
2. console.log("Hello world"); // przekierowanie komunikatu do okna konsoli
```

Najpierw pojawi się alert w wyskakującym okienku — to skutek wykonania instrukcji `alert` z pierwszej linijki (rysunek 2.8).

Rysunek 2.8. Okno z komunikatem alertu

Następnie, po kliknięciu *OK*, ten sam komunikat zostanie wyświetlony w konsoli przeglądarki dzięki drugiej instrukcji skryptu (rysunek 2.9).



Rysunek 2.9. Ten sam komunikat wyświetlony w konsoli

2.2.4. Warto korzystać z IDE — mechanizmy autouzupełniania kodu i sygnalizacji błędów

Skoro już wiesz, jak uruchomić nasz przykładowy skrypt, dowiesz się teraz, jak wygląda pisanie poleceń w JavaScripcie: jakich słów należy użyć, kiedy i jakie stawiać punktory (średniki, kropki). Słowem: zapoznasz się z podstawami składni. Oczywiście w tym podręczniku skupimy się tylko na tych jej elementach, które są nam niezbędne. Nie wszystko będzie jasne od razu, więc zalecamy przeczytać poszczególne fragmenty kilkakrotnie, a przede wszystkim od razu pisać je w IDE, np. WebStorm, które można pobrać ze strony jetbrains.com lub z darmowych edytorów, takich jak Atom lub Visual Studio Code. Warto zapisywać sobie elementy składni na kartce w celu jej zapamiętania, ale korzystanie z IDE ma tę zaletę, że udostępnia ono mechanizm uzupełniania składni, a ponadto poprawia (lub przynajmniej sygnalizuje) większość błędów.

Przykład 2.1

Zacznijmy teraz pisać wywołanie `alert` (rysunek 2.10).

Rysunek 2.10. Mechanizm autouzupełniania poleceń

Po wpisaniu kilku liter pojawiła się podpowiedź sugerująca pełną nazwę funkcji. Jak się można domyślić, `alert` przyjmuje i wyświetla dowolny napis. Napis w JavaScripcie można umieścić w cudzysłowie, przy czym trzeba pamiętać, aby podać zarówno cudzysłów otwierający, jak i zamykający (rysunek 2.11).

Rysunek 2.11. Mechanizm poprawiania składni wskazuje błędy we wpisywanych poleceniach

Jak widać, nasze IDE już podpowiada nam, że coś jest nie tak. My możemy jednak tego nie zauważyć i mimo to uruchomić nasz skrypt. Próba wykonania kodu skończy się wówczas błędem (rysunek 2.12).

Rysunek 2.12. Wyświetlony w konsoli komunikat o błędzie

Akurat w tym przypadku zabrakło cudzysłowu zamykającego.

UWAGA

Liczba cudzysłowów w skrypcie musi być parzysta — każdy cudzysłów musi mieć zakończenie.

Przykład 2.2

Spróbujmy teraz uruchomić skrypt napisany w taki oto sposób (rysunek 2.13):

```
1    alert("Hello world"
```

Rysunek 2.13. Mechanizm poprawiania składni wskazuje błędy we wpisywanych poleceniach (cd.)

Tym razem wpisaliśmy cudzysłów zamykający. Jednak, jak widać, IDE podkreśliło miejsce za ostatnim wpisanym znakiem — właśnie cudzysłowem. To sugeruje, że i tym razem popełniliśmy błąd. Aby się o tym przekonać, znów uruchomimy skrypt i zobaczymy, co pojawi się w konsoli (rysunek 2.14).

Rysunek 2.14. Komunikat o błędzie informuje, że na końcu polecenia zabrakło nawiasu zamykającego

UWAGA

Każdy nawias (czy to kwadratowy, czy półokrągły, czy klamrowy) musi być domknięty. Podobnie jak w przypadku cudzysłowów, liczba nawiasów musi być zatem parzysta. Ponadto nawiasy muszą być zamykane w kolejności odwrotnej do tej, w której były otwierane: `{([])}`.

Przykład 2.3

Spróbujmy zapisać słowo `alert`, zaczynając od wielkiej litery (rysunek 2.15).

```
1    Alert("Hello world");
```

Rysunek 2.15. Wielkość liter w JS ma znaczenie

Uzupełniliśmy do pary nawiasy i cudzysłowy, nic nie zostało podkreślone na czerwono. Czy zatem wykonamy kod bez błędu? Zobaczmy (rysunek 2.16).

Rysunek 2.16. Interpreter nie rozpoznaje słowa „Alert" jako nazwy metody

Tym razem wyświetlony został komunikat z informacją „Alert is not defined". Oznacza to, że dane słowo nie występuje wśród dostępnych możliwych wywołań, w szczególności nie zostało rozpoznane jako polecenie `alert`.

UWAGA

Wielkość liter w JS ma znaczenie.

Przykład 2.4

Spróbujmy teraz wypisać „Hello world" i w konsoli, i w oknie alertu. Użyjmy poleceń z rysunku 2.17 (jeśli widzisz w nich błędy, to dobrze; na razie jednak nie przejmujmy się nimi):

```
1  Alert("Hello world");
2  console.log("Hello world"
```

Rysunek 2.17. Jeśli w skrypcie popełnimy dwa błędy…

Po uruchomieniu tak napisanego skryptu uzyskamy w konsoli komunikat pokazany na rysunku 2.18.

Rysunek 2.18. …po jego uruchomieniu zostanie zgłoszony jeden z nich

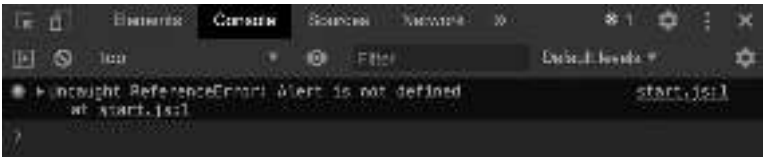
Informuje on nas o tym, że brakuje nawiasu w drugiej linijce (zgodnie z informacją zawartą na końcu linijki — *start.js:2*).

Poprawmy ten błąd i odświeżmy stronę (rysunek 2.19).

```
1  Alert("Hello world");
2  console.log("Hello world");
```

Rysunek 2.19. Po poprawieniu zgłoszonego błędu…

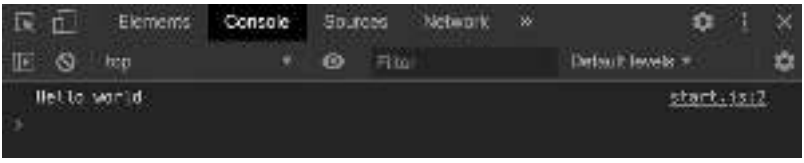
Tym razem w konsoli otrzymaliśmy kolejną informację o błędzie (rysunek 2.20).



Rysunek 2.20. ...zgłoszony zostanie kolejny błąd

Ten błąd także widzieliśmy już wcześniej, wiemy więc, na czym polega. Wpisaliśmy `Alert`, a poprawnym poleceniem jest `alert`.

Po poprawieniu tego błędu wreszcie pojawi się okno alertu, a w konsoli otrzymamy (rysunek 2.21):



Rysunek 2.21. Poprawny wynik skryptu wyświetlony w konsoli

Wynika z tego, że **błędy w konsoli pojawiają się kolejno, zgodnie z występowaniem w kodzie, a zatem nawet po usunięciu jednego błędu może pojawić się drugi**.

Widzieliśmy ponadto, że komendy wykonują się po kolei. W konsoli napis pojawił się dopiero po kliknięciu *OK* w alercie z przeglądarki.

Co więcej, można było zauważyć, że w JS, inaczej niż w wielu innych językach programowania, średnik nie kończy polecenia. Możemy go postawić na końcu instrukcji, ale nie musimy. To już zależy od tego, w jaki sposób łatwiej będzie zrozumieć, gdzie kończy się polecenie. Jeśli wyraźnie widać, że jedno polecenie przypada na jedną linijkę, to możemy nie stosować średnika na końcu, jednak nie powinno się łączyć dwóch podejść i jeśli stosujemy zapis bez średników, to należy go stosować wszędzie.

2.2.5. Komentarze

W celach edukacyjnych będziemy stosowali komentarze, by opisać, co w którym miejscu kodu się dzieje. Ich stosowanie jest dobrą praktyką programistyczną. Ułatwiają orientację w kodzie — zarówno własnym, kiedy wracamy do niego po dłuższej przerwie, jak i (tym bardziej) cudzym. Oczywiście nie ma sensu wyjaśniać w komentarzach każdej linii kodu, nie powinniśmy też tego robić w przypadku tych fragmentów, których przeznaczenie jest dla osób znających składnię języka oczywiste. Trzeba ponadto pamiętać, że w praktyce kod powinien być pisany tak, by komentarze były zbędne. Nazwy zmiennych i metod powinny być na tyle jasne, by osoba czytająca kod nie potrzebowała dodatkowych wyjaśnień.

Wyróżniamy dwa rodzaje komentarzy:

- komentarze jednolinijkowe, które rozpoczynamy dwoma ukośnikami (`//`); pomiędzy tymi znakami nie może występować spacja ani żaden inny znak (listing 2.2);

Listing 2.2

Komentarz jednolinijkowy

```
// Komentarz jednolinijkowy
```

- komentarze zajmujące wiele linii, które umieszczamy pomiędzy znakami `/*` a `*/` (listing 2.3).

Listing 2.3

Komentarz składający się z wielu linii

```
/* Komentarz
wielolinijkowy
*/
```

Jeśli poprawnie udało nam się napisać ten element kodu, to jest on wyszarzony.

Warto być świadomym, że nic spośród tego, co jest komentarzem, nie ma wpływu na działanie programu — jest to podczas jego wykonywania pomijane. Z tego względu dzięki komentarzom możemy w łatwy sposób „wyłączać" fragmenty kodu, których wykonywanie chcielibyśmy chwilowo zatrzymać (np. niedokończone lub potencjalnie błędne).

2.3. Podstawy programowania

Aby móc stworzyć pełną aplikację, trzeba najpierw nauczyć się podstaw programowania. Pewne podstawowe informacje związane ze składnią języka już zaprezentowaliśmy. Ten rozdział stanowi wprowadzenie w tajniki programowania; zasady tu poznane wykorzystamy w ostatnim rozdziale.

Nie jest to pełne kompendium wiedzy dotyczącej programowania w języku JavaScript. Jest to zaledwie niezbędne minimum, pozwalające na stworzenie prostej aplikacji webowej.

2.3.1. Zmienne (var)

UWAGA

W tym i kilku innych punktach rozdziału skupiamy się na opisaniu deklarowania zmiennych i tworzenia obiektów w języku JavaScript. Należy tutaj podkreślić, że w innych językach (choć nie we wszystkich) deklarowanie odbywa się poprzez jawne oznaczenie, jakiego typu zmienną chcemy utworzyć. Język dynamicznie typowany, taki jak JavaScript, nie wymusza na programiście deklaracji typu zmiennej, lecz pozostawia interpretację zmiennej interpreterowi na podstawie jej zawartości.

→

UWAGA cd.

JavaScript jest ponadto językiem interpretowanym, co oznacza, że specjalny program — interpreter — analizuje napisany kod (czyli kod źródłowy) i następnie przekłada go na kod maszynowy, czyli informacje przetwarzane przez komputer.

Niektóre języki (języki kompilowane) zamiast interpretera stosują kompilator, który różni się od interpretera tym, że od razu tłumaczy cały kod programu. W przypadku interpretera natomiast program tłumaczony jest na bieżąco od momentu jego uruchomienia, instrukcja po instrukcji.

Językami interpretowanymi są np. JavaScript albo Python. Językiem kompilowanym jest np. C++.

W przypadku programowania zmienne mają za zadanie zbieranie, przechowywanie i przekazywanie dalej (np. do metod) informacji potrzebnych do wykonania programu.

Przykład 2.5

Wyobraź sobie aplikację bankową. Wiadomo, że musi być w niej przechowywany, wyświetlany i bardzo często modyfikowany stan konta. Nie będzie on zatem wartością stałą, a zarazem musi być zawsze dostępny. W świecie programowania do przechowywania stanu konta (i innych wartości mogących się zmieniać) doskonale nadaje się zmienna, która jest dynamicznie modyfikowana w zależności od tego, czy coś zostało wpłacone na konto, czy też wykorzystano środki z konta na zakupy. Przy każdej operacji bankowej zawartość zmiennej będzie inna. Informacja o modyfikacji, podobnie jak bieżący stan konta, nie musi być każdorazowo wyświetlana użytkownikowi; ważne, by każda zmiana była odzwierciedlana w zawartości zmiennej, dostępnej w kodzie aplikacji.

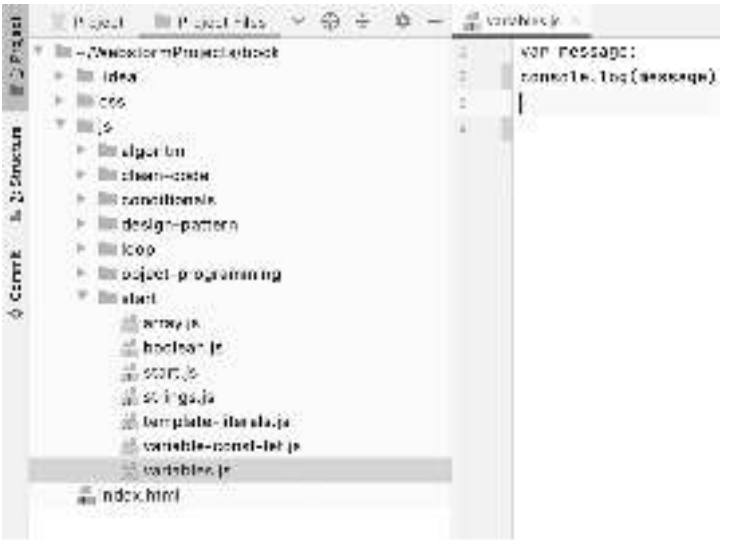
UWAGA

Im mniejsza liczba zmiennych, tym czytelniejszy jest nasz kod. Przed utworzeniem kolejnej zmiennej musimy się zastanowić, czy nie istnieje już gdzieś jakaś inna, która przechowuje potrzebne nam wartości.

Trzeba też pamiętać o tym, żeby jednoznacznie identyfikować zmienne. Jeśli wyobrazimy sobie zmienną jako adresata listu, a program jako skrzynkę na korespondencję, łatwo zdać sobie sprawę z tego, że omyłkowe wysłanie listu do niewłaściwego adresata nie byłoby najlepszym rozwiązaniem.

Przykład 2.6

Utwórzmy pierwszą zmienną. W naszym kodzie będzie ona służyła do przechowywania komunikatu do wyświetlenia (rysunek 2.22).



Rysunek 2.22. Utworzenie (deklaracja) zmiennej

W celu utworzenia zmiennej trzeba podać słowo kluczowe `var`, które jest skrótem od angielskiego *variable*, oznaczającego zmienną. Następnie podajemy nazwę naszej zmiennej — w tym przypadku `message`.

UWAGA

Warto pamiętać, że nazwy zmiennych powinny być takie, by jak najlepiej informowały o przechowywanych w nich wartościach — powinny być znaczące. Nazywanie zmiennych `x` (może z wyjątkiem niektórych działań matematycznych) czy `mojaZmienna` nie jest zatem dobrym pomysłem. Dobrym zwyczajem jest nadawanie zmiennym nazw w języku angielskim, ale nie jest to wymagane. Warto natomiast spójnie trzymać się raz obranego sposobu nazywania zmiennych.

WSKAZÓWKA

Warto zauważyć, że zmieniliśmy skrypt, który wykonujemy. Widać to po lewej stronie rysunku 2.22. Tym razem jest to *variables.js*. Musimy więc wpisać go do kodu strony *index.html* (listing 2.4).

Listing 2.4

Zmieniamy w pliku index.html informację o skrypcie, który ma być wykonywany

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Helion</title>
</head>
<body>
<script src="js/start/variables.js"></script>
</body>
</html>
```

Obecnie nasza zmienna nie ma żadnej wartości. Czy to oznacza, że kiedy wypiszemy ją w konsoli, będzie pusta?

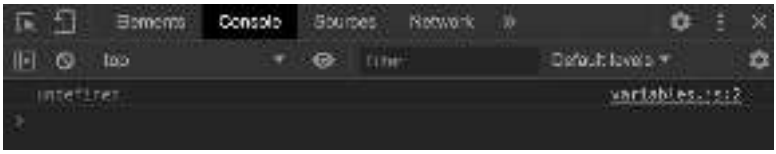
Spróbujmy to zrobić (listing 2.5).

Listing 2.5

Próba odwołania się do zmiennej, której nie przypisano żadnej wartości…

```
var message;
console.log("zawartość message " + message);
```

Efekt widać na rysunku 2.23. Uzyskaliśmy błąd *undefined*, który w tym przypadku oznacza, że do zmiennej nie została przypisana żadna wartość.



Rysunek 2.23. …skutkuje wyświetleniem informacji, że nie zdefiniowano zawartości zmiennej

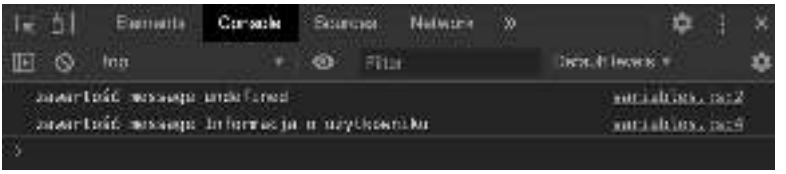
Teraz przypiszmy do naszej zmiennej wartość. Robimy to w 3. linii kodu z listingu 2.6. Przypisanie wartości do zmiennej uzyskuje się za pomocą operatora =, który wpisujemy pomiędzy nazwą zmiennej a wartością, jaką chcemy jej nadać. Pamiętaj, że aby przypisać do zmiennej ciąg znaków, trzeba ująć go w znaki cudzysłowu.

Listing 2.6

Przypisujemy do zmiennej wartość

```
1 var message;
2 console.log("zawartość message " + message);
3 message = "Informacja o użytkowniku";
4 console.log("zawartość message " + message);
```

Po wykonaniu skryptu uzyskamy taki rezultat (rysunek 2.24):



Rysunek 2.24. Wyświetlona w konsoli zawartość zmiennej message

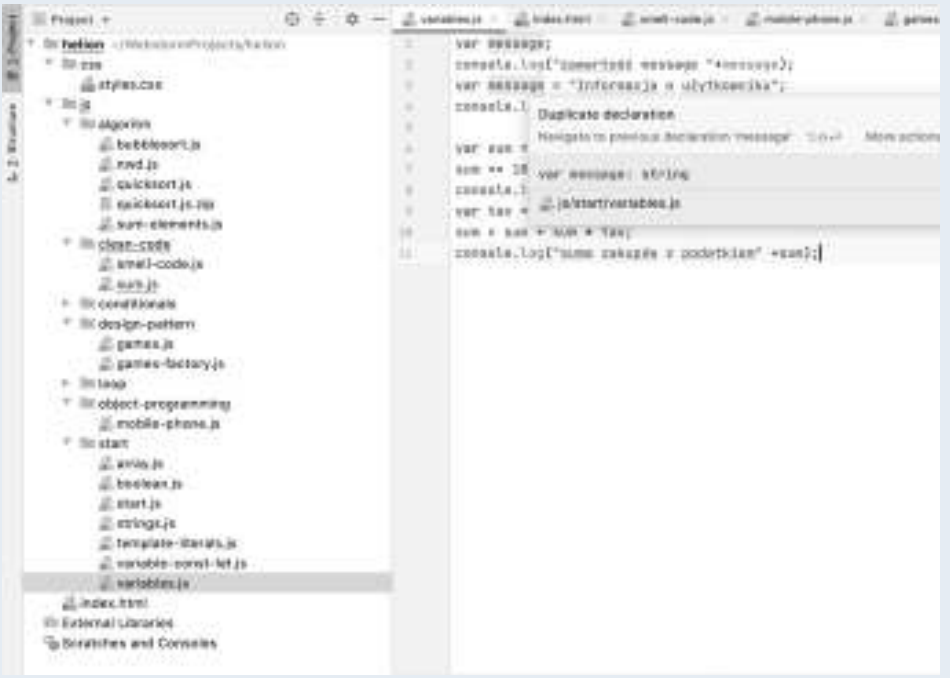
Tak oto przypisaliśmy poprawnie wartość do naszej zmiennej.

Od tej pory (o ile nie zmienimy wartości zmiennej `message`) możemy mówić, że `message` jest referencją (odwołaniem/wskazaniem) do wartości „Informacja o użytkowniku".

UWAGA

Przy zmianie wartości zmiennej nie używa się już słowa kluczowego `var`, które służy wyłącznie do tworzenia (deklaracji) zmiennej.

Gdybyśmy je zastosowali, utworzylibyśmy nową zmienną, a WebStorm (IDE) podpowiedziałby nam, że taka zmienna już istnieje (rysunek 2.25).



Rysunek 2.25. Ostrzeżenie przed zdublowaniem zmiennej

Przypisanie wartości do zmiennej w C++ wymagałoby jawnego wskazania (i to wcześniejszego, już na etapie deklarowania zmiennej), jakiego typu chcemy użyć. W JS, który jest językiem dynamicznie typowanym, nie jest to konieczne.

W przypadku napisu w C++ należałoby użyć tablicy znaków typu `char`, czyli przypisanie do zmiennej `message` napisu wyglądałoby tak:

```
char message[] = "Informacja o użytkowniku";
```

Alternatywnie (i prościej) można by użyć typu `string`, ale wymagałoby to importowania zewnętrznej biblioteki (o czym jeszcze napiszemy później).

UWAGA

Trzeba być także świadomym, że utworzenie zmiennej o takiej samej nazwie, jaką ma już inna zmienna, skutkuje *przesłonięciem* tej pierwszej. Oznacza to, że od tej pory widoczna będzie w kodzie tylko ta druga zmienna. Jakakolwiek próba odwołania się do zmiennej poprzez jej nazwę — np. w celu wyświetlenia za pomocą polecenia `alert(message)` — będzie skutkowała działaniem na tej wartości, która została przypisana do później utworzonej zmiennej.

UWAGA

Nie wszystkie nazwy zmiennych są dozwolone. Nie można np. używać do nazywania zmiennych słów kluczowych języka JavaScript, które zestawiono w tabeli 2.1.

Tabela 2.1. Lista zarezerwowanych słów kluczowych

abstract	boolean	break	byte	case
catch	char	class	const	continue
debugger	default	delete	do	double
else	enum	export	extends	false
final	finally	float	for	function
goto	if	implements	import	in
instanceof	int	interface	long	native
new	null	package	private	protected
public	return	short	static	super
switch	synchronized	this	throw	throws
transient	true	try	typeof	var
void	volatile	while	with	

2.3.2. Zmiana wartości zmiennych

Czasem w trakcie działania aplikacji chcemy zmienić wartość już raz zdeklarowanej zmiennej.

W kodzie zaprezentowanym na listingu 2.7 mamy np. zmienną `sum`. Początkowo przypisaliśmy jej wartość 3. W linijce 2. chcemy jednak zmienić wcześniej zadeklarowaną wartość, dodając do niej 10.

UWAGA

Operatory matematyczne działają w JS tak samo jak w matematyce.

Listing 2.7

Modyfikowanie wartości zmiennej sum poprzez dodanie 10 do jej początkowej wartości

```
1 var sum = 3;
2 sum += 10;
3 console.log(sum);
```

UWAGA

W językach C++ lub Java zadeklarowanie zmiennej typu liczbowego i jednoczesne przypisanie do niej wartości wyglądałoby tak:

```
int sum = 3;
```

a powiększenie tak:

```
sum += 3;
```

UWAGA

Dostępne w JS operatory matematyczne to: dodawanie (+), odejmowanie (–), mnożenie (*), dzielenie (/) i modulo, czyli reszta z dzielenia (%). Ich priorytet (kolejność wykonywania) jest taki sam jak w matematyce.

WSKAZÓWKA

W praktyce konstrukcję typu:

```
sum = sum + 10;
```

zazwyczaj zastępuje się następującym równoważnym (a krótszym) zapisem:

```
sum += 10;
```

WSKAZÓWKA cd.

Oczywiście w taki skrócony sposób można zapisać działanie z użyciem dowolnego operatora matematycznego (`+=`, `-=`, `*=`, `/=`, `%=`).

Przykład 2.7

Załóżmy, że utworzona wcześniej zmienna `sum` przechowuje kwotę do zapłaty za zakupione produkty. Zwiększa się ona o odpowiednią kwotę po każdym zakupie. W tym przykładzie dodajemy do sumy zakupów jeszcze podatek, który jest naliczany do każdego kupionego produktu.

Podatek zadeklarujemy na poziomie 30%. Możemy go zapisać w kodzie jako ułamek 0,3 — linijka 1. na listingu 2.8. Zwróć jednak uwagę na to, że części dziesiętne ułamków zapisuje się w programowaniu po kropce, a nie po przecinku.

Listing 2.8

Wyliczenie i wyświetlenie kwoty do zapłaty powiększonej o podatek

```
1 var tax = 0.3;
2 sum += sum * tax;
3 console.log("suma zakupów z podatkiem " + sum);
```

Ostatecznie sumą będzie zaś dotychczasowa suma zakupów plus podatek, naliczany od już obliczonej kwoty. Podatek to inaczej suma zakupów pomnożona przez wskazaną wartość podatku, czyli `sum*tax`. Po obliczeniach wypisujemy wartość powiększoną o kwotę podatku w konsoli.

Zauważ, że + ma w programowaniu jeszcze jedną funkcję. W wierszu 3. nie wykonujemy już żadnych obliczeń, ale za pomocą plusa informujemy program, że po komunikacie ujętym w cudzysłowy ma wyświetlić jeszcze aktualną wartość zmiennej `sum`.

UWAGA

Do wyliczenia podatku użyliśmy liczby 0,3, czyli tzw. liczby zmiennoprzecinkowej. W JS zmienne do przechowywania takich liczb deklaruje się tak jak pozostałe zmienne, czyli bez określania typu.

W przypadku języka C++ typów liczb zmiennoprzecinkowych jest więcej. Może to generować problemy podczas wyliczeń, gdyż `float` zaokrągla dane z 7, 8 cyfr po przecinku, a `double` z 15 cyfr. Jeśli w naszym programie porównalibyśmy liczby zmiennoprzecinkowe o takich samych wartościach, ale różnych typach, nie moglibyśmy oczekiwać, że we wszystkich przypadkach wynik będzie taki sam, gdyż w rzeczywistości liczby nie byłyby takie same.

W każdym języku, w jakim tworzymy, warto na początku dokładnie zapoznać się z dostępnymi typami zmiennych.

2.3.3. Definiowanie zmiennych (let i const)

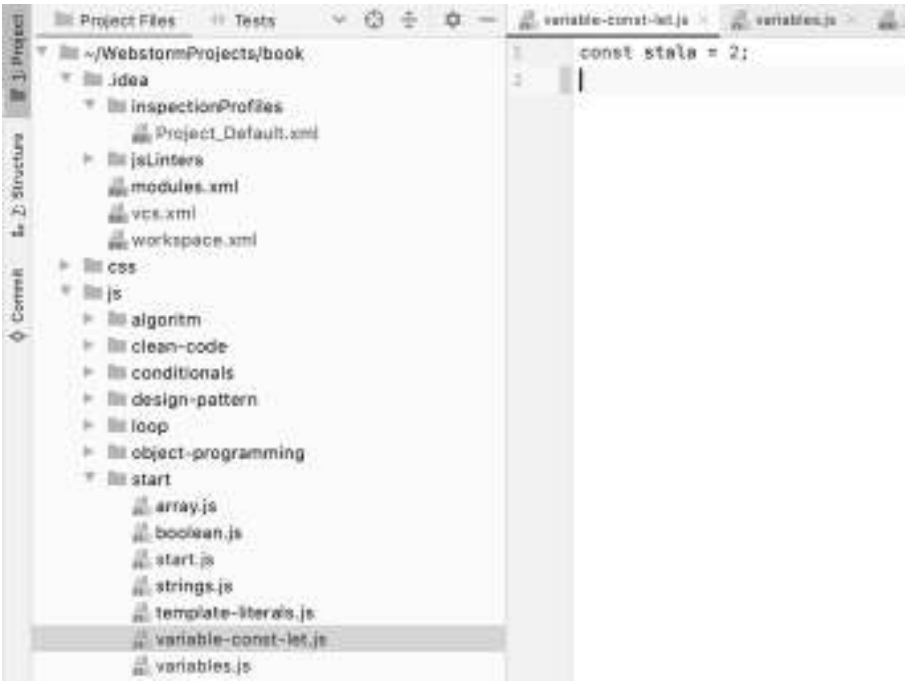
Jeśli definiujemy zmienne za pomocą słowa kluczowego `var`, czasem może wystąpić problem nadpisania zmiennej w trakcie działania programu przez inną zmienną. Dlatego w nowszych wersjach JS używane są słowa kluczowe `let` i `const`.

Omówmy najistotniejsze różnice między nimi.

Nazwa `const` jest skrótem od słowa *constant*, oznaczającego niezmienność/stałość; w praktyce oznacza to, że raz zdefiniowanej za pomocą `const` zmiennej nie można zmienić. Nazywamy ją wtedy stałą.

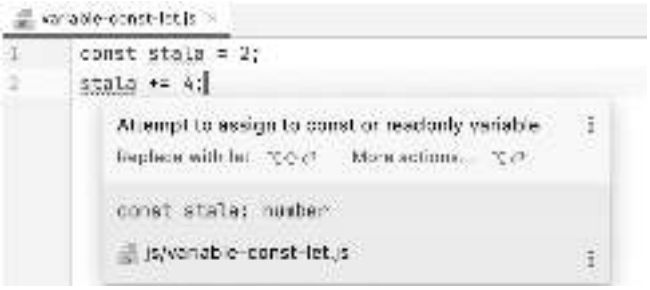
Przykład 2.8

Dodaliśmy do naszego projektu nowy plik: *variable-const-let.js*. W związku z tym w pliku *index.html* zmieniliśmy skrypt, do którego się odwołujemy. Na rysunku poniżej widzimy zawartość tego skryptu. Utworzyliśmy w nim (za pomocą słowa kluczowego `const`) zmienną `stala` (dla nazw zmiennych lepiej jest nie używać polskich znaków) i przypisaliśmy jej wartość 2 (rysunek 2.26). Zauważ, że można zadeklarować zmienną i przypisać do niej wartość w jednej linijce; dotychczas robiliśmy to w dwóch osobnych poleceniach.



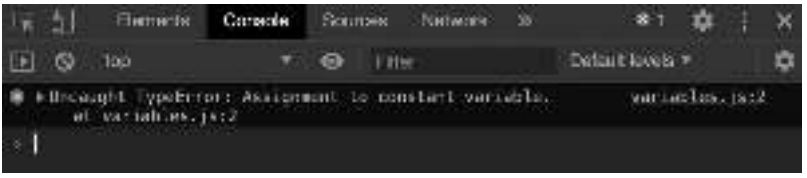
Rysunek 2.26. Tworzenie stałej za pomocą słowa kluczowego const

Spróbujmy teraz powiększyć wartość tak utworzonej (za pomocą słowa kluczowego `const`) zmiennej. Użyjemy do tego prezentowanej już wcześniej konstrukcji += (rysunek 2.27).



Rysunek 2.27. Wartości zmiennej utworzonej za pomocą słowa const nie da się zmienić

WebStorm podpowiada nam, że następuje próba przypisania wartości do zmiennej typu `const`, i podkreśla naszą zmienną; możemy się zatem spodziewać, że program zostanie uruchomiony, ale w konsoli zobaczymy błąd. Sprawdźmy to (rysunek 2.28).



Rysunek 2.28. Wyświetlenie błędu polegającego na próbie zmiany wartości stałej

UWAGA

Jak widać, zmienna zdeklarowana przez `const` powoduje, że raz przypisanej do niej wartości nie można zmienić. Z tego względu do takich zmiennych musimy przypisywać wartości już na etapie ich tworzenia.

Deklarowanie poprzez słowo kluczowe `let` jest analogiczne do użycia `var`. Różnica pomiędzy nimi jest widoczna dopiero wtedy, kiedy utworzymy w kodzie kolejną zmienną, która ma inne zadanie niż ta zadeklarowana wcześniej, ale przypadkowo obie nazwiemy tak samo.

Przykład 2.9

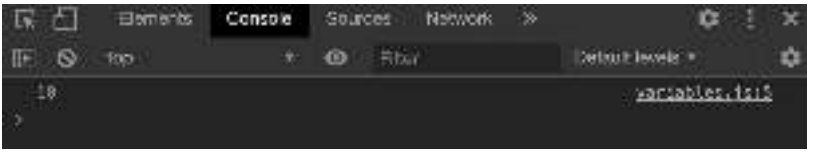
Oto co się dzieje, gdy taka zmienna jest zadeklarowana za pomocą `var` (listing 2.9):

Listing 2.9

Deklaracja dwóch zmiennych o takiej samej nazwie za pomocą słowa kluczowego var

```
1 var sum = 4;
2 let sumInitByLet = 10;
3 // inne fragmenty kodu
4 var sum = 10;
5 console.log(sum);
```

W linii 1. zadeklarowaliśmy zmienną `sum` i przypisaliśmy jej wartość 4. Później, w wierszu 4. (w zwykłych okolicznościach pewnie byłoby to dużo dalej w kodzie, w przypadku zmiennych deklarowanych tak blisko siebie zapewne zauważylibyśmy, że nazywają się one tak samo), utworzyliśmy inną zmienną, znów za pomocą słowa kluczowego `var`, i przypisaliśmy jej wartość 10. Co się stanie, kiedy wykonamy skrypt? W konsoli otrzymamy taki oto wynik (rysunek 2.29):



Rysunek 2.29. Zmienna utworzona jako pierwsza została przesłonięta przez drugą

Jak widać, później utworzona zmienna *przesłoniła* tę utworzoną wcześniej (pisaliśmy już o tym), w związku z czym wyświetlona została wartość właśnie tej drugiej zmiennej. Program „nie widzi" obecnie wartości `sum` z wiersza 1.

UWAGA

Warto zwrócić uwagę na to, że składnia JS czy Pythona nie wymaga kończenia poleceń średnikiem. W innych językach, jak C++ czy Java, średnik ma kluczowe znaczenie i jego niezastosowanie spowoduje błąd kompilacji. W JS używanie średników jest uznawane za dobrą praktykę, choć nie jest wymagane.

Przykład 2.10

Inaczej będzie w przypadku zmiennej zadeklarowanej za pomocą konstrukcji `let`. Postępujemy podobnie jak w poprzednim przykładzie. Najpierw tworzymy zmienną `sumInitByLet` (listing 2.10) i przypisujemy jej wartość 10 (linia 2.), po czym deklarujemy inną zmienną o tej samej nazwie i przypisujemy jej wartość 1 (linia 6.).

Listing 2.10

Deklaracja dwóch zmiennych o takiej samej nazwie za pomocą słowa kluczowego let

```
1 var sum = 10;
2 let sumInitByLet = 10;
3 // inne fragmenty kodu
4 var sum = 10;
5 console.log(sum);
6 var sumInitByLet = 1;
7 console.log(sumInitByLet);
```

Zobaczmy, jaki będzie skutek wyświetlenia tej zmiennej (rysunek 2.30).

Rysunek 2.30. Interpreter nie pozwala na utworzenie zmiennej o takiej samej nazwie, jaką ma zmienna zadeklarowana wcześniej za pomocą słowa kluczowego let

Tym razem skrypt nie wykonał się poprawnie, ale zwrócił informację o błędzie, wyświetlającą ostrzeżenie dotyczące 7. wiersza kodu, że zmienna `sumInitByLet` została już wcześniej zadeklarowana.

UWAGA

Jeśli piszemy skrypty, które zawierają wiele linijek, i nie chcemy sami dbać o to, by przy deklarowaniu zmiennych nie następowała nieoczekiwana zmiana wartości, używajmy `let`. Stosowanie tej konstrukcji pozwala nam uniknąć dużej liczby błędów w programie.

2.3.4. String (czyli zmienne typu napisowego)

Mamy — w ogólnym pojęciu — dwa główne rodzaje zmiennych: liczby i napisy. Do tej pory skupialiśmy się na liczbach.

Teraz skupmy się na napisach. Reprezentuje je w JS (i w innych językach) typ `String`. Napis to zestawienie słów, znaków lub cyfr. Często takie zestawienie nazywamy ciągiem znakowym lub łańcuchem.

Warto w tym miejscu przypomnieć sobie przykład, który zamieszczony był niemal na początku tego rozdziału (w punkcie 2.2.3). W naszym pierwszym, wygenerowanym automatycznie skrypcie wyświetlaliśmy komunikat „Hello world” (rysunek 2.5). To właśnie napis.

Do zdefiniowania zmiennej typu `String` należy użyć **podwójnego lub pojedynczego cudzysłowu** na początku i na końcu słowa.

UWAGA

W tym miejscu przechodzimy do pliku *strings.js*; oznacza to, że w *index.html* należy zmienić wskazanie na źródło skryptu. Zapewne pamiętasz, jak to zrobić. Jeśli nie, sprawdź we wcześniejszej części rozdziału.

Przykład 2.11

Na listingu 2.11 widać kilka przykładów poleceń deklarujących zmienne typu łańcuchowego. Jak widać, można używać zarówno cudzysłowu podwójnego (linia 1.), jak i pojedynczego (linia 2.). Gdybyśmy jednak w samym łańcuchu znakowym chcieli użyć apostrofu (linia 3.), musielibyśmy poprzedzić go lewym ukośnikiem (ang. *backslash*, \), by uniknąć zinterpretowania go jako znaku zamykającego łańcuch. Nie musimy tego robić, jeśli ciąg wydzielamy cudzysłowem podwójnym (linia 4.). Ostatnie polecenie tworzy napis, który — np. ze względu na jego długość — chcemy podzielić na dwa (lub więcej) wiersze.

Listing 2.11

Zmienne typu String możemy tworzyć za pomocą cudzysłowu podwójnego i cudzysłowu pojedynczego

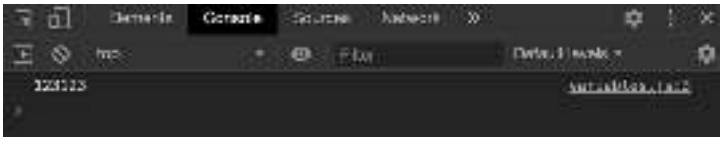
```
1 const nameDoubleQuote = "Marek";
2 const nameSingleQuote = 'Marek';
3 const sentenceSingleQuote = 'I\'m programmer';
4 const sentenceDoubleQuote = "I'm programmer";
5 const multiline = 'Dzień dobry! \
6                     niech ten dzień będzie słoneczny';
```

Warto dodać, że można także tworzyć napisy złożone z samych liczb. Na przykład zmienna utworzona w tym fragmencie kodu: `ciagLiczb = "123"` nie jest liczbą, lecz łańcuchem znakowym. Z tego względu nie da się wykonywać na niej obliczeń matematycznych, a próba dodania dwóch w ten sposób utworzonych zmiennych (listing 2.12) przyniesie z pewnością nieoczekiwane wyniki (rysunek 2.31)!

Listing 2.12

Połączenie dwóch zmiennych typu tekstowego

```
const numbers = "123";
const anotherNumber = "123";
console.log(numbers + anotherNumber);
```



Rysunek. 2.31. Widok wyświetlonych w konsoli połączonych napisów

Rezultatem jest wypisanie dwóch słów obok siebie. Pomimo wpisania czegoś, co przypominało liczby (ale nie było liczbami), użycie cudzysłowu spowodowało, że obie zmienne zostały potraktowane jako łańcuchy znakowe. Napis dodany do napisu jest zaś innym napisem.

Na tym etapie umiesz już rozróżniać ciągi znaków (napisy) i liczby. Potrafisz definiować zmienne, które pozwalają na używanie ich w interakcji z użytkownikiem lub programem. To bardzo ważne, podczas programowania będziesz z tych umiejętności korzystać wielokrotnie.

UWAGA

W C++ w celu utworzenia zmiennej typu łańcuchowego (napisu) należy najpierw dodać bibliotekę `string`:

```
#include <string>
```

Kiedy to zrobimy, możemy zadeklarować zmienną poleceniem:

```
std::string message = "wiadomość dla użytkownika";
```

Bardzo ważne jest to, że nazwę typu zmiennej będącej ciągiem znaków zapisuje się w C++, rozpoczynając od małej litery: `string`. W języku Java ten sam typ zmiennej deklaruje się, używając zapisu zaczynającego się wielką literą: `String`.

Trzeba być tego świadomym i zachowywać ostrożność, jeśli pracując nad jakimś projektem, „przełączamy się" między różnymi językami. Niestety bardzo często w takich przypadkach zdarzają się błędy składni spowodowane właśnie przez błędne nazwy typów zmiennych.

2.3.5. Zmienne typu logicznego (boolean)

Innym bardzo istotnym typem zmiennej jest `boolean`. Zmienna tego typu może przyjmować tylko dwie wartości logiczne: `true` albo `false`, czyli prawdę albo fałsz (listing 2.13).

Listing 2.13

Deklarowanie zmiennych typu boolean

```
let correct = true;

let notCorrect = false;
```

Interpreter „wie", że tak zadeklarowane zmienne są typu logicznego, bo tworząc je, przypisaliśmy do nich wartości logiczne: `true` (prawda) albo `false` (fałsz).

Więcej o wykorzystaniu tego typu zmiennych napiszemy w podrozdziale dotyczącym instrukcji warunkowych.

2.3.6. Interakcja z użytkownikiem

Do tej pory tworzyliśmy zmienne bez interakcji z użytkownikiem. Tym razem spróbujemy zaangażować go w działanie aplikacji, pobierając od niego dane. Jednym z najprostszych sposobów na to jest użycie metody `prompt`.

Metoda w kontekście programowania jest czynnością do wykonania (dlatego zwykle nazywana jest za pomocą czasownika). Podczas ćwiczeń z WF-u od czasu do czasu słychać polecenia: „Wykonajcie 15 skłonów" czy „Zróbcie 30 przysiadów". Oba te polecenia znakomicie nadawałyby się na nazwy metod w programowaniu. Wywołanie tych metod mogłoby np. wyglądać tak: `wykonajSklony(15)` oraz `zrobPrzysiady(30)`. W takim zapisie część przed nawiasem to nazwa (etykieta) metody. Jeśli jest dobrze utworzona, informuje o tym, *co robi* kod metody. Liczba w nawiasie zaś to jej parametr, określający dane, na jakich metoda ma pracować — w tym przypadku informuje on, ile razy mają być wykonane wspomniane w etykiecie czynności.

Metod używaliśmy już zresztą wcześniej. Jeśli wrócisz do naszego pierwszego skryptu, wyświetlającego komunikat „Hello world" (rysunek 2.5), zobaczysz, że robił to za pomocą metody `alert` z parametrem w postaci łańcucha znakowego `"Hello world"`. Wywołanie metody wyglądało wówczas tak: `alert("Hello world")`.

Wróćmy do metody `prompt`, która wyświetli użytkownikowi okno dialogowe (czyli okno, w którym on może coś wpisać, a my możemy to przechwycić i wykorzystać).

Przykład 2.12

Oto wywołanie metody (listing 2.14). Warto zauważyć, że zwrócony przez użytkownika tekst przypisujemy od razu do zmiennej `choice`, dzięki czemu będziemy mogli ją później wykorzystać do odwołania się do treści przekazanego komunikatu.

Listing 2.14

Składnia metody prompt

```
let choice = prompt('Co sądzisz o nauce programowania?');
```

Wynik wykonania tej instrukcji jest widoczny na rysunku 2.32.

I tak oto nawiązaliśmy pierwszy interaktywny kontakt z użytkownikiem.

Rysunek 2.32. Okno dialogowe wyświetlone za pomocą metody prompt

Teraz wypada wyświetlić w konsoli wszystkie uzyskane informacje (listing 2.15).

Listing 2.15

Za pomocą tych poleceń wyświetlamy w konsoli pobrany od użytkownika tekst

```
1 const statement = 'Nauka programowania jest ';
2 let choice = prompt('Co sądzisz o nauce programowania?');
3 console.log(statement + choice);
```

Jak widać, w pierwszej linijce utworzyliśmy stałą `statement`, a w drugiej poprosiliśmy użytkownika o wyrażenie opinii na temat programowania, zaś wpisany przez niego tekst przypisaliśmy do zmiennej `choice`. W ostatnim wierszu widzimy znane już nam polecenie `console.log`. Nowością jest to, że — jak widać — za pomocą plusa można łączyć łańcuchy znakowe. (To właśnie dlatego `"123" + "123"` da w wyniku 123123, a nie 246, o czym wspominaliśmy już wcześniej). Efekt wywołania kodu z listingu 2.15 po wpisaniu przez użytkownika słowa `"OK"` (czyli podstawiliśmy słowo `"OK"` do zmiennej `choice`) pokazano na rysunku 2.33.

Rysunek 2.33. Tekst pobrany od użytkownika wyświetlony w konsoli

Od tej pory naszą zmienną `choice` można przekazać bezpośrednio do łańcucha wynikowego poprzez wskazanie kompilatorowi znacznikiem `$` poprzedzającym wpisaną w nawiasach klamrowych nazwę zmiennej i ujęcie napisu w odwrócone apostrofy (`` `napis` ``) zamiast zwykłych (`'napis'`) — to tzw. łańcuch szablonowy (ang. *template literal*). Czyli to, co wpisał użytkownik, można przekazać dalej za pomocą składni `${choice}` bezpośrednio do łańcucha tekstowego. Upraszcza to wypisanie całego komunikatu (listing 2.16).

Listing 2.16

Łańcuch szablonowy upraszcza łączenie napisów ze zmiennymi JS

```
const choice = prompt('Co sądzisz o nauce programowania?');
const message = `Nauka programowania jest ${choice}`;
console.log(message);
```

2.3.7. Tablice

Do tej pory przechowywaliśmy w zmiennych pojedyncze wartości. Czy istnieje sposób na to, by w jednej zmiennej zawrzeć wiele wartości tego samego typu, np. w zmiennej `labels` przechować ceny produktów zapisane na etykietach sklepowych? Coś takiego umożliwiają tablice. Poniżej zadeklarowaliśmy zmienną tablicową `labels` (listing 2.17).

Przykład 2.13

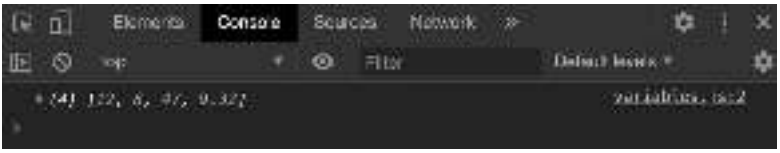
Listing 2.17

Deklarowanie i wyświetlanie zmiennej tablicowej

```
let labels = [12.0, 8, 47, 9.32];
console.log(labels);
```

Nowym elementem składni są nawiasy kwadratowe, wewnątrz których wpisujemy, rozdzielone przecinkami, wartości do przechowywania w zmiennej.

Kiedy uruchomimy nasz skrypt (*array.js* — pamiętaj o zmianie odwołania w pliku *index.html*), otrzymamy wynik przedstawiony na rysunku 2.34.



Rysunek 2.34. Wynik wyświetlenia w konsoli tablicy zadeklarowanej na listingu 2.17

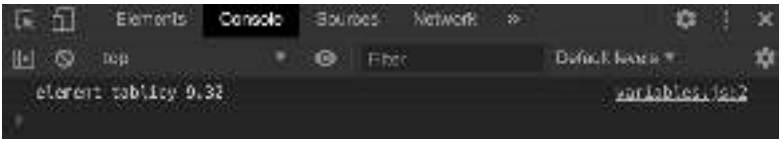
Powyżej wyświetliliśmy całą zawartość tablicy. Możemy jednak wypisać jej dowolny element. W tym celu po nazwie tablicy wpisujemy w nawiasach kwadratowych numer interesującego nas elementu (jego indeks). Widać to na listingu 2.18.

Listing 2.18

Wyświetlenie w konsoli elementu tablicy o indeksie 3

```
console.log('element tablicy ' + labels[3]);
```

I tutaj czeka nas niespodzianka. Otóż w konsoli otrzymamy taki wynik (rysunek 2.35):



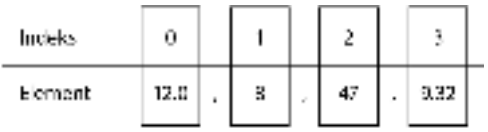
Rysunek 2.35. Wyświetlony w konsoli element tablicy z indeksem 3

Być może spodziewaliśmy się ujrzeć trzeci element, czyli 47. W rzeczywistości w składni `labels[3]` odwołujemy się nie do trzeciego elementu, ale do **elementu z indeksem równym 3. Indeksy elementów tablicy numerowane są zaś od zera.** Aby ujrzeć

UWAGA

Indeksy elementów tablicy numerowane są od zera!

w wyniku 47, musielibyśmy zatem użyć odwołania `labels[2]`; 12 zobaczymy po wpisaniu `labels[0]` itd. (rysunek 2.36).



Rysunek 2.36. Odwzorowanie indeksu na element tablicy

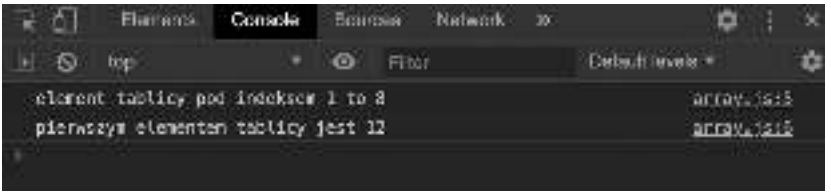
Zatem dzięki tym dwóm poleceniom (listing 2.19):

Listing 2.19

Drugi element tablicy i element z indeksem 2 to nie to samo

```
console.log("element tablicy pod indeksem 1 to " + labels[1]);
console.log("pierwszym elementem tablicy jest " + labels[0]);
```

uzyskamy komunikaty z rysunku 2.37:



Rysunek 2.37. Pierwszym elementem tablicy jest ten z indeksem 0

Na początku dość trudno się do tego przyzwyczaić, co jest przyczyną wielu błędów. Gdybyśmy bowiem chcieli wyświetlić ostatni, czwarty element naszej tablicy i użylibyśmy do tego odwołania `labels[4]` (listing 2.20), wyszlibyśmy poza granicę tablicy — element z indeksem 4 nie został bowiem w niej zdefiniowany.

Listing 2.20

Takie polecenie spowoduje wykroczenie poza zakres tablicy

```
console.log("element tablicy " + labels[4]);
```

Najczęściej wykorzystywaną właściwością tablicy jest jej długość (ang. *length*). Dzięki niej możemy uzyskać informację o tym, ile elementów jest w tablicy. Zrobimy to, wpisując najpierw nazwę tablicy, następnie kropkę, po niej zaś nazwę właściwości: `labels.length` (listing 2.21).

Listing 2.21

Pobieramy informację o długości tablicy

```
console.log('długość tablicy/liczba elementów w tablicy ' + labels.length);
```

Co oczywiście zwróci 4 (rysunek 2.38).

Rysunek 2.38. Tablica labels ma 4 elementy

Dodanie elementu do już istniejącej tablicy uzyskamy poprzez metodę `push` wywołaną bezpośrednio na tablicy, czyli za pomocą polecenia `labels.push(34);`.

Ponieważ `push` dodaje elementy na końcu tablicy, obecnie ostatnim elementem `labels` jest 34.

UWAGA

Deklarację tablicy wraz z jednoczesnym przypisaniem do niej wartości uzyskamy w C++ za pomocą polecenia:

```
std::string tablica[] = {"element1", "element 2", "3"};
```

Warto zwrócić szczególną uwagę na to, że w JS przy deklaracji tablicy używamy nawiasów kwadratowych, a w C++ klamrowych.

Bardzo ważne jest też wskazanie na początku rozmiaru tablicy.

W JS wielkość tablicy jest dynamiczna, co można łatwo zauważyć poprzez dynamiczne dodawanie elementów za pomocą metody `push`.

W przypadku C++ nie uzyskamy rozmiaru tablicy za pomocą znanej nam z JS właściwości `length`, bo jej w tym języku nie ma (występuje natomiast także w Javie). W związku z tym należałoby obliczyć różnicę bajtów pomiędzy pierwszym a ostatnim elementem tablicy.

2.4. Instrukcje warunkowe (conditionals)

W życiu często stoimy przed wieloma wyborami. To, jaką podejmiemy decyzję, jest uzależnione od spełnienia różnorakich warunków, czasami prostych („Jeśli będzie ładna pogoda, pójdę na spacer, jeśli nie, poczytam książkę”), innym razem dość skomplikowanych („Jeśli uda mi się zdać egzamin maturalny z wysokim wynikiem, pójdę na dobre studia, o których marzę. Dodatkowo, jeśli rodzicom uda się zaoszczędzić trochę pieniędzy, wybiorę lepszą uczelnię poza moim miejscem zamieszkania”).

W programowaniu jest podobnie. Bardzo często działanie programu będzie uzależnione od danych dostarczonych np. przez użytkownika. Najprostszym przykładem jest okno dialogowe, w którym pytamy, czy chce on subskrybować ofertę wybranego sklepu. Jeśli kliknie *Tak*, powinniśmy umożliwić mu przejście do formularza służącego do wprowadzania danych. Jeśli wybierze odpowiedź *Nie*, powinien pozostać na bieżącej stronie. Do obsługiwania takich sytuacji służą instrukcje warunkowe.

UWAGA

Zanim zaczniesz wykonywać kody z tego podrozdziału, przeczytaj go dokładnie w całości!

Przykład 2.14

Gdy podczas planowania naszej aplikacji pojawią się słowa *jeśli* lub *kiedy*, to znak, że powinniśmy użyć bloku kodu w takiej postaci (listing 2.22):

Listing 2.22

Instrukcja warunkowa if

```
if (4 > 3) {
    console.log("4 jest większe od 3");
}
```

Rozłóżmy go na części pierwsze. Słowo *jeśli* to po angielsku *if*, dlatego to właśnie od niego (`if`) zaczyna się instrukcja warunkowa. Warunek, jaki ma być spełniony, podajemy w nawiasach okrągłych. W naszym przykładzie sprawdzamy, czy 4 jest większe od 3; oczywiście tak jest, w konsoli pojawi się zatem stwierdzenie, że 4 jest większe od 3. Gdybyśmy natomiast zamienili znak większości na znak równości, nic by się nie wykonało. Kod w klamrach następujący po instrukcji warunkowej wykonuje się tylko wtedy, kiedy warunek jest prawdziwy.

UWAGA

Zapewne pamiętasz, że omawiając zmienne typu `boolean` (logiczne), wspominaliśmy, że pojawią się one przy okazji instrukcji warunkowych. To właśnie ten moment. Zmienna taka przyjmuje, jak powiedzieliśmy, tylko dwie wartości: prawdę (`true`) albo fałsz (`false`). W zależności od tego, która z tych wartości zostanie zwrócona w wyniku sprawdzenia warunku, kod w nawiasach klamrowych zostanie wykonany albo nie.

UWAGA

Bardzo ważne jest tutaj prawidłowe użycie nawiasów klamrowych. Tak jak w przypadku nawiasów okrągłych, musimy pamiętać, by zamykać każdy otwarty nawias. Jeśli po warunku otworzymy nawias klamrowy i nie zamkniemy go na końcu bloku instrukcji do wykonania, zostanie zwrócony błąd.

2.4.1. Operatory warunkowe

Bardzo często rolę operatorów używanych w instrukcjach warunkowych odgrywają operatory matematyczne:

- > większe niż;
- < mniejsze niż;
- >= większe lub równe;
- <= mniejsze lub równe;
- == równe — operator porównuje tylko wartości;
- === równe — ten operator porównuje wartości i zarazem typy.

UWAGA

W pierwszej chwili może dziwić, że porównanie wartości wykonywane jest za pomocą dwóch znaków równości, a nie — jak w matematyce — jednego. Aby to zrozumieć, trzeba pamiętać, że pojedynczy znak „równa się" używany jest do przypisania wartości do zmiennej.

UWAGA

W przypadku operatorów porównania łączonych (>= oraz <=) trzeba pamiętać, że znak równości wpisywany jest **po** znaku mniejszości lub większości.

Przykład 2.15

Teraz skupmy się na dwóch ostatnich operatorach (listing 2.23 — przykład dostępny w skrypcie *simple-conditionals.js*).

Listing 2.23

Różnica w działaniu operatorów == i ===

```
if ('7' == 7) {
    console.log("wartości są sobie równe");
}
if ('7' === 7) {
    console.log("wartości są sobie równe, ale typy nie;
po lewej stronie jest String, po prawej liczba");
}
```

Rezultatem wykonania tego skryptu będzie wyświetlenie komunikatu tylko z pierwszego warunku, bo drugi (zgodnie z opisem) porównuje także typy, a żaden napis nie jest tym samym co liczba.

UWAGA

Tylko w ramach ciekawostki wspomnieliśmy o operatorze ==. W dalszej części tej książki będziemy używali wyłącznie ===, aby mieć pewność, że porównujemy wartości tego samego typu — w końcu, jak wspomnieliśmy, `"7"` to nie to samo co 7.

Należy też zauważyć, że w językach C++ czy Java, w których typ zmiennej jest jawnie określany, nie trzeba porównywać typów, dlatego do porównywania używany jest wyłącznie operator ==.

2.4.2. Wykorzystanie zmiennych typu boolean w instrukcjach warunkowych

Przykład 2.16

Załóżmy, że chcemy napisać krótki program, w którym prosimy użytkownika, by zgadł dzień naszych urodzin. Jeśli zgadnie, wówczas wyświetlimy mu w konsoli komunikat z gratulacjami. W przeciwnym wypadku komunikat będzie informował o tym, że próba była nieudana (listing 2.24).

Listing 2.24

Instrukcja if wykorzystująca zmienne typu boolean

```
1 const dayOfBirthday = 14;
2 const guess = prompt("Zgadnij dzień moich urodzin");
3
4 let correct = false;
5
6 if (+guess === dayOfBirthday) {
```

```
 7     correct = true;
 8 }
 9
10 if (correct === true) {
11     console.log(`Zgadza się, dzień moich urodzin to: ${dayOfBirthday}`);
12     document.write("BRAWO!");
13 }
14
15 if (correct === false) {
16     console.log(`Zgaduj dalej`);
17     document.write("Odśwież stronę, by spróbować jeszcze raz");
18 }
```

Przeanalizujmy nasz kod.

Linijka 1. Utworzenie stałej `dayOfBirthday` typu liczbowego i przypisanie do niej wartości 14.

Linijka 2. Utworzenie stałej `guess`, do której zostanie przypisana wartość wpisana przez użytkownika.

Linijka 4. Utworzenie zmiennej `correct`, do której na początek przypiszemy wartość `false` (fałsz).

Linijka 6. Ustawienie warunku — jeśli (*if*) wpisana przez użytkownika wartość (`guess`) jest równa pod względem wartości i typu zmiennej `dayOfBirthday`.

UWAGA

Znak dodawania (+) w tej linijce to tak zwany jednoargumentowy plus (ang. *unary plus*), realizujący zamianę typów. W tym przypadku zmienna łańcuchowa `guess` (trzeba pamiętać, że każda wartość wpisana przez użytkownika jest ciągiem znaków, czyli napisem) zostaje zamieniona na liczbę, by można było porównać obie wartości.

Zamiast tego można by było w tym miejscu użyć operatora ==, ale warto poznawać nowe funkcjonalności.

Linijka 7. Jeśli po sprawdzeniu warunku w linijce 6. zwrócona zostaje wartość `true` (prawda), to wówczas właśnie tę wartość przypiszemy do zmiennej `correct`.

Linijka 10. Jeśli wartość w zmiennej `correct` jest równa `true`, to wówczas…

Linijka 11. …wypisujemy w konsoli wskazany napis z gratulacjami.

Linijka 15. i następne. Jeśli wartość zmiennej `correct` jest równa `false`, to wypiszemy w konsoli inny napis, informujący o nieudanej próbie.

Kod jest oczywiście dostępny w przygotowanych materiałach; wystarczy odpowiednio zmienić w pliku *index.html* odwołanie do skryptu.

UWAGA

W C++ nazwa typu zmiennej logicznej to `bool`, a w Java — `boolean`.

2.4.3. Instrukcja warunkowa oraz operator logiczny AND

Przykład 2.17

Załóżmy, że idziemy do sklepu kupić buty. Wiadomo, że buty muszą mieć odpowiedni rozmiar oraz ich cena musi mieścić się w naszym z góry ustalonym budżecie.

Jak wyglądałoby ujęcie tego problemu w programowaniu?

Musimy utworzyć dwie zmienne (a właściwie stałe), przechowujące informacje o poszukiwanym rozmiarze buta i o maksymalnej cenie, jaką możemy zapłacić (listing 2.25). Jeśli buty, które sobie upatrzyliśmy, spełniają te dwa warunki (co do rozmiaru i ceny), to możemy je kupić.

Listing 2.25

Zadeklarowanie stałych

```
const shoesSize = 39;

const price = 230;
```

Zapewnienie interakcji z użytkownikiem także nie powinno przysporzyć problemów (listing 2.26).

Listing 2.26

Pobieranie danych

```
const answerSize = prompt("Jaki jest dostępny rozmiar buta?");

const answerPrice = prompt("Ile kosztują buty?");
```

Teraz utworzymy zmienną przechowującą stan decydujący o tym, czy możemy kupić buty, czy nie. Ponieważ mamy tylko dwa możliwe stany, na pewno będzie to zmienna typu logicznego (listing 2.27).

Listing 2.27

Zmienna logiczna canBuy będzie przyjmować tylko wartość prawda/fałsz (true/false)

```
let canBuy;
```

Teraz tworzymy warunek określający, czy w sklepie dostępne są buty w rozmiarze 39; jeśli tak, to ustawiamy zmienną `canBuy` na `true` (listing 2.28):

Listing 2.28

Warunek dotyczący rozmiaru butów

```
if (+answerSize === 39){
    canBuy = true;
}
```

UWAGA

Znak dodawania przy zmiennej `answerSize` powoduje rzutowanie napisu na liczbę.

Ponieważ rozmiar podany przez sprzedawcę to 39, przypisaliśmy do zmiennej `canBuy` wartość `true`. Czy jednak się nie pospieszyliśmy? Przecież mamy do rozpatrzenia jeszcze jeden warunek. Musimy sprawdzić, czy cena obuwia mieści się w naszym zaplanowanym budżecie. Innymi słowy, buty muszą kosztować nie więcej niż 230 zł.

Zmodyfikujmy zatem nasz kod (listing 2.29). Musimy dodać drugi warunek, połączony z pierwszym spójnikiem *i* (*oraz*). **W programowaniu zapisywany on jest zazwyczaj jako `&&`.**

UWAGA

Spójnik *i* zapisujemy w JS jako `&&`.

Listing 2.29

Dodajemy drugi warunek do spełnienia

```
if (+answerSize === 39 && +answerPrice <= 230) {
    canBuy = true;
}
```

Po modyfikacji zmienna `canBuy` przyjmie wartość `true` tylko wtedy, kiedy zostaną spełnione **oba** warunki — rozmiar buta równy 39 i cena nie większa niż 230 zł. Nie potrzebujemy przecież butów za małych ani za dużych, nie kupimy też butów w odpowiednim rozmiarze, jeśli nie będziemy mieli wystarczająco dużo pieniędzy.

UWAGA

Cały kod dostępny jest w skrypcie *shopping.js*.

2.4.4. Instrukcja warunkowa oraz operator logiczny OR

Kontynuujmy nasz przykład z butami. Mamy już dobrany odpowiedni rozmiar buta, cena też jest prawidłowa, czyli nie przekracza naszego budżetu. Ale czy to oznacza, że na pewno je kupimy? Mogą przecież po prostu nam się nie podobać, np. mogą mieć inny kolor niż ten, który byśmy chcieli.

Przyjmijmy, że buty, które chcemy kupić, powinny być koloru białego lub czarnego.

Przykład 2.18

Standardowo tworzymy stałe, do których przypiszemy wskazane przez nas wartości (listing 2.30):

Listing 2.30

Deklarowanie stałych przechowujących informację o kolorze

```
const preferredColorWhite = 'białe';
const preferredColorBlack = 'czarne';
```

Następnie pobieramy odpowiedź od sprzedawcy (listing 2.31):

Listing 2.31

Stała sellerAnswer przechwytuje odpowiedź udzieloną przez sprzedawcę

```
const sellerAnswer = prompt("Jaki jest dostępny kolor butów?");
```

Oczywiście potrzebna nam zmienna do przechowywania wartości decyzji — kupimy te buty czy nie (listing 2.32):

Listing 2.32

Zmienna przechowująca stan decyzji

```
let canBuy;
```

Zapis warunku będzie podobny do zapisu warunku logicznego AND. Zmieniamy tylko operator logiczny; zamiast `&&` odpowiadającego spójnikowi *i* wpisujemy **dwa pipeline'y — czyli `||`. Znaki te odpowiadają spójnikowi *lub* (OR).**

UWAGA

Spójnik *lub* zapisujemy w JS jako `||` (listing 2.33).

Listing 2.33

Instrukcja warunkowa z operatorem OR

```
if (sellerAnswer === preferredColorWhite ||
sellerAnswer === preferredColorBlack) {
    canBuy = true;
}
```

Wynik takiego warunku będzie spełniony (instrukcja zwróci `true`), jeśli w sklepie znajdą się buty w kolorze białym lub czarnym. Oczywiście prawdę uzyskamy także w przypadku, kiedy jakaś para będzie biała, a inna czarna. Zmienna `canBuy` przyjmie wartość `false` tylko wtedy, gdy w sklepie nie będą mieli butów w żadnym z tych kolorów.

Następnie wyświetlamy w konsoli odpowiednią informację, w zależności od wartości zmiennej `canBuy` (listing 2.34):

Listing 2.34

Wyświetlenie w konsoli jednego z alternatywnych komunikatów

```
if (canBuy) {
    console.log("Udało mi się zdobyć wymarzony kolor butów");
} else {
    console.log("Szukam butów dalej");
}
```

UWAGA

Polecenie `if(canBuy)` to skrócony zapis `if(canBuy === true)`.

2.5. Instrukcje sterujące i pętle

W życiu codziennym bardzo często posługujemy się zdaniami złożonymi albo powtarzamy czynności. Właśnie do obsługi takich przypadków w programowaniu potrzebne nam są instrukcje sterujące.

2.5.1. Instrukcja sterująca if ... else-if

Przeanalizujmy przykład powiązany ze szkołą. Świetnie nadaje się on do zaprezentowania instrukcji sterującej — w zależności od spełnienia jednego z wielu warunków przenosi ona kod do odpowiedniego wiersza, który jest wykonywany. Tak się dzieje, kiedy sprawdzamy wynik uzyskany na sprawdzianie w celu ustalenia oceny.

Przykład 2.19

Podczas sprawdzianu można uzyskać maksymalnie 100 punktów. Aby dostać piątkę, należy zdobyć powyżej 90 punktów, czwórka to wartość między 80 a 90, trójka między 70 a 79, a dwójkę dostajemy wtedy, gdy punkty mieszczą się między 60 a 69. Oczywiście, jeśli nie spełniliśmy żadnego z tych wymagań, to wtedy otrzymujemy jedynkę.

Jak rozpocząć ten skrypt? Najpierw należy zapytać użytkownika, ile punktów uzyskał (listing 2.35). Ze względu na to, że jest to zmienna, która nie ulega modyfikacji (przecież w trakcie obliczania nie zapytamy użytkownika ponownie o to, ile uzyskał punktów, prawda?), użyjemy `const`.

Listing 2.35

Prosimy użytkownika o podanie liczby punktów i zapisujemy odpowiedź w zmiennej `score`

```
const score = prompt("Ile punktów uzyskano?");
```

Następnie tworzymy (ale bez inicjowania, czyli przypisania początkowej wartości) zmienną do przechowywania oceny (`grade` — listing 2.36).

Listing 2.36

Zmienna `grade` do przechowywania oceny

```
let grade;
```

Nie będziemy inicjować tej zmiennej, gdyż na początku nie wiemy jeszcze, jaką ocenę otrzymał nasz użytkownik.

Teraz napiszmy podane w opisie warunki uzyskania poszczególnych ocen. Aby użytkownik dostał piątkę, musi mieć powyżej 90 punktów (listing 2.37).

Listing 2.37

Warunek uzyskania piątki

```
if (score > 90) {
    grade = 5;
}
```

Teraz czas na sprawdzenie, czy użytkownikowi nie należy się czwórka (listing 2.38). Oczywiście będzie ono wykonywane tylko wtedy, kiedy będzie już wiadomo, że uzyskane punkty nie pozwalają na otrzymanie oceny bardzo dobrej. Tutaj z pomocą przychodzi nam instrukcja `else if`. Podobnie jak `else`, której już używaliśmy, nie występuje ona nigdy sama, może być jedynie uzupełnieniem wcześniejszej instrukcji `if`. Jeżeli warunek określony w `if` nie jest spełniony, `else if` informuje, że należy wejść do kolejnego warunku.

Listing 2.38

Warunek uzyskania czwórki

```
if (score > 90) {
    grade = 5;
} else if (score >= 80) {
    grade = 4;
}
```

Kolejne warunki są analogiczne (listing 2.39):

Listing 2.39

Pozostałe warunki

```
if (score > 90) {
    grade = 5;
} else if (score >= 80) {
    grade = 4;
} else if (score >= 70) {
    grade = 3;
} else if (score >= 60) {
    grade = 2;
} else {
    grade = 1;
}
```

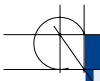
Zauważ, że ciąg instrukcji `if … else if` kończy polecenie `else`; zawiera ono kod do wykonania w ostateczności, jeśli żaden z poprzednich warunków nie został spełniony.

Na końcu wypiszmy uzyskaną ocenę w konsoli, wykorzystując znane już nam łańcuchy szablonowe (listing 2.40).

Listing 2.40

Wypisanie oceny

```
console.log(`Twoja ocena to ${grade}`);
```



UWAGA

Ten przykład warto przepisać samodzielnie 3, 4 razy, by utrwalić sekwencję zapisywania kolejnych instrukcji `else if`.

Pamiętaj, że programowanie nie jest umiejętnością, której nauczysz się, wyłącznie czytając; potrzebna jest praktyka oraz bardzo dużo czasu, a także wola walki z ewentualnymi problemami. Rzeczy, których dziś jeszcze nie rozumiesz, za 2 – 3 tygodnie będą proste, o ile nie przestaniesz ćwiczyć.

2.5.2. Pętla for

Potrafisz już samodzielnie napisać prostą aplikację. Zajmijmy się teraz kolejną, jakże ważną funkcjonalnością — powtarzaniem kroków kilka razy w tak zwanej pętli. Pętle pozwolą nam znacznie ulepszyć nasze programy. Jest kilka rodzajów pętli. W kolejnych punktach tego rozdziału omówimy je kolejno, zaczynając od pętli `for`.

Pętlę `for` rozpoczynamy od słowa kluczowego `for`, po którym w nawiasach okrągłych umieszczamy kolejne kroki rozdzielone średnikami:

```
for (start; warunek_zakończenia_pętli; krok)
```

W tej składni *`start`* to punkt, w którym zaczynamy pętlę. Zazwyczaj jest nim tzw. zmienna licznikowa (zwykle `i`), do której przypisujemy określoną wartość liczbową. *`warunek_zakończenia_pętli`* to informacja o tym, kiedy kończymy pętlę — np. kiedy zmienna licznikowa osiągnie jakąś określoną w tym miejscu wartość. *`krok`* określa instrukcję, która zostanie uruchomiona przy każdym wykonaniu pętli — zazwyczaj w tym miejscu następuje zwiększenie licznika `i` o zadaną wartość. Na przykład pętla z listingu 2.41 wyświetli w konsoli liczby od 0 do 100 — ale tylko co piątą (0, 5, 10, 15 itd.), ponieważ krok zdefiniowaliśmy w ten sposób, że zmienna licznikowa `i` jest po każdej iteracji pętli zwiększana o 5.

Listing 2.41

Wyświetlenie w pętli co piątej liczby z zakresu od 0 do 100

```
for (i = 0; i <= 100; i += 5) {
    console.log(i);
}
```

Przykład 2.20

Aby to wyjaśnić jeszcze lepiej, przejdźmy do innego przykładu. Stwórzmy tablicę ulic obiegających budynek, zawierającą ulice: Wita Stwosza, Bażyńskiego, Grunwaldzką i Abrahama.

Tworzymy zmienną statyczną (gdyż ulice wokół budynków nie ulegną zmianie) — listing 2.42.

Listing 2.42

Utworzenie tablicy ulic i przypisanie jej do stałej streets

```
const streets = ["Bażyńskiego", "Grunwaldzka", "Abrahama", "Wita Stwosza"];
```

Teraz piszemy kod pętli (listing 2.43).

Listing 2.43

Pętla for wyświetlająca jedną po drugiej ulice przechowywane w stałej streets

```
for (let i = 0; i < streets.length; i += 1) {
    console.log(streets[i]);
}
```

Najpierw utworzyliśmy — zgodnie ze składnią pętli `for` — zmienną licznikową i przypisaliśmy do niej wartość 0; następnie zdecydowaliśmy, że skończymy wykonywać kroki, gdy zmienna ta będzie równa liczbie elementów tablicy (czyli długości tablicy), a ponieważ zależy nam, by wyświetlić kolejne elementy tablicy, nakazujemy powiększenie wartości `i` o 1 (ten zapis może również wyglądać tak: `i = i+1` lub `i++`). Oczywiście, gdy powiększany w ten sposób licznik `i` przyjmie wartość 4 (taka jest długość tablicy), wykonywanie pętli się zakończy.

2.5.2.1. Iterowanie pętlą for po tablicach dwuwymiarowych

Tablica dwuwymiarowa to coś, co przypomina dane uporządkowane na osiach X i Y. Każdy punkt w strukturze dwuwymiarowej identyfikowany jest za pomocą pary dwóch liczb.

Przykład 2.21

W programowaniu tablice dwuwymiarowe mogą się przydać np. podczas zapisywania rozgrywki szachowej. Doskonale nadają się także do zaprogramowania quizu. Każdy quiz ma przecież pytania i odpowiedzi. Możemy zatem utworzyć tablicę dwuwymiarową zawierającą zarówno pytania, jak i odpowiedzi na nie. Oto kod tworzący taką tablicę (listing 2.44):

Listing 2.44

Deklaracja tablicy dwuwymiarowej

```
const questionAndAnswer = [
    ['Gdziekolwiek', 'Edward Stachura'],
    ['Wyspa', 'Jonasz Kofta'],
    ['Życie', 'Jan Twardowski']
];
```

Tablicę dwuwymiarową deklarujemy jak zwykłą tablicę (za pomocą nawiasów kwadratowych), jednak zamiast rozdzielonych przecinkami pojedynczych wartości (liczb, napisów itp.) tutaj mamy ujęte w nawiasy kwadratowe i rozdzielone przecinkami **pary** wartości. W przypadku naszego quizu każdą taką parę tworzy tytuł utworu literackiego i nazwisko jego autora.

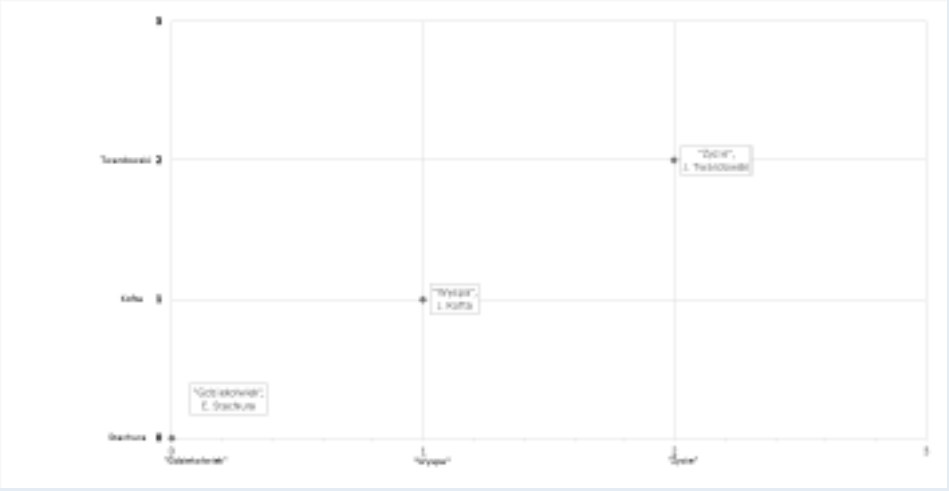
UWAGA

Strukturę tak opracowanego (za pomocą tablicy dwuwymiarowej) quizu obrazuje tabela 2.2.

Tabela 2.2. Odwzorowanie tablicy dwuwymiarowej za pomocą tabeli

	0	1
0	„Gdziekolwiek”	E. Stachura
1	„Wyspa”	J. Kofta
2	„Życie”	J. Twardowski

Można też przedstawić ją za pomocą wykresu (rysunek 2.39).



Rysunek 2.39. Wizualizacja tablicy dwuwymiarowej jako wykresu

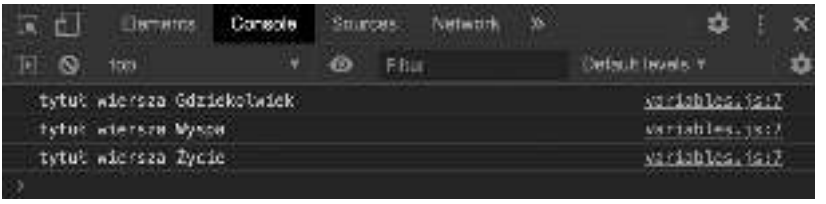
Teraz wykorzystamy znaną już nam pętlę `for` do wyświetlenia zawartości tablicy `questionAndAnswer` (listing 2.45):

Listing 2.45

Iteracja po dwuwymiarowej tablicy za pomocą pętli for

```
for (let i = 0; i < questionAndAnswer.length; i += 1) {
    console.log("tytuł wiersza " + questionAndAnswer[i][0]);
}
```

Jak widać, kod pętli (poza nazwą tablicy) w ogóle się nie zmienił. Nieco inaczej natomiast będzie wyglądał efekt jego wykonania, bo tym razem wyświetlone zostaną po kolei pary powiązanych wartości (rysunek 2.40).



Rysunek 2.40. Rezultat wykonania pętli for

W trakcie nauki programowania najważniejsze jest powtarzanie. Dlatego warto przeczytać ten punkt rozdziału jeszcze raz i powtórzyć sobie wszystko, co zostało w nim omówione, oczywiście wykonując zamieszczone tu kody.

2.5.3. Pętla for ... of

Istnieje odmiana pętli `for`, która jest szczególnie przydatna do przeprowadzania operacji na obiektach iterowalnych, np. na tablicach (ale także kolekcjach czy łańcuchach znakowych), kiedy trzeba wykonać jakieś działanie na każdym elemencie takiego obiektu. To pętla `for ... of`. Rzecz jasna w takim przypadku można wykorzystać tradycyjną postać pętli `for`, jednak użycie `for ... of` będzie lepszym rozwiązaniem, choćby dlatego, że nie trzeba będzie zastanawiać się nad ustawianiem warunku wykonania pętli.

Oto składnia pętli `for ... of`:

```
for (zmienna of obiekt_iterowalny) {
    // kod do wykonania
}
```

W tym zapisie *`zmienna`* oznacza zmienną, która przyjmować będzie kolejno wartość każdego elementu obiektu, *`obiekt_iterowalny`* natomiast to obiekt, po którym możemy iterować, np. tablica, łańcuch znakowy bądź kolekcja.

Przykład 2.22

Załóżmy, że chcemy się dowiedzieć, jaka jest wartość sumy kolejnych liczb od 1 do 10 zapisanych w tablicy.

Listing 2.46 pokazuje, jak zaimplementować rozwiązanie takiego zadania z wykorzystaniem pętli `for`.

Listing 2.46

Obliczanie sumy liczb za pomocą pętli for

```
1 let suma = 0;
2 let liczby = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
3 for (let i = 0; i < liczby.length; i++) {
4     suma += liczby[i];
5 }
6 console.log(suma);
```

Wynik to, oczywiście, 55.

Zobaczmy teraz, jak wyglądałoby rozwiązanie tego zadania z wykorzystaniem pętli `for … of` (listing 2.47).

Listing 2.47

Sumowanie liczb za pomocą pętli for … of

```
1 let suma = 0;
2 let liczby = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
3 for (let liczba of liczby) {
4     suma += liczba;
5 }
6 console.log(suma);
```

Wynikiem także tym razem będzie 55.

Na czym polega korzyść z zastosowania takiego rozwiązania? Jeśli porównamy liczbę linii kodu w tych konkretnych przypadkach, zauważymy, że jest dokładnie taka sama. Porównajmy jednak deklaracje pętli — w wierszu 3. W pierwszym przypadku musieliśmy ustawiać zmienną licznikową (`i`), za jej pomocą deklarować, że kod pętli ma być wykonywany tak długo, jak długo `i` będzie mniejsza lub równa długości zadeklarowanej tablicy (`i < liczby.length`), na końcu powiększać `i` (`i++`). Stosując pętlę `for … of`, wskazujemy tylko, że instrukcje zawarte w ciele pętli mają być wykonane dla każdej wartości zmiennej `liczba`, a zmienna ta będzie przybierać wartość kolejnych elementów tablicy `liczby` (`let liczba of liczby`). Mamy zatem konstrukcję `(let i = 0; i < liczby.length; i++)` przeciwko `(let liczba of liczby)`. Ta druga jest znacznie prostsza i bardziej intuicyjna, w szczególności dla początkującego programisty.

UWAGA

Pętlę `for ... of` można zastosować tylko wtedy, kiedy chcemy przeprowadzić operacje na *każdym* elemencie obiektu iterowalnego. Gdybyśmy chcieli w naszym poprzednim przykładzie zsumować tylko co drugą (2, 4, 6, 8, 10) lub co trzecią (3, 6, 9) liczbę z tablicy `liczby`, pętla `for ... of` w niczym by nam nie pomogła. Doskonale natomiast sprawdziłaby się odpowiednio zaimplementowana pętla `for`. Pokazuje to kod z listingu 2.48.

Listing 2.48

Sumowanie co drugiego elementu tablicy za pomocą pętli for

```
let suma = 0;
let liczby = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
for (let i = 1; i < liczby.length; i += 2) {
    suma += liczby[i];
}
console.log(suma);
```

Tym razem wynik to 30.

Przykład 2.23

Jak wcześniej wspomniano, pętli `for ... of` można używać na dowolnym obiekcie iterowalnym, np. na ciągu znaków. Załóżmy, że chcielibyśmy wyświetlić w nowym wierszu każdy znak (włącznie ze spacjami) zdania „Witaj w świecie programowania". Umożliwi nam to kod z listingu 2.49.

Listing 2.49

Pętla for … of umożliwia wypisanie kolejnych znaków dowolnego napisu

```
let zdanie = "Witaj w świecie programowania";
for (let znak of zdanie) {
    console.log(znak);
}
```

2.5.4. Pętla while

Kiedy chcemy wykonywać jakieś działanie tak długo, jak długo prawdziwy jest określony warunek, możemy użyć pętli `while`. Jej składnia to:

```
while (warunek) {
    // kod do wykonania
}
```

UWAGA

Przed każdorazowym wykonaniem poleceń zawartych w ciele pętli interpreter sprawdza, czy warunek jest prawdziwy. Jeśli nie, pomija ciało pętli i od razu przechodzi do pierwszej linii kodu poza pętlą. Jeśli tak, wykonuje polecenia, po czym ponownie sprawdza warunek i jeśli wciąż jest on prawdziwy, wykonuje polecenia — i tak aż do momentu, kiedy warunek będzie fałszywy. Wtedy nastąpi wyjście z pętli.

Oznacza to, że kod zawarty w ciele pętli `while` *nie zostanie* wykonany w ogóle, jeśli warunek początkowy okaże się fałszywy.

Przykład 2.24

Załóżmy, że chcemy napisać program odliczający w dół od 10 do 0, na końcu zaś wyświetlić komunikat „Koniec odliczania”. Możemy do tego użyć kodu z listingu 2.50.

Listing 2.50

Odliczanie od 10 do 0

```
let licz = 10;
while (licz >= 0) {
    console.log(licz);
    licz--;
}
console.log("Koniec odliczania");
```

W wyniku wykonania tego kodu kolejna liczba będzie wyświetlana do momentu, gdy pomniejszana po każdej iteracji zmienna `licz` stanie się mniejsza od 0. Wtedy wykonywanie pętli zakończy się, a interpreter przejdzie do następnego wiersza (już poza pętlą), w wyniku czego wyświetlony zostanie komunikat o zakończeniu odliczania.

UWAGA

Konieczne jest zadbanie o to, by w pewnym momencie warunek wykonania pętli stał się fałszywy. W przeciwnym wypadku utworzymy tzw. **pętlę nieskończoną**, która będzie się wykonywać w nieskończoność, chyba że zostanie przerwana, np. poprzez wymuszenie zamknięcia programu.

Przykład 2.25

Załóżmy, że w programie z listingu 2.50 zapomnieliśmy wpisać polecenie `licz--`, które pomniejsza w każdej iteracji zmienną `licz`, sprawiając, że po odliczeniu do 0 i kolejnym pomniejszeniu warunek staje się fałszywy. Kod wyglądałby zatem tak, jak pokazano na listingu 2.51.

Listing 2.51

Pętla nieskończona

```
let licz = 10;
while (licz >= 0) {
    console.log(licz);
}
console.log("Koniec odliczania");
```

W takim przypadku warunek nigdy nie zostałby sfalsyfikowany, pętla wykonywałaby się więc raz za razem, wyświetlając kolejne dziesiątki, a komunikatu „Koniec odliczania" nie zobaczylibyśmy nigdy.

CIEKAWOSTKA

W naszym programie odliczającym w dół dobrym pomysłem byłoby opóźnienie wyświetlania kolejnych liczb np. o jedną sekundę. Jednak w JS jest to dość skomplikowane i nie będziemy tutaj o tym pisać. Znacznie prościej uzyskać ten efekt np. w Pythonie, po zaimportowaniu biblioteki `time` i użyciu pochodzącej z niej funkcji `sleep`. Niezbędny kod w tym języku ma taką postać:

```
import time
licz = 10
while licz >= 0:
    time.sleep(1)
    print(licz)
    licz -= 1
print("Koniec odliczania")
```

Warto zauważyć, że w Pythonie nie stosujemy średników ani nawiasów klamrowych; początek pętli wskazuje dwukropek, a ciało pętli wyznaczane jest wcięciami. Ostatnie polecenie, wyświetlające komunikat „Koniec odliczania", znajduje się już poza pętlą.

2.5.5. Pętla do ... while

Pętla `do ... while` różni się od `while` tylko tym, że *najpierw* wykonuje polecenia zawarte w swym ciele, a dopiero potem sprawdza warunek wykonania pętli i jeśli jest on prawdziwy, wykonuje je ponownie, aż do momentu, gdy warunek stanie się fałszywy. W związku z tym polecenia zostaną wykonane przynajmniej raz, niezależnie od wartości warunku. Tymczasem kod pętli `while` nie wykona się ani razu, jeśli warunek w niej ustawiony będzie fałszywy.

Oto składnia polecenia `do … while`:

```
do {
    // kod do wykonania
}
while (warunek)
```

Przykład 2.26

Na listingu 2.52 przedstawiono zmodyfikowany program do odliczania. Tym razem ma on wyświetlać w pętli komunikat „Odliczam”. Dla porównania w programie użyto zarówno pętli `do … while`, jak i `while`. W obu przypadkach warunek początkowy jest fałszywy. W obu pętlach wpisano także komunikat, który pomoże w identyfikacji wykonanego kodu.

Listing 2.52

Różnica między do … while i while

```
let licz = 10;
// pętla do ... while
do {
    console.log("Pętla do ... while")
    console.log("Odliczam");
    licz--;
}
while (licz <= 0)

// pętla while
while (licz <= 0) {
    console.log("Pętla while")
    console.log("Odliczam");
    licz--;
}
```

Wynik wykonania tego kodu jest następujący:

```
Pętla do ... while
Odliczam
```

Jak widać, kod pętli `while` nie został wykonany, ponieważ najpierw interpreter sprawdził warunek i po określeniu, że jest on fałszywy, w ogóle nie wchodził do ciała pętli. Inaczej było w przypadku pętli `do … while`, która wykonała się raz, ponieważ, jak to zostało wcześniej wyjaśnione, w tej pętli kod wykonywany jest raz, na samym początku, dopiero po tym sprawdzany jest warunek, a kolejne wykonania poleceń uzależnione są od jego wartości.

2.6. Wprowadzenie do programowania obiektowego

W programowaniu da się wyróżnić kilka tak zwanych paradygmatów, czyli sposobów organizacji i tworzenia kodu. Do szczególnie często stosowanych należą:

- programowanie proceduralne,
- programowanie strukturalne,
- programowanie funkcyjne,
- programowanie obiektowe.

Wszystkie te paradygmaty mogą być stosowane łącznie.

Zgodnie z **paradygmatem proceduralnym** należy dzielić kod na procedury — fragmenty wykonujące ściśle określone operacje.

Według założeń **programowania strukturalnego** kod powinien być organizowany w hierarchicznie ułożone bloki, z jednym punktem wejścia i jednym lub wieloma punktami wyjścia — jak jest np. w przypadku bloku instrukcji `if … else if … else`.

W **programowaniu funkcyjnym** kładzie się nacisk na funkcje rozumiane na wzór funkcji matematycznych — każda taka funkcja dla danych wartości argumentów zwraca zawsze tę samą wartość, np. 2+2 zawsze będzie równe 4, bez względu na jakiekolwiek warunki wstępne.

Ogromne możliwości daje stosowane dziś niemal powszechnie **programowanie obiektowe**, oparte na obiektach będących abstrakcyjnym odwzorowaniem elementów świata rzeczywistego. Każda rzecz może być opisana jak klasa, będąca swoistym schematem czy „przepisem" na przedmiot, który ma być utworzony na jej podstawie. Klasa to po prostu byt z właściwościami i zachowaniami (działaniami), coś, o czym możemy powiedzieć, jak wygląda oraz jak się zachowuje. Na podstawie tego schematu można tworzyć wiele konkretnych obiektów, które mogą różnić się konkretną właściwością, jednak każdy musi posiadać wszystkie właściwości zawarte w definicji klasy. Samochód może np. mieć silnik benzynowy albo na olej napędowy, ale nie może *nie mieć* silnika. Zwierzę w schronisku może być psem albo kotem, ale musi być jakimś rodzajem zwierzęcia.

Każda klasa może mieć ponadto, jak już wspomniano, zdefiniowane pewne zachowania lub czynności. Pies szczeka, kot miauczy, papuga lata. O ile trzecie działanie nie ma nic wspólnego z pierwszymi dwoma, to szczekanie i miauczenie mogą być opisane jako różne rodzaje (zależne od konkretnego egzemplarza klasy zwierzę) jednej czynności: wydawania głosu.

Po utworzeniu konkretnego obiektu na podstawie jego definicji można odwoływać się do każdej z jego właściwości czy dowolnego zachowania właśnie poprzez ten obiekt.

W dalszej części tego rozdziału przedstawimy (w bardzo dużym skrócie) podstawy programowania obiektowego.

Przykład 2.27

Spróbujmy opisać obiektowo telefon komórkowy.

Najpierw tworzymy pustą definicję klasy (listing 2.53).

Listing 2.53

Początek tworzenia pustej definicji klasy Phone

```
class Phone {

}
```

Definiowanie klasy rozpoczynamy od użycia słowa kluczowego `class`. Nazwę klasy rozpoczynamy od wielkiej litery. Następnie określamy właściwości i metody klasy. Wszystkie te właściwości i metody podajemy zamknięte w nawiasach klamrowych.

Zdefiniowaliśmy klasę telefonu bez żadnych właściwości. Dodajmy mu kolor i markę (listing 2.54).

Listing 2.54

Dodanie właściwości do klasy

```
class Phone {
    brand = "Samsung";
    color = 'black';
}
```

Dodanie własności do klasy realizujemy poprzez wpisanie (wewnątrz nawiasów klamrowych) nazw własności, a potem, po znaku równości, przypisanych im wartości. W naszym przypadku utworzyliśmy własność `brand` i przypisaliśmy jej domyślną wartość `"Samsung"` (ponieważ jest to łańcuch znakowy, musi być zapisany w cudzysłowie. Gdybyśmy chcieli zdefiniować własność określającą cenę telefonu, nie wpisywalibyśmy cudzysłowu, lecz samą liczbę, jak widać na listingu 2.55). Druga własność, `color`, informuje o kolorze telefonu.

Listing 2.55

Dodanie kolejnych własności klasy

```
class Phone {
    brand = "Samsung";
    color = 'black';
    price = 123.21;
    usbC = false;
}
```

W wielu przypadkach najpierw definiuje się szkielet klasy, nie podając konkretnych właściwości (takich jak `Samsung` czy `black` z powyższego listingu). Właściwości te mogą zostać określone później, podczas tworzenia obiektu. W tym przypadku ustawiliśmy `Samsung` i `black` jako wartości domyślne. Oczywiście podczas tworzenia nowych obiektów własności te mogą być modyfikowane.

Jeśli chcemy wypisać obecną wartość klasy (lub odwołać się do niej w dowolny inny sposób), najpierw musimy zainicjować zmienną nowo utworzonym obiektem typu `Phone`. Zmienną obiektową inicjujemy, wpisując jej nazwę, znak równości, a po nim słowo kluczowe `new` i nazwę klasy, której instancję chcemy utworzyć.

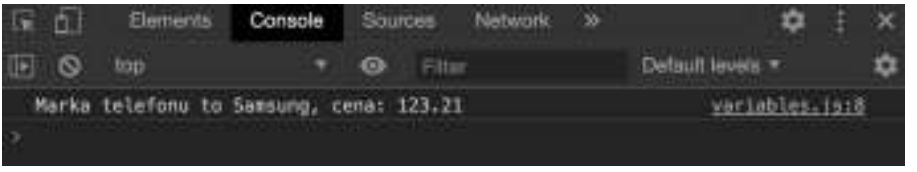
W celu uzyskania właściwości obiektu, który jest przypisany do zmiennej `phone`, należy wpisać nazwę zmiennej, a po niej, oddzieloną kropką, nazwę własności (listing 2.56).

Listing 2.56

Wypisanie wartości własności obiektu

```
let phone = new Phone();
console.log("Marka telefonu to " + phone.brand + ", cena: " + phone.price);
```

Oto efekt (rysunek 2.41):



Rysunek 2.41. Własności obiektu wypisane w konsoli

Jak już wspomniano, klasy mogą mieć nie tylko własności (cechy), ale też zachowania czy czynności, które wykonują. Telefon np. dzwoni lub wysyła SMS. Czynności w programowaniu nazywamy metodami. W JS tworzy się je za pomocą słowa kluczowego `function`.

Wiele funkcji, by wykonać jakąś czynność, musi pobrać parametr (jeden lub wiele). Telefon nie wyśle np. wiadomości, jeśli nie napiszemy jej treści i nie określimy odbiorcy. Pobierzmy zatem treść SMS-a (załóżmy, że odbiorca jest już zdefiniowany), prosząc użytkownika o jej podanie.

Na początku skupmy się na napisaniu metody `sendSms` (definiujemy ją wewnątrz definicji obiektu).

Oto przykładowa, pusta funkcja (listing 2.57):

Listing 2.57

Pusta funkcja

```
function() {

}
```

Co „robi" tak zdefiniowana funkcja? Nic. Nie ma ciała (to część wpisywana pomiędzy nawiasami klamrowymi), które określałoby wykonywane w niej czynności. Nie ma nawet nazwy, dzięki której moglibyśmy ją wywołać. W tej chwili nie jest zatem specjalnie przydatna.

Metodę dodajemy do klasy tak samo jak właściwość, co pokazano na listingu 2.58.

Listing 2.58

Dodanie do obiektu definicji metody

```
class Phone {
    brand = "Samsung";
    color = 'black';
    price = 123.21;
    usbC = false;
    sendSms = function(text) {
    }
}
```

Oczywiście ta metoda, choć ma już nazwę (`sendSms`), także nie robi nic. Krok po kroku temu zaradzimy. Teraz dodamy do niej kod, który przypisze parametr `text` do właściwości `sms` (listing 2.59).

Listing 2.59

Przekazanie parametru do funkcji

```
1 class Phone {
2     brand = "Samsung";
3     color = 'black';
```

```
4      price = 123.21;
5      usbC = false;
6      sendSms = function(text) {
7          this.sms = text;
8      }
9  }
```

W linijce 6. dodaliśmy pomiędzy dwoma okrągłymi nawiasami parametr.

W linijce 7. poinformowaliśmy, że do tego obiektu (`this` — to ważne słowo kluczowe oznacza bieżący obiekt), czyli do telefonu (`phone`), do właściwości `sms` (tak, w tym samym miejscu utworzyliśmy właściwość `sms`) przypisujemy wpisaną przez użytkownika wartość.

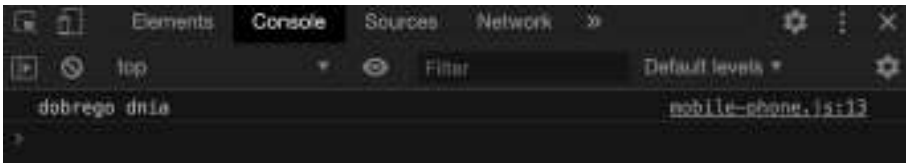
Wywołanie tej funkcji wygląda tak samo jak odwołanie się do własności: najpierw piszemy nazwę zmiennej obiektowej, która jest typu naszego obiektu, potem wpisujemy kropkę, po niej podajemy nazwę funkcji. W nawiasach wpisujemy z kolei parametr (tutaj treść wiadomości — listing 2.60).

Listing 2.60

Wywołanie funkcji

```
let phone = new Phone();
phone.sendSms("dobrego dnia");
console.log(phone.sms);
```

To pozwoli uzyskać (tu wyświetlony w konsoli) komunikat (rysunek 2.42):



Rysunek 2.42. Wyświetlenie treści wiadomości

Dodajmy jeszcze informację, od kogo jest SMS (listing 2.61).

Listing 2.61

Klasa z uzupełnioną metodą sendSms i dodanym atrybutem user

```
class Phone {
    brand = "Samsung";
    color = 'black';
    price = 123.21;
```

```
        usbC = false;
        user = "Marcin";
        sendSms = function(text) {
            this.sms = this.user + " wysłał sms: " + text;
        }
    }
```

Dodaliśmy kolejny atrybut (`user`) i wykorzystaliśmy tę informację w metodzie `sendSms` poprzez odwołanie się do niej za pomocą konstrukcji `this.user`.

UWAGA

Ten podrozdział nie jest prosty. Jego zrozumienie wymaga sporego skupienia i dokładnego przemyślenia jego treści. Najlepiej przeczytać go co najmniej dwa razy.

2.7. Kolekcje

Po zgłębieniu tematu tablic oraz programowania obiektowego warto zaznajomić się z kolekcjami. Można powiedzieć, że kolekcja to „opakowana" w funkcjonalne metody tablica. Może natomiast mieć własne metody, ponieważ w rzeczywistości jest obiektem.

UWAGA

Wykorzystanie kolekcji ma sens w przypadku dużych zbiorów danych.

Wyróżniamy dwa podstawowe typy kolekcji: zbiór i listę.

2.7.1. Set

Set to zbiór elementów niepoukładanych. Jest bardzo podobny do tablicy, jednakże wszystkie jego elementy są unikatowe (nie mogą się powtórzyć), co często bardzo się przydaje. Próba dodania do setu elementu, który jest już w nim zawarty, nie spowoduje błędu. Po prostu nowy element nie zostanie dodany, jeśli miałby zdublować już istniejący.

Zobaczmy, jak zadeklarować kolekcję typu `Set` (listing 2.62).

Listing 2.62

Deklaracja kolekcji typu Set

```
let clothes = new Set();
```

UWAGA

Zwróć uwagę na konieczność użycia słowa kluczowego `new`, świadczącego o tym, że zbiór jest **klasą**

Tak dodajemy elementy do zbioru (listing 2.63):

Listing 2.63

Dodawanie elementów do zbioru

```
clothes.add('sweter');
clothes.add('jeans');
```

Teraz czas pokazać potęgę kolekcji. W celu sprawdzenia, czy dany element występuje w tablicy, trzeba przeiterować po niej (przejść po wszystkich jej elementach). W kolekcjach oczywiście dzieje się to samo. Niemniej w przypadku tablic musimy sami pisać implementację iteracji (np. z wykorzystaniem pętli `for`), tym razem zaś możemy użyć gotowej metody `has`, która zwraca wartość `true`, jeśli szukany element znajduje się w kolekcji, `false` w przeciwnym wypadku. Zatem wywołanie z listingu 2.64:

Listing 2.64

Sprawdzenie, czy zbiór zawiera określony element

```
clothes.has('jeans'); // true
```

zwróci wartość `true`.

Usuwanie elementu z tablicy również było dość skomplikowane; tutaj zastąpiono to metodą `delete`, pokazaną na listingu 2.65:

Listing 2.65

Usuwanie elementu zbioru

```
clothes.delete('sweter');
```

Wielkość kolekcji uzyskamy, stosując właściwość `size` (listing 2.66).

Listing 2.66

Pobranie informacji o wielkości zbioru

```
clothes.size; // 2
```

To oczywiście tylko przykłady metod związanych z typem `Set`. Aby zapoznać się z innymi, warto zajrzeć do oficjalnej dokumentacji tego typu.

2.7.2. List

Innym rodzajem kolekcji, ale używanym wtedy, gdy zależy nam na tym, by elementy mogły się powtarzać, jest lista.

Lista to poukładana kolekcja tworzona w ten sposób, że do elementu początkowego można się odnieść, dodając kolejny element.

W przeciwieństwie do zbioru, w języku JavaScript nie ma wbudowanego typu dla tego rodzaju kolekcji. Lista jest dostępna jako część wielu zewnętrznych bibliotek.

Przykład 2.28

Działanie listy najprościej sobie wyobrazić na przykładzie pociągu z wagonami. Na początku jest lokomotywa, po niej 4 wagony. Gdy chcemy dodać wagon restauracyjny pomiędzy wagonami 2. i 3., to należy odpiąć wagon 2. od 3., następnie dopiąć wagon restauracyjny do wagonu 2., a potem do 3. Miejsce dopinania nazywane jest węzłem. W przypadku gdy chcemy usunąć jakiś wagon, procedura jest analogiczna. Możemy przejść po wszystkich elementach kolekcji z wykorzystaniem wiedzy o zależnościach między wagonami (wiemy, który wagon jest po wagonie 1. i że wagon 1. jest doczepiony do lokomotywy).

Skoro potrzebujemy aż tylu operacji, to do czego jest nam potrzebna lista?

Często chcemy mieć kontrolę nad tym, gdzie dokładnie wstawiamy nowy element — dodanie go na końcu tablicy czy zbioru nas nie zadowala. Właśnie w takich przypadkach nieoceniona jest lista, pozwalająca umieszczać nowe elementy w wybranym miejscu kolekcji na podstawie indeksów elementów.

Podstawowe kolekcje zostały przez nas omówione, jednakże w przypadku bardziej złożonych problemów warto zaznajomić się z pozostałymi, np. mapą (`Map`).

2.7.3. Obiekt iterator

Funkcjonalność podobną do listy wykorzystuje obiekt `iterator`. Implementację wyświetlenia elementów tablicy za jego pomocą, bez wykorzystania pętli `for`, pokazano na listingu 2.67. W pierwszym wierszu utworzyliśmy tablicę `blocks`, po której iterujemy z wykorzystaniem iteratora. Obiekt ten można wykorzystać dopiero wtedy, gdy jawnie wskażemy, że go utworzyliśmy, stosując do tego konstrukcję `Symbol.iterator`. Iterator za pomocą swojej metody `next` sprawdza, czy istnieje następny element zbioru.

Listing 2.67

Wykorzystanie obiektu iterator do iterowania po tablicy

```
const blocks = ["Lego", "Cobi"];

const iterator = blocks[Symbol.iterator]();

console.log(iterator.next()); // {value: "Lego", done: false}
```

```
console.log(iterator.next()); // {value: "Cobi", done: false}
console.log(iterator.next()); // {value: undefined, done: true}
```

Tak oto zakończyliśmy długą przygodę z poznawaniem podstawowych elementów programowania. Jest to spora dawka wiedzy, a jej przyswojenie na pewno nie jest łatwe. Niemniej jest to etap, przez który trzeba przejść (najlepiej kilkakrotnie), by móc w przyszłości tworzyć dobrze działające i przydatne aplikacje. Po jakimś czasie zaś, kiedy nabierze się wprawy, pisanie programów stanie się łatwe, intuicyjne i przyjemne.

2.8. Zadania

W tym podrozdziale znajdziesz zadania do całego rozdziału. Poprosimy Cię o napisanie wielu prostych programów. Wszystkie wiadomości potrzebne do rozwiązania zadań znajdziesz w treści tego rozdziału. Jeśli musisz je sobie przypomnieć, przeczytaj odpowiednią sekcję jeszcze raz.

Zadanie 2.1

Utwórz własną listę zakupów zawierającą ceny produktów potrzebnych do ugotowania obiadu. Niech w koszyku zakupowym znajdą się warzywa oraz mięso. Po dodaniu kilku produktów oblicz ich cenę uwzględniającą podatek.

Zadanie 2.2

Policz łączny wzrost członków Twojej najbliższej rodziny. Utwórz tyle zmiennych, ilu masz najbliższych, przypisz do zmiennych wzrost poszczególnych osób w centymetrach. Na koniec zsumuj wartości przypisane do zmiennych.

Zadanie 2.3

Zdeklaruj zmienną `piosenka` i przypisz do niej tekst zwrotki ulubionej piosenki, a następnie wyświetl go za pomocą tej zmiennej w konsoli.

Zadanie 2.4

Zdeklaruj kilka zmiennych, przypisz do nich tylko pojedyncze wersy piosenki z poprzedniego zadania, a następnie przy użyciu tych zmiennych wyświetl w konsoli całą piosenkę.

Zadanie 2.5

Napisz aplikację, w której pobierzesz od użytkownika jego imię oraz informację o jego ulubionym kolorze. Następnie wyświetl w konsoli komunikat: „Ulubiony kolor użytkownika (…) to (…)”. W miejscu wielokropków powinny zostać wyświetlone wprowadzone wcześniej dane.

Zadanie 2.6

Napisz aplikację, w której użytkownik poda kwotę brutto, a w konsoli zwrócona zostanie kwota netto.

Zadanie 2.7

Utwórz tablicę z popularnymi markami obuwia i wypisz rozmiar tej tablicy oraz jej przedostatni element.

Zadanie 2.8

Utwórz kolekcję marek laptopów zawierającą więcej niż sześć marek, po czym wypisz elementy drugi i czwarty.

Zadanie 2.9

Poproś użytkownika o wpisanie dowolnej liczby parzystej. Jeśli wpisze on liczbę nieparzystą, wyświetl alert z komunikatem: „Wpisana liczba jest nieparzysta".

Zadanie 2.10

Użytkownik proszony jest o wskazanie numeru budynku na ulicy Kolskiej. Z bazy pocztowej wiemy, że na ulicy Kolskiej znajdują się budynki wyłącznie z numerami pomiędzy 10 a 15. Aplikacja powinna wyświetlić alert w przypadku podania numeru budynku spoza tego zakresu.

Zadanie 2.11

Utwórz tablicę zawierającą nazwy produktów. Użytkownik powinien móc wpisać numer i uzyskać informację o produkcie zapisanym w tablicy pod indeksem odpowiadającym wpisanemu numerowi. Jeśli poda numer wykraczający poza zakres tablicy, to powinien zostać wyświetlony odpowiedni komunikat.

Zadanie 2.12

Wszyscy znamy kolejność planet w Układzie Słonecznym. Napisz aplikację, która po wpisaniu przez użytkownika numeru planety wyświetli jej nazwę. Jeśli użytkownik wpisze liczbę większą niż 8, powinien wyświetlić się alert z informacją, że Układ Słoneczny ma tylko 8 planet.

Zadanie 2.13

W ogrodzie botanicznym jest cennik biletów:

- normalny — 3 zł
- ulgowy — 1,5 zł

Napisz aplikację, w której użytkownik podaje liczbę biletów normalnych oraz ulgowych, po czym uzyskuje informację o kwocie do zapłaty.

Zadanie 2.14

Załóżmy, że dzień rozpoczyna się wschodem słońca o 6, a kończy zachodem o 21. Napisz aplikację, która po wpisaniu przez użytkownika pełnej godziny wyświetli informację o porze dnia. Przyjmijmy, że ranek trwa do 11, godzina 12 to południe, od 13 do 18 mamy popołudnie, od 18 do 21 wieczór, a między godzinami 21 i 6 noc.

Zadanie 2.15

Stwórz prostą aplikację pogodową, która, w zależności od wyświetlonych poniżej warunków, będzie wyświetlała odpowiadające im komunikaty:

- Warunek: „Jest słonecznie i pochmurnie”; komunikat: „Idę pobiegać”.
- Warunek: „Jest słonecznie i powyżej 25 stopni”; komunikat: „Idę się opalać”.
- Warunek: „Pada”; komunikat: „Zostaję w domu i czytam książkę”.

Zadanie 2.16

Napisz aplikację typu quiz. Opracuj trzy pytania i odpowiedzi na nie. Pytania powinny wyświetlać się kolejno użytkownikowi, a on powinien wprowadzać odpowiedzi. Na końcu, zależnie od liczby trafień, powinien otrzymać informację o uzyskanej randze:

- Czarnoksiężnik — gdy odpowiedział na wszystkie pytania;
- Mistrz — gdy odpowiedział na dwa pytania;
- Adept — gdy odpowiedział na jedno pytanie.

Zadanie 2.17

W sklepie zoologicznym sprzedawca potrzebuje programu, który poda klientowi, ile karmy należy odmierzyć dla kota na podstawie jego wagi, zgodnie z przelicznikiem podanym na opakowaniu.

cat (kg)	1	2	3	4	5	6
daily amount (g)	20	31	41	50	58	66

Napisz taki kalkulator, który obliczy potrzebną ilość karmy oraz wyświetli odpowiednią informację w konsoli.

Zadanie 2.18

Napisz kalkulator do obliczenia, czy z góry znane wagi kilku elementów nie przekraczają wskazanej przez użytkownika ładowności ciężarówki.

Zadanie 2.19

Stwórz quiz (podobny do tego z zadania 2.16) przy użyciu tablicy dwuwymiarowej. Quiz ma zawierać 10 pytań z odpowiedziami. Po udzieleniu odpowiedzi użytkownik powinien uzyskać informację rankingową:

- 0 – 4 poprawnych odpowiedzi — Spróbuj jeszcze raz;
- 5 – 8 poprawnych odpowiedzi — Całkiem nieźle;
- 8 – 10 poprawnych odpowiedzi — Jesteś mistrzem!

Zadanie 2.20

Utwórz tablicę zawierającą pięć wybranych stolic europejskich. Używając pętli `for ... of`, wypisz wszystkie stolice zawarte w tablicy.

Zadanie 2.21

Użyj pętli `for` do zsumowania co trzeciego elementu tablicy `liczby = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`.

UWAGA

W poniższych zadaniach wykorzystaj zdobytą wiedzę o obiektach.

Zadanie 2.22

Stwórz aplikację, która pozwoli uzupełnić dane klienta (znamy już imię i nazwisko) o wiek i płeć.

Zadanie 2.23

Napisz aplikację do obsługi stoiska warzywnego, dzięki której właściciel podaje informacje o cenach swoich towarów (załóżmy, że znamy już nazwę firmy).

Zadanie 2.24

Utwórz klasę opisującą windę w bloku. Utwórz parametry określające, ile osób może wejść do windy oraz jej wymiary. Dodaj metodę do wskazania, na które piętro winda ma wjechać (użytkownik podaje numer piętra).

Zadanie 2.25

Stwórz aplikację wyświetlającą informacje na temat Lublina: liczbę mieszkańców, współrzędne geograficzne oraz powierzchnię. Wykorzystaj konstrukcję łańcuchów szablonowych.

3

Projektowanie aplikacji

Sama umiejętność poprawnego napisania kodu podczas tworzenia aplikacji nie jest wystarczająca. Dzisiaj wymagane jest, by programy były wydajne, testowalne i proste w utrzymaniu.

Projektowanie aplikacji to proces, który na ogół trwa wiele miesięcy i którego celem jest uproszczenie pracy wszystkich członków zespołu. Dobrze zaprojektowana aplikacja to taka, która jest łatwa w utrzymaniu i rozbudowie.

Jedną z ważniejszych ról w zespole tworzącym oprogramowanie odgrywa dzisiaj osoba UX designera. Odpowiada ona za poprawę odbioru aplikacji przez użytkownika końcowego. Do jej zadań należy m.in.:

- wskazanie preferencji grupy docelowej;
- dobranie prawidłowej wielkości elementów aplikacji zależnie od urządzenia, na którym będą wyświetlane;
- zaprojektowanie makiet, które zaprezentują rozmieszczenie elementów na ekranach aplikacji.

UX designer, opierając się na znajomości profilu docelowego użytkownika, przekazuje nam informacje dotyczące tego, jak ma wyglądać makieta aplikacji. Jeśli nie mamy w zespole projektanta, możemy uzyskać takie dane od klienta. Jednak nawet jeśli wymagania co do wyglądu będą jasno sprecyzowane, może to nie wystarczyć, by zadowolić użytkownika. Przez pierwszych kilka chwil zostanie z nami i będzie używał naszej aplikacji, ale jeśli nie będzie ona wydajna (np. strona czy jakiś jej element będą się ładować za długo), dość szybko straci nią zainteresowanie.

Rozwijanie aplikacji od strony wizualnej jest bardzo ważne, ale w tym rozdziale skrótowo omówiony zostanie sposób wykonania wydajnej i prostej w utrzymaniu aplikacji w kontekście programisty.

3.1. Dobre praktyki związane z programowaniem obiektowym

Ważne jest, by kod był jak najbardziej spójny. Spełnienie tego wymogu ułatwia wykorzystywanie wspomnianych w poprzednim rozdziale obiektów wraz z jego metodami oraz funkcji.

Dla przypomnienia: funkcje są to luźne, wydzielone instrukcje zamknięte w ramach bloku. Dzięki temu ten sam ciąg instrukcji może być wykonany wielokrotnie, bez potrzeby powielania (kopiowania) tego samego kodu. Metody, podobnie jak funkcje, są to wydzielone kawałki kodu opakowanego w blok, jednak umieszczone w ramach danej klasy. Pomaga to w powiązaniu danego zestawu właściwości klasy oraz operacji, jakie taka klasa może wykonać.

Poniżej zaprezentowane zostanie użycie funkcji (przykłady 3.1 i 3.2) oraz metod (przykład 3.3).

Przykład 3.1

Na listingu 3.1 zdefiniowaliśmy funkcję `maximum`, która przyjmuje jako argument tablicę liczb i zwraca największy spośród jej elementów. Najpierw zadeklarowaliśmy zmienne tablicowe, na których będzie działać metoda. Następnie podaliśmy definicję metody. Jej wywołanie powinno wystąpić na końcu.

UWAGA

Prezentowane tu kody znaleźć można w serwisie GitHub, pod adresem: *http://github.com/weronikakortas/book*.

Listing 3.1

Definicja i wywołanie metody do wyznaczenia największego elementu tablicy

```
// deklaracja zmiennych tablicowych
const numbers = [11, 17, 13];
const numbersDouble = [3.0, 1.7, 5.1];
// definicja metody maximum
function maximum(array) {
    let max = array[0];
    for (let i = 1; i < array.length; i++) {
        var arrayElement = array[i];
        if (max < arrayElement) {
            max = arrayElement;
        }
    }
```

```
    return max;
}
// wywołanie metody maximum na tablicy numbers
maximum(numbers);
console.log(maximum(numbers));
```

Gdy wywołamy `maximum` bez podania argumentu, w konsoli wyświetlony zostanie błąd. Jeśli przekażemy jej — jak zrobiono w tym kodzie — tablicę liczb całkowitych (`numbers`), to wyświetlona zostanie największa z nich, 17. Gdyby przekazanym metodzie argumentem była tablica liczb zmiennoprzecinkowych (`numbersDouble`), wyświetlona zostałaby wartość 5,1.

Wynik działania metody możemy przypisać do zmiennej. W takim przypadku zamiast posługiwać się w ostatniej linii (w metodzie `console.log`) wywołaniem metody `maximum(numbers)` — jest to w gruncie rzeczy zdublowanie kodu — moglibyśmy przekazać do wyświetlenia sam wynik wywołania. Ostatnie trzy linijki kodu mogłyby więc wyglądać tak jak na listingu 3.2.

Listing 3.2

Wynik wywołania metody można przypisać do zmiennej

```
// wywołanie metody maximum na tablicy numbers
const score = maximum(numbers);
console.log(score);
```

Przykład 3.2

Przeprowadźmy teraz sumowanie maksymalnych elementów pobranych z dwóch tablic, przy założeniu, że jedna tablica składa się z liczb całkowitych, a druga ze zmiennoprzecinkowych (listing 3.3).

Listing 3.3

Sumowanie największych elementów tablic

```
function sumOfMaxFromArrays(array, array2) {
    const maxNumber = maximum(array);
    const maxDouble = maximum(array2);

    return maxNumber + maxDouble;
}
console.log(sumOfMaxFromArrays(numbers, numbersDouble));
```

Warto zauważyć, że w ciele metody `sumOfMaxFromArrays` następuje wywołanie funkcji `maximum` dla obu tablic, a wyniki tych wywołań zostają przypisane do zmiennych (`maxNumber` i `maxDouble`). Wykorzystując tę wcześniej utworzoną metodę i zmienne

przechowujące wyniki jej wykonania, można nie duplikować kodu do wyznaczenia maksymalnego elementu z każdej tablicy.

Wynik sumowania od razu został wypisany na konsoli. Gdybyśmy jednak potrzebowali skorzystać z niego w innych miejscach kodu, lepiej byłoby przypisać go do zmiennej i odwoływać się do niego za jej pomocą.

W przypadku obiektów do odniesienia się do zmiennych obiektowych używanych w metodach wykorzystujemy słowo kluczowe `this`. Jego użycie spowoduje, że w danym obiekcie użyta zostanie właściwość danego obiektu. Takie odwołanie nazywamy kontekstem wywołania.

Przykład 3.3

Utwórzmy uproszczoną klasę odwzorowującą pokój z jego wybranymi właściwościami (długością i szerokością), wśród których wymienione zostaną także niektóre znajdujące się w nim elementy (w naszym przykładzie określimy, czy w pokoju znajduje się biurko). Zdefiniujemy także dla obiektu metodę, która wyświetli informacje o pokoju (listing 3.4).

Listing 3.4

Przykładowa implementacja klasy Room

```
class Room {
    width = 200;
    length = 220;
    desk = false;
    space = function () {
        return this.width * this.length;
    }
    display = function () {
        console.log(`szerokość ${this.width}, długość ${this.length},
powierzchnia ${this.space()}, czy ma biurko ${this.desk}`);
    }
}

const room = new Room();
console.log(room.display());
```

UWAGA

Poprzez słowo kluczowe `this` możemy odwołać się zarówno do właściwości obiektu, jak i metody w nim utworzonej.

Używanie tak zdefiniowanych metod oraz świadomość możliwości odwołania się do zmiennych w klasie otwierają przed nami duże możliwości tworzenia własnych aplikacji. Jednakże zanim do tego przejdziemy, warto zapoznać się z kolejnymi dobrymi zasadami tworzenia aplikacji.

3.2. Clean code, czyli czysty kod

Każdy programista ma swoją własną listę ulubionych zasad, których pilnuje i które wdraża w życie.

DEFINICJA

Główną zasadą dla większości jest zasada **KISS** (akronim od ang. *Keep It Simple, Stupid* — nie komplikuj, głupcze), mówiąca o tym, w jaki sposób tworzyć kod, tak żeby nawet osoba postronna była w stanie zrozumieć go w maksymalnie określonym skończonym czasie, najlepiej na podstawie samego kodu, bez potrzeby czytania dokumentacji i zawartych w komentarzach opisów. Ułatwiają to dobrze nazwane zmienne i metody, jak również utrzymywanie porządku w kodzie.

Uzupełnieniem KISS jest **zasada DRY** (akronim od ang. *Don't Repeat Yourself* — nie powtarzaj się), mówiąca o tym, że kod praktycznie nigdy nie powinien być powielany. Dlaczego? W sytuacji, gdy ten sam kod zostanie powielony dziesięć razy w kilkunastu plikach, po czym okaże się, że zawiera on błędy, należy odnaleźć i poprawić wszystkie te miejsca, w których wystąpił. Sprawia to, że szybkość zarządzania kodem (jego modyfikacji czy poprawiania) jest odwrotnie proporcjonalna do szybkości, z jaką został wytworzony.

Jak najprościej uzyskać zgodność z DRY? Najwygodniej jest wydzielać logikę programu (zadania, jakie ma wykonywać) do odpowiednich funkcji, które definiujemy raz, ale możemy ich używać w wielu miejscach. Warto też dbać o to, by nie były one zbyt długie. Przyjmuje się, że jedna funkcja to tyle kodu, ile jesteśmy w stanie zobaczyć na połowie ekranu.

W tym miejscu należy wspomnieć o **YAGNI** (akronim od ang. *You Aren't Gonna Need It* — nie będziesz tego potrzebował). Ta zasada nakazuje, by pisać tylko wymagane przez klienta funkcjonalności, co oznacza, że nie należy tworzyć zbędnego kodu.

CIEKAWOSTKA

Zanim upowszechniła się filozofia KISS, można było spotkać tak nazywane zmienne (listing 3.5):

Listing 3.5

Przykład źle dobranych nazw dla zmiennych

```
1  let _ = 3;
2  let __ = 4;
3  let ___ = 5;
```

CIEKAWOSTKA cd.

```
4  let ____ = 6;
5
6  function number() {
7      if (_ === __) {
8        console.log("Porównanie liczb 3 i 4");
9      }
10     if (___ === ____) {
11        console.log("Porównanie liczb 5 i 6");
12    }
13 }
14 number();
```

O ile porównanie liczb w linijce 7. jest jeszcze dość proste do zrozumienia, o tyle linijka 10. jest już bardzo mało czytelna; w gruncie rzeczy nie wiadomo, które liczby porównujemy, trzeba się skupić, by to poprawnie zinterpretować. A sytuacja znacznie by się pogorszyła, gdyby zmienne zostały utworzone nieco dalej od miejsca, w którym ich używamy.

3.3. Dokumentowanie kodu

Jeden z najpopularniejszych sposobów dokumentowania kodu w przypadku języka JavaScript wiąże się z użyciem formatu JSDoc. Narzędzia do pracy z nim są dostępne w WebStorm od razu, więc nie trzeba niczego instalować.

UWAGA

JSDoc to nic innego jak komentarz z odpowiednimi tagami w nim.

Najczęściej używane tagi zestawiono w tabeli 3.1.

Tabela 3.1. Niektóre tagi dostępne w bibliotece JSDoc, pomocne w dokumentacji kodu

Tag	Składnia	Uwagi
`@description`	*opis* `@description`	Dodaje opis funkcji lub metody.
`@returns`	`@returns {`*typ*`}`	Określa typ zwracanej wartości.
`@param`	`@param {`*typ*`}` *opis parametru*	Określa parametry użyte w funkcji wraz z typem parametru i opisem.

Przykład 3.4

Utworzono (z zachowaniem zasad czystego kodu, np. z dbałością o odpowiednie nazwy) plik *sum.js*, który zawiera kod funkcji sumującej dwa elementy. Oto podstawowa implementacja tej metody:

```
function sum(first, second)
```

Jeśli w wywołaniu tej metody:

```
console.log(sum(2,3))
```

postawimy kursor przy słowie `sum` i wciśniemy klawisz *F1* (lub skrót klawiszowy *Ctrl+Q*), w tym przypadku odpowiedzialny za uzyskanie informacji z dokumentacji tej metody, zobaczymy jedynie definicję funkcji, bez jakiejkolwiek dodatkowej podpowiedzi, jak pokazano na rysunku 3.1.

Rysunek 3.1. Widok podpowiedzi (w WebStorm) dla funkcji sum

Wiemy na tej podstawie jedynie tyle, że istnieje metoda `sum` o parametrach `first` i `second`. Nie uzyskaliśmy informacji o tym, czy ta metoda coś zwraca. Czy `first` i `second` to liczby? Właściwie nie wiemy, co się zdarzy, gdy wywołamy metodę.

Kliknijmy tym razem w innym miejscu wywołania, po słowie `log`, i ponownie wciśnijmy *F1* (lub skrót klawiszowy *Ctrl+Q*). Efekt pokazano na rysunku 3.2.

Rysunek 3.2. Widok podpowiedzi (w WebStorm) dla funkcji log

To jest przykładowa, poprawnie napisana dokumentacja metody, zawierająca:

- opis metody,
- informację o typie zwracanej wartości (w tym przypadku jest to akurat metoda `void`, czyli taka, która nic nie zwraca).

Korzystając z wiadomości dotyczących tagów z tabeli 3.1, można utworzyć dokumentację naszej funkcji (listing 3.6):

Listing 3.6

Tworzenie dokumentacji funkcji sum

```
/**
 * @description return sum of two numbers
 * @param {Number} first
 * @param {Number} second
 * @returns {Number}
 */
```

Po dodaniu tego kodu i wciśnięciu klawisza *F1* (lub skrótu klawiszowego *Ctrl+Q*) wewnątrz wywołania funkcji `sum` tym razem uzyskamy bardziej kompletną podpowiedź (rysunek 3.3):

Rysunek 3.3. Widok podpowiedzi (w WebStorm) po dodaniu komentarzy tworzących dokumentację

Dokumentacja kodu ma służyć innym programistom, by zrozumieli, co autor miał na myśli, tworząc daną funkcję czy klasę. Dobrze udokumentowany kod jest lepiej oceniany (i rzeczywiście ma większą wartość, ponieważ programista, który będzie w przyszłości nad nim pracował, będzie mógł szybko odnaleźć interesujące go miejsce), a przede wszystkim usprawnia utrzymanie aplikacji już po jej wdrożeniu.

3.4. Algorytmy

Świadomość tego, że wydajność programu jest oparta na algorytmach, doceniamy dopiero wtedy, gdy piszemy którąś aplikację z rzędu. By pisać rzeczywiście wydajny kod, trzeba znać chociaż podstawowe algorytmy.

Podczas projektowania procesów można się posłużyć jednym z bardziej znanych sposobów rozwiązywania problemów. Następuje w nim kolejno:

1. Sformułowanie zadania.
2. Określenie wyniku.

3. Poszukiwanie metody rozwiązania (czyli właśnie algorytmu).
4. Przedstawienie algorytmu za pomocą wybranego schematu.
5. Analiza poprawności rozwiązania.
6. Testowanie rozwiązania.

W tym podrozdziale zostaną opisane punkty 4. i 5.

3.4.1. Czym jest algorytm?

Wyjaśnijmy sobie najpierw, co decyduje o wyborze takiego, a nie innego rozwiązania.

Na wybór metody sortowania decydujący wpływ na ogół ma czas sortowania (im krótszy, tym lepiej), a niekiedy liczba utworzonych zmiennych (także ich powinno być możliwie mało).

Architekci oprogramowania, wybierając takie czy inne API (interfejs, za pomocą którego aplikacje mogą ze sobą rozmawiać), decydują się na rozwiązanie, które najlepiej spełnia oczekiwania klienta.

UWAGA

Czym jest algorytm w pracy programisty? Jest to sposób postępowania pozwalający na jednoznaczne i jak najprostsze rozwiązanie problemu.

CIEKAWOSTKA

Był czas, że na rozmowach kwalifikacyjnych pojawiało się zadanie:

„Proszę o zaprojektowanie procesu mającego na celu uzyskanie mleka z lodówki".

Z tego względu na podstawie tego właśnie procesu w kolejnych podrozdziałach zostaną omówione rodzaje projektowania.

Wróćmy na chwilę do przedstawionej na początku tego podrozdziału listy. Mamy już wyraźnie nakreślone zadanie do wykonania. Wiemy, jaki jest oczekiwany wynik. Czas skupić się na poszukiwaniu metody rozwiązania — czyli algorytmu.

3.4.2. Projektowanie algorytmu

Jak zaprojektować algorytm do czegoś, co jest codzienną czynnością? Najlepiej zacząć od wymienienia czynności, które są wykonywane, a następnie zapisać je w postaci listy kolejnych kroków.

3.4.2.1. Zasada „dziel i zwyciężaj”

Najczęściej wykorzystywaną metodą projektowania algorytmów jest zasada „dziel i zwyciężaj”. Zgodnie z jej nazwą pierwszą rzeczą, jaką trzeba zrobić, jest *podzielenie* problemu na mniejsze, porównywalne pod względem skomplikowania, części — to za ich pomocą zostanie najpierw przedstawione rozwiązanie. Dopiero na końcowym etapie następuje scalenie tych małych kroków w jeden, gotowy algorytm.

Dlaczego ta zasada jest najbardziej popularna? Dlatego że algorytmy zbudowane z jej użyciem często mają najmniejszą złożoność obliczeniową.

Zasada ta zostanie wykorzystana w sortowaniu i wyszukiwaniu elementów.

3.4.2.2. Lista kroków

Ten rodzaj opisu procesu ma za zadanie przedstawienie sekwencji kroków. Stosowany jest głównie podczas opisu sytuacji ogólnych i zwykle nie opisuje różnych wariantów.

W tym przypadku nie ma dobrej odpowiedzi na pytanie o to, kiedy lista jest kompletna. Dlaczego? Dlatego że niemal zawsze, kiedy ktoś inny spojrzy na tę listę, podpowie krok, który pominęliśmy.

Przykład 3.5

Oto przykładowa lista kroków procesu mającego na celu wyciągnięcie mleka z lodówki:

1. Rozpocznij proces.
2. Zlokalizuj lodówkę.
3. Otwórz lodówkę.
4. Wyciągnij mleko.
5. Zamknij lodówkę.
6. Postaw mleko na stole.
7. Zakończ proces.

3.4.2.3. Pseudokod

Rozwiązanie problemu znalezienia mleka (a także dowolnego problemu programistycznego) można również przedstawić za pomocą konstrukcji pośredniej pomiędzy językiem naturalnym a kodem komputerowym. Tak opisaną strukturę nazywamy *pseudokodem*.

Przykład 3.6

Oto przykład pseudokodu ilustrującego rozwiązanie problemu wyjmowania mleka z lodówki.

```
begin
    znajdzLodowke()
    otworzLodowke()
```

```
    wyciagnijMleko()
    zamknijLodowke()
    postawMleko()
end
```

Jak widać, stosując pseudokod, zamieniamy opisywane słownie kroki na nazwy metod; jeśli są potrzebne parametry, na jakich mają działać metody, powinniśmy je dodać.

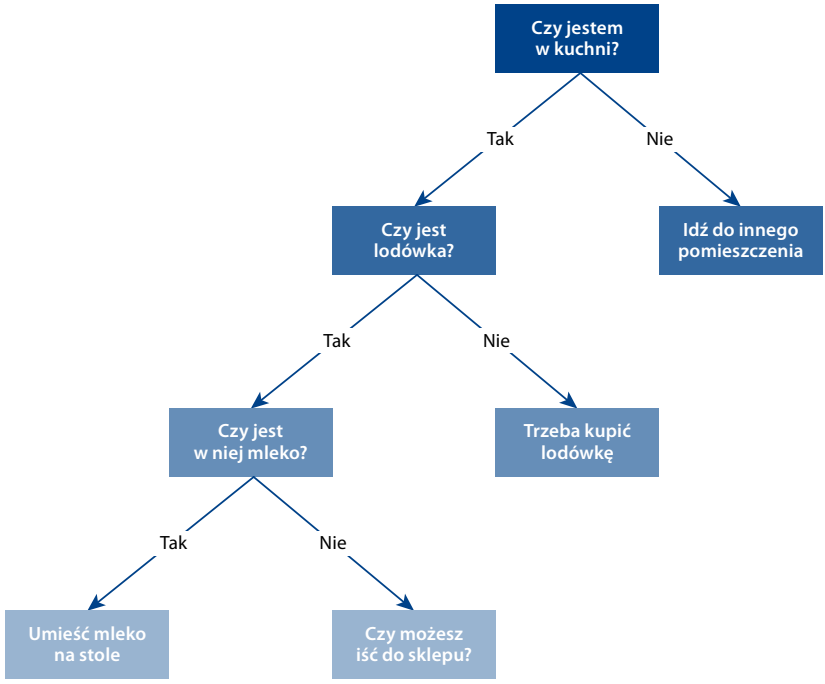
3.4.2.4. Drzewo decyzyjne

Najczęściej wybieranym graficznym elementem procesu decyzyjnego, stosowanym przy tzw. uczeniu maszynowym oraz w obliczaniu prawdopodobieństwa, jest drzewo decyzyjne. Jest to odwzorowanie wszystkich kroków decyzji za pomocą schematu przypominającego drzewo, patrząc od korzenia (przy czym korzeń występuje w tym przypadku u góry schematu). Każda gałąź to droga do decyzji, którą obrazuje liść.

Przykład 3.7

Proces znalezienia mleka można również przedstawić za pomocą drzewa decyzyjnego (rysunek 3.4).

Jak już wspomniano, drzewko jest odwrotnością rzeczywistego drzewa, czyli na samej górze jest korzeń, od którego przechodzimy wzdłuż gałęzi przez kolejne decyzje.



Rysunek 3.4. Drzewo decyzyjne

3.4.2.5. Schemat blokowy

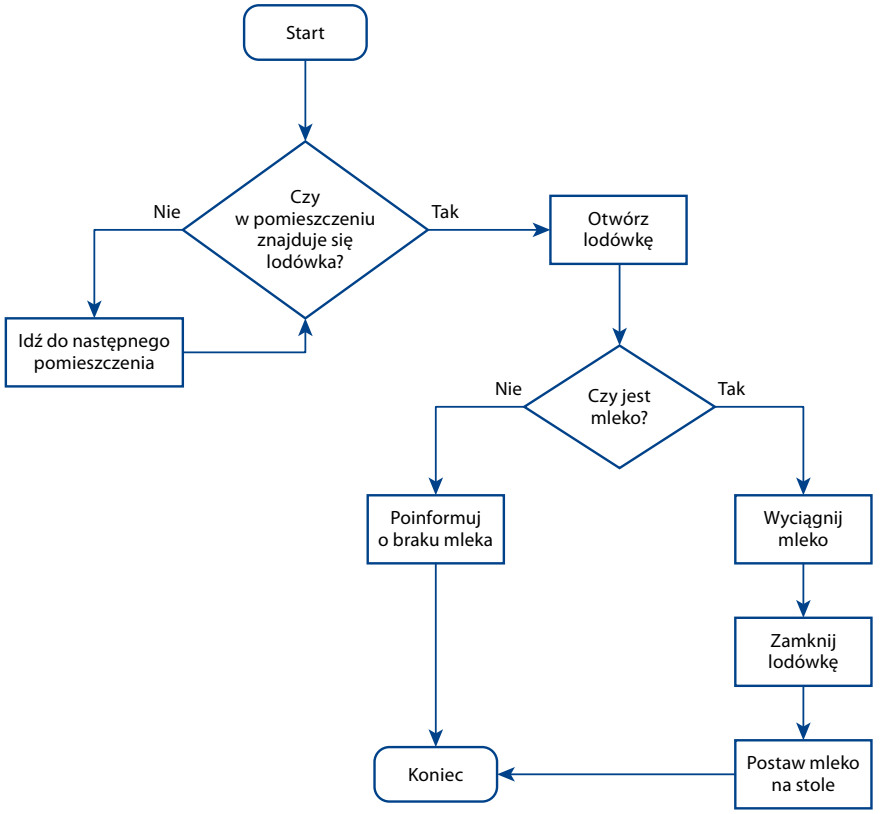
Taką listę czynności dobrze jest teraz zaprezentować za pomocą schematu blokowego.

DEFINICJA

Schemat blokowy to graficzne przedstawienie sekwencji czynności, zazwyczaj uwzględniające warianty, których wykonanie jest uzależnione od spełnienia różnorakich warunków.

Przykład 3.8

Oto sekwencja czynności koniecznych do wyciągnięcia mleka z lodówki zaprezentowana za pomocą schematu blokowego (rysunek 3.5).

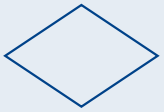


Rysunek 3.5. Przykładowy schemat blokowy

UWAGA

Na rysunku 3.5 posłużono się symbolami, których znaczenie wyjaśniono w tabeli 3.2.

Tabela 3.2. Znaczenie symboli użytych w schemacie blokowym

Symbol	Znaczenie
	Blok startu i zakończenia algorytmu
	Blok decyzyjny (czyli pytanie, na które odpowiedzią jest tak/nie)
	Blok operacyjny, określający działanie do podjęcia (np. wyświetlenie informacji)
	Blok wejścia — miejsce, w którym użytkownik musi podać jakąś informację
	Wskazanie kolejnego elementu w sekwencji

Jak widać, schemat blokowy różni się od listy kroków, gdyż jest od niego znacznie czytelniejszy dla klienta, który może mieć problemy ze zrozumieniem algorytmu zapisanego za pomocą listy kroków. Dodatkowym plusem takiego zapisu jest to, że w łatwy sposób można zobaczyć cały przepływ algorytmu i dzięki temu wyłapać potencjalne błędy.

W projekcie w rozdziale 7. wykorzystamy schemat blokowy do opisania algorytmu.

3.4.2.6. Złożoność obliczeniowa

Opracowaliśmy już pierwszy prosty algorytm. Wróćmy na chwilę do rozważań na ich temat.

Jak już wspomniano, najlepiej, gdy algorytmy są rozpisane na drobne kroki — to mocno przyspiesza proces implementacji.

Poza tym trzeba oszacować ich *złożoność obliczeniową*. Dzięki temu można wykazać, że np. na niektórych smartfonach dany algorytm zadziała szybciej niż na innych, bo ich procesory mają większą *moc obliczeniową*, co pozwala im poradzić sobie z większą liczbą operacji w tym samym czasie.

Czym jednak jest złożoność obliczeniowa?

DEFINICJA

Złożoność obliczeniowa to wspólny mianownik (część wspólna) dla algorytmów, utworzony w celu definiowania ich wydajności.

W trakcie badania złożoności tak naprawdę obliczana jest suma wykonywanych operacji:

f(n) = operacja_1 + operacja_2 + … + operacja_n.

Przykład 3.9

W celu rozjaśnienia tematu odwołajmy się do kolejnego przykładu. Wykorzystując wiedzę z poprzedniego rozdziału, napiszmy kod zliczający sumę elementów w tablicy. Następnie spróbujemy policzyć, ile operacji jest wykonywanych podczas liczenia sumy (listing 3.7).

Listing 3.7

Kod zliczający sumę elementów tablicy

```
1 function sumElements(arr) {
2     let sum = 0;
3     for (let i = 0; i < arr.length; i += 1) {
4         sum += arr[i];
5     }
6     return sum;
7 }
```

Znasz już (z rozdziału 2.) sposób tworzenia funkcji, więc ten kod powinien być znajomy. Niemniej przeanalizujmy jego kolejne wiersze. Ich zestawienie zawarto w tabeli 3.3. Dla każdej linii podano także liczbę wykonywanych operacji.

Tabela 3.3. Analiza wierszy kodu funkcji sum

Numer linii	Opis wykonywanych operacji	Liczba operacji
1	Deklaracja funkcji `sumElements`, która w parametrze przyjmuje tablicę liczb.	0
2	Deklaracja zmiennej `sum` i przypisanie do niej wartości początkowej — pierwsza operacja.	1
3	Inicjalizacja pętli, która przechodzi po elementach tablicy.	0
4	Dodanie do zmiennej `sum` wartości z tablicy pod indeksem `i` — ta operacja jest wykonywana tyle razy, ile jest elementów w tablicy. Przyjęło się zapisywać nieustaloną liczbę elementów jako *n*, dlatego ta instrukcja wykona się *n* razy.	*n*
6	Zwracamy sumę.	1

A zatem złożoność to $f(n) = 1 + n + 1 = n + 2$.

Najczęściej obliczanie złożoności obliczeniowej wykonuje się, gdy klient wymaga, by aplikacja była możliwie jak najbardziej wydajna. Wtedy każdy element i każdy krok mają znaczenie.

3.4.3. Rekurencja

W celu zrozumienia istoty algorytmów najpierw trzeba zrozumieć, czym jest rekurencja. Od czasu do czasu spotykamy się ze sformułowaniami typu: „Będę grał w tę grę tak długo, aż zdobędę wszystkie trofea". Mówiący to ma na myśli, że będzie wykonywał cały czas tę samą czynność, aż do spełnienia pewnego warunku.

DEFINICJA

Rekurencja bezpośrednia to wywołanie metody w niej samej.

Rekurencja pośrednia to występujące w łańcuchu funkcji wywoływanie funkcji zależnych od siebie (A wywołuje B, B wywołuje A itd.).

Rekurencja pośrednia jest wykorzystywana wówczas, gdy zależność pomiędzy dwoma czynnościami jest stała.

Przykład 3.10

Codziennie wstajesz rano do szkoły. Nastawiasz budzik, którego oprogramowanie zawiera funkcję drzemki. Kiedy drzemka zostaje wywołana i zakończona, wywołuje alarm budzenia na kolejny dzień — który, rzecz jasna, także może wywołać funkcję drzemki.

Przykład 3.11

Niektóre wieże audio mają w sobie zmieniarkę płyt CD. Można w nich zmienić losowo płytę i po odtworzeniu z niej losowej piosenki następuje kolejne losowe wybranie płyty. Odtwarzanie zostanie zakończone, gdy skończą się wszystkie piosenki na wszystkich płytach.

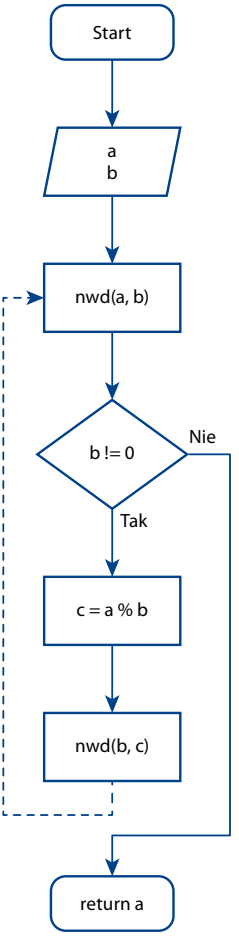
Rekurencja pośrednia najczęściej znajduje zastosowanie w matematyce. W dalszej części tego podrozdziału skupimy się na rekurencji bezpośredniej.

Przykład 3.12

Spróbujemy wyjaśnić rekurencję bezpośrednią na dobrze znanym przykładzie wyznaczania największego wspólnego dzielnika (NWD).

Przypomnijmy, że $NWD(4, 12) = 4$ (największy wspólny dzielnik liczb 4 i 12 to 4), a $NWD(16, 24) = 8$ (największy wspólny dzielnik liczb 16 i 24 to 8).

Warto tutaj sięgnąć do algorytmu Euklidesa. Na jego podstawie możemy zbudować schemat pokazany na rysunku 3.6:



Rysunek 3.6. Schemat blokowy algorytmu Euklidesa dotyczącego NWD

Jak można zauważyć, funkcja sprawdza, czy drugi parametr osiągnął wartość równą zeru, i jeżeli jej nie osiągnął, to oblicza resztę z dzielenia parametrów i wywołuje samą siebie z drugim parametrem i obliczoną wcześniej resztą z dzielenia tak długo, aż obliczenia sprowadzą drugi parametr do zera. Tak właśnie implementuje się rekurencję.

3.4.4. Implementacje algorytmów sortujących

Wspomnimy tutaj o dwóch najczęściej używanych algorytmach sortujących. Chociaż w praktyce potrzeba ich samodzielnego pisania występuje bardzo rzadko, to mimo wszystko znajomość tych algorytmów jest wymagana, gdyż często są one zaimplementowane w bibliotekach.

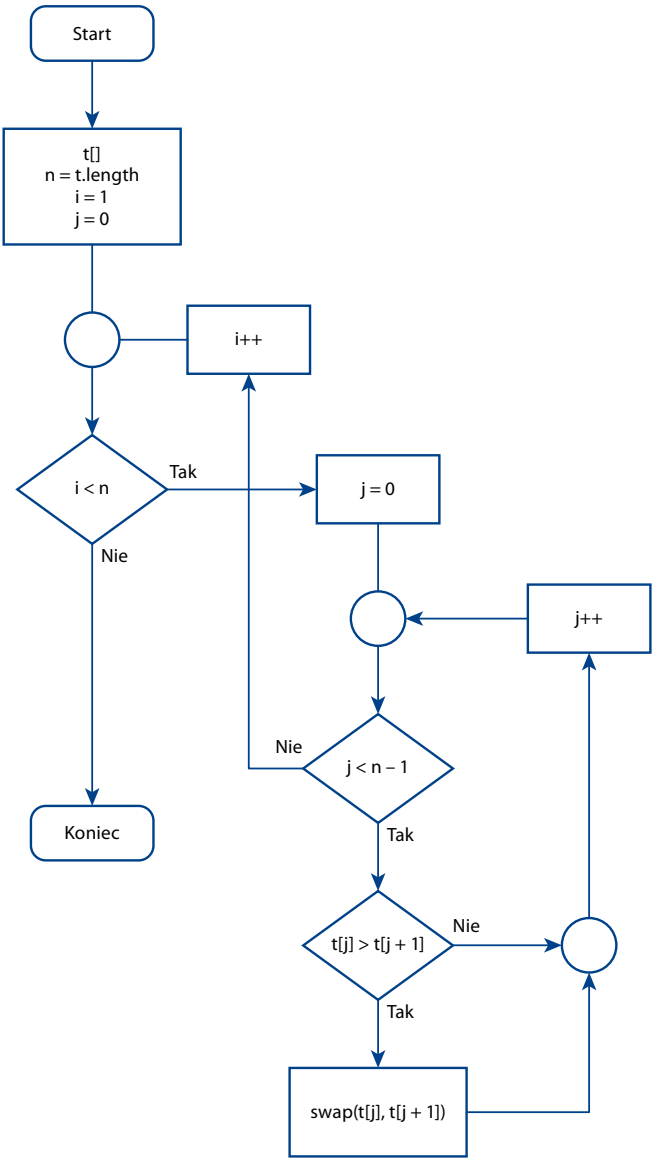
3.4.4.1. Sortowanie metodą bąbelkową

Kolejny schemat blokowy, który zostanie tu zaprezentowany, przedstawia sortowanie metodą bąbelkową.

DEFINICJA

Metoda bąbelkowa to metoda sortowania, która iteracyjnie porównuje dwa elementy i je przestawia, o ile pierwszy jest większy od drugiego.

Oto schemat blokowy ilustrujący kolejne kroki wykonywane podczas sortowania bąbelkowego (rysunek 3.7):



Rysunek 3.7. Przykładowy schemat blokowy sortowania bąbelkowego

Przykład 3.13

Wykonajmy na jego podstawie analizę sortowania zadeklarowanej poniżej tablicy `array`.

```
let array = [2, -5, 3, 1]
```

Przebieg sortowania przedstawiono w tabeli 3.4. Na każdym etapie wyróżnieniem wpisywane są te elementy tablicy wyjściowej, które zostały właśnie przestawione.

Tabela 3.4. Sortowanie tablicy array za pomocą algorytmu bąbelkowego

Tablica wejściowa	Element a	Element b	Porównanie elementów a i b (a > b)	Tablica wyjściowa
Pierwsza iteracja				
2 –5 3 1	2	–5	Prawda	*–5 2* 3 1
–5 2 3 1	2	3	Fałsz	–5 2 3 1
–5 2 3 1	3	1	Prawda	–5 2 *1 3*
Druga iteracja				
–5 2 1 3	–5	2	Fałsz	–5 2 1 3
–5 2 1 3	2	1	Prawda	–5 *1 2* 3
–5 1 2 3	2	3	Fałsz	–5 1 2 3

Rozwiązanie to jest mniej wydajne od sortowania quicksort, ale o wiele prostsze w samodzielnej implementacji.

UWAGA

W przypadku sortowania kolekcji zdefiniowanych w wybranym języku programowania można z góry wykorzystać metody już opisane i zaimplementowane przez społeczność. Dla tablicy:

```
let numbers = [3, 5, 12, 3.3]
```

w przypadku zastosowania metody `numbers.sort()` uzyskamy tablicę posortowaną w kolejności od najmniejszej do największej wartości.

Gdybyśmy chcieli uzyskać ułożenie elementów w odwrotnej kolejności, należałoby jeszcze użyć funkcji `reverse()`, która odwraca porządek sortowanych elementów — po jej zastosowaniu nastąpi zatem zamiana elementów: pierwszego z ostatnim itd.

Trzeba więc najpierw posortować tablicę, a następnie odwrócić kolejność elementów:

```
numbers.sort();
numbers.reverse();
```

→

UWAGA cd.

Gdybyśmy w kolekcji umieścili napisy, sortowanie byłoby realizowane w porządku alfabetycznym.

W przypadku języka C++ sortowanie jest dostępne po zaimportowaniu biblioteki `<algorithm>`. Metoda `sort()` przyjmuje parametry pozwalające wskazać pierwszy i ostatni element tablicy, a także to, czy chcemy sortować rosnąco, czy malejąco:

```
int tab[6] = {43, 1, 7, 21, 3, 12};
int n = sizeof(tab) / sizeof(tab[0]);

sort(tab, tab + n, greater <int>());
```

Metoda `sizeof(tab)` zwraca wielkość tablicy, a do uzyskania pierwszego elementu wykorzystamy wskaźnik na tablicę — nazwa wskaże na pierwszy element, a `tab + n` na ostatni. Ostatni parametr to informacja, czy tablica ma być posortowana malejąco; jeśli tak, to parametr jest wymagany, w przeciwnym wypadku nie jest.

3.4.4.2. Szybkie sortowanie (quicksort)

Quicksort (rysunek 3.8) jest obecnie najczęściej używanym algorytmem do sortowania elementów. Jego główną zaletą jest niski stopień złożoności obliczeniowej.

Quicksort to algorytm oparty na strategii „dziel i zwyciężaj".

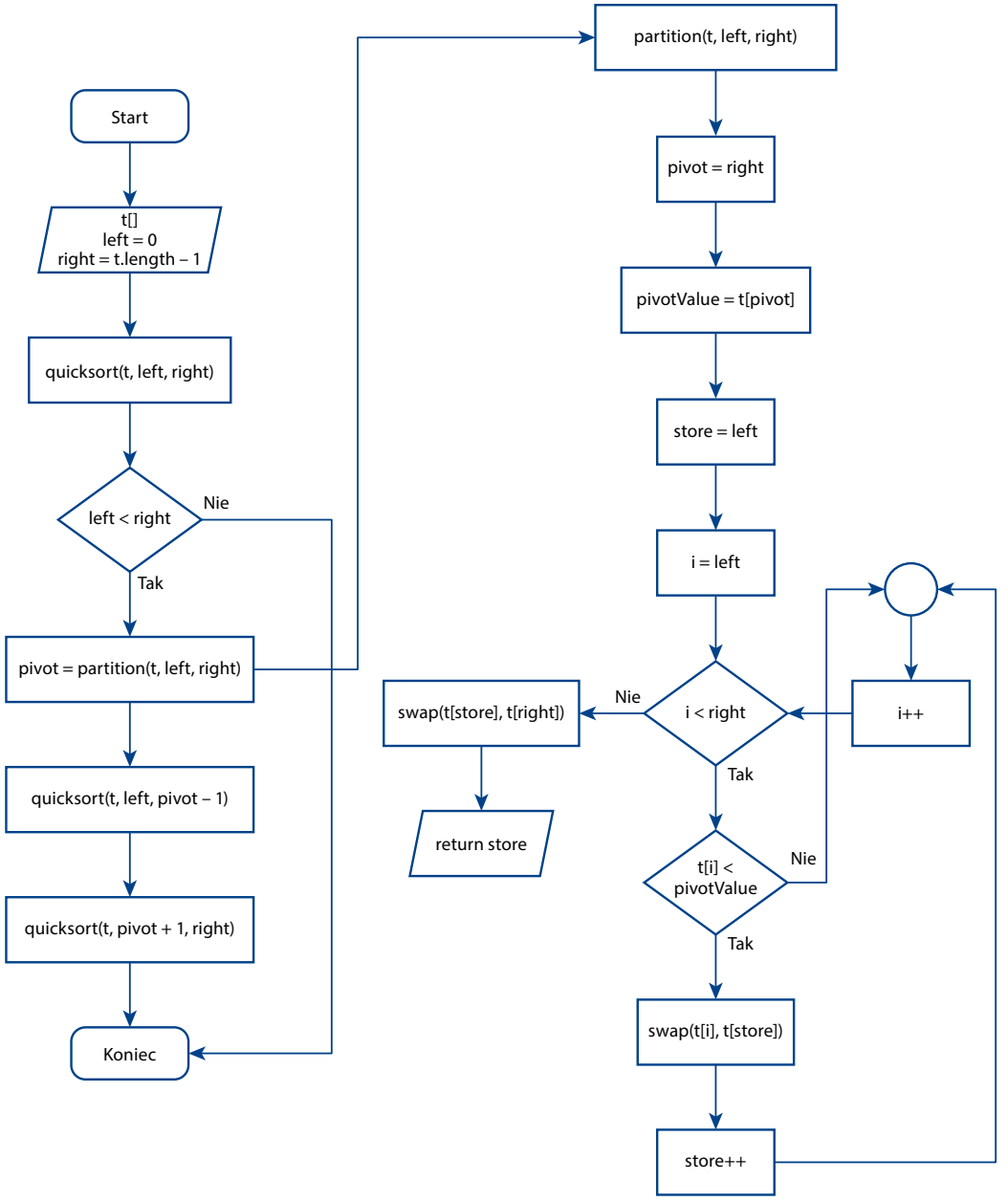
Dziel — w celu podzielenia tablicy musimy wybrać element rozdzielający (*pivot*). Względem tego elementu przeniesiemy wszystkie dane znajdujące się w tablicy, tak aby elementy, które są od niego mniejsze lub mu równe, znalazły się po jego lewej stronie, a wszystkie pozostałe wartości przeniesiemy na prawo od niego. Procedurę tę nazywamy partycjonowaniem. Nieważny jest porządek elementów, zależy nam jedynie na ich pozycji względem elementu rozdzielającego. Z reguły wybiera się element położony skrajnie w prawo, ale są też inne podejścia — wybór losowy bądź strategia „środkowy z trzech".

Zwyciężaj — rekurencyjnie (bezpośrednio) sortujemy powstałe podtablice.

Scal — nie musimy już nic robić. Wszystkie elementy na lewo od pivotu są od niego mniejsze lub mu równe i zostały posortowane, tak samo jak wartości po jego prawej — one z kolei są od niego większe.

Przykład 3.14

Posortujmy tablicę o zawartości [4, 7, 5, 10, 8, 3, 6]. Na pivot wybieramy wartość 6, po partycjonowaniu tablica przyjmuje strukturę [4, 7, 5, 3, 6, 10, 8], a dokładniej mamy dwie podtablice: [4, 7, 5, 3, 6] oraz [10, 8]. Dla pierwszej podtablicy wybieramy pivot 6 i partycjonujemy: [4, 5, 3, 6], które musimy już posortować, oraz [7]. Druga podtablica podzieli się na [8] i [10]. Scalamy wszystko i mamy posortowany pierwotny zbiór [3, 4, 5, 6, 7, 8, 10].



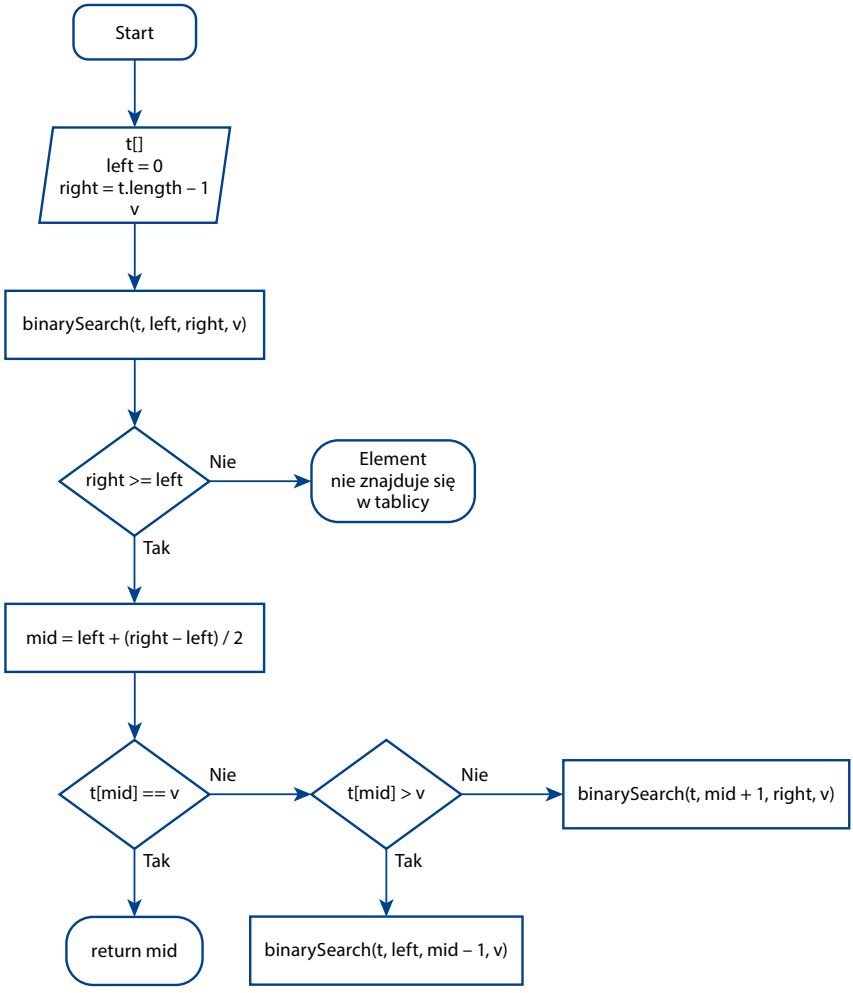
Rysunek 3.8. Przykładowy schemat blokowy sortowania quicksort

3.4.5. Wyszukiwanie binarne

Jest to rodzaj algorytmu, który stosowany jest na co dzień, chociaż zazwyczaj nie wiemy, że tak się nazywa. Przykładem jest chociażby przeszukiwanie telefonu komórkowego w celu odnalezienia danej aplikacji. W bardzo prosty sposób przeprowadzane jest sprawdzanie (w porządku alfabetycznym), czy na danej stronie pulpitu zmieścił się skrót do wyszukiwanej przez nas aplikacji. Kolejny raz wykorzystuje się tu zasadę „dziel i zwyciężaj". W tym przypadku polega to na podzieleniu pulpitu telefonu na strony i na każdej z nich jest sprawdzane, czy skrót do aplikacji może na niej być; suma odpowiedzi da zwrotną informację o tym, na której stronie może być wyszukiwana aplikacja.

Na podstawie wyżej wspomnianego przykładu można wnioskować, że założeniem początkowym jest to, że dana kolekcja do przeszukania jest posortowana.

Przeanalizowanie schematu blokowego wyszukiwania binarnego (rysunek 3.9) podpowiada, że mamy możliwość implementowania rozwiązania przez rekurencję.



Rysunek 3.9. Wyszukiwanie binarne — schemat blokowy

Schemat blokowy odzwierciedla wyszukiwanie binarne, które polega na dzieleniu listy na pół do momentu, w którym ograniczy się ją do jednej pozycji. Wyznaczone zostały następujące kroki:

1. Określenie `min` i `max`, na początku oznaczających cały zakres zbioru.
2. Obliczenie średniej z `min` i `max`, z zaokrągleniem w dół do liczby całkowitej jako indeksu środkowego elementu:
   a) jeśli element środkowy jest poszukiwaną wartością, możemy zakończyć program, ponieważ znaleźliśmy indeks szukanego elementu;
   b) jeżeli element środkowy jest mniejszy od szukanej wartości, zmieniamy `min` na wartość o jeden większą od średniej;
   c) jeżeli element środkowy jest większy od szukanej wartości, do zmiennej `max` przypisujemy liczbę o jeden mniejszą od średniej;
   d) wracamy do kroku b.

Przykład 3.15

Rozważmy następujący przykład. Chcemy znaleźć nasz ulubiony utwór zespołu Metallica. Mamy w swoich zasobach 107 piosenek zespołu, ale bez podziału na albumy. Mamy po prostu listę utworów posortowaną alfabetycznie. Załóżmy, że interesuje nas ostatni utwór. Przeszukując nasz zbiór piosenek element po elemencie, czyli stosując wyszukiwanie liniowe, musimy sprawdzić każdy element naszej dyskografii. Byłoby to dość kłopotliwe i czasochłonne. Tymczasem wyszukiwanie binarne sprawdzi zaledwie 7 utworów przed zwróceniem właściwego wyniku.

Jak?

Powiedzmy, że chcemy znaleźć utwór 86. Obliczamy średnią według wskazanego wzoru `(left + (right-left))/2` z danymi `left=0`, `right=107`. Otrzymujemy wynik 53 (po zaokrągleniu w dół). Nasz utwór znajduje się dalej; odrzucamy wszystko poniżej średniej i szukamy średniej z nowego przedziału: 54–107. Wynosi ona 80. Nasz utwór ponownie znajduje się powyżej tej wartości, rozważamy więc kolejny przedział: 81–107. Nowa średnia to 94; tym razem nasz utwór znajduje się wcześniej. Kolejny nowy przedział to 81–93. Średnia z tego przedziału wynosi 87, jesteśmy więc już blisko. Obecny przedział to 81–86. Średnia to 83. Powstaje następny przedział, 84–86. Ostatnia średnia to 85, końcowy przedział jest jednoelementowy.

Oznacza to, że znaleźliśmy utwór.

W tym celu sprawdziliśmy 7 pozycji.

UWAGA

Przewagę wyszukiwania binarnego nad innymi algorytmami wyszukiwania widać dobrze na dużym zbiorze danych.

3.4.6. Wyszukiwanie w zaawansowanych strukturach

W językach programowania mamy dodatkowe struktury, na których można oprzeć bardziej złożone algorytmy. Jedną z tych struktur jest kolejka. Elementy kolejki są pobierane i obsługiwane w takiej samej kolejności, w jakiej były do niej dodawane. Wyraża to akronim FIFO (ang. *First In, First Out* — pierwszy na wejściu, pierwszy na wyjściu).

Przykład 3.16

Najlepszą ilustracją dla tej struktury jest właśnie kolejka do kasy w sklepie. Osoba, która się w niej ustawi najwcześniej, zostanie obsłużona jako pierwsza i pierwsza wyjdzie ze sklepu. Ostatni klient w kolejce zapłaci na samym końcu i opuści sklep po wszystkich, którzy stali w kolejce przed nim.

Specyficzną odmianą kolejki jest kolejka priorytetowa (ang. *Priority Queue*). Elementy tej struktury posiadają swoje priorytety i są obsługiwane zgodnie z nimi. Dane o najwyższym priorytecie zostaną pobrane jako pierwsze, elementy o niższych priorytetach zostaną obsłużone dopiero, gdy wszystkie wartości z wyższymi priorytetami zostaną usunięte z kolejki.

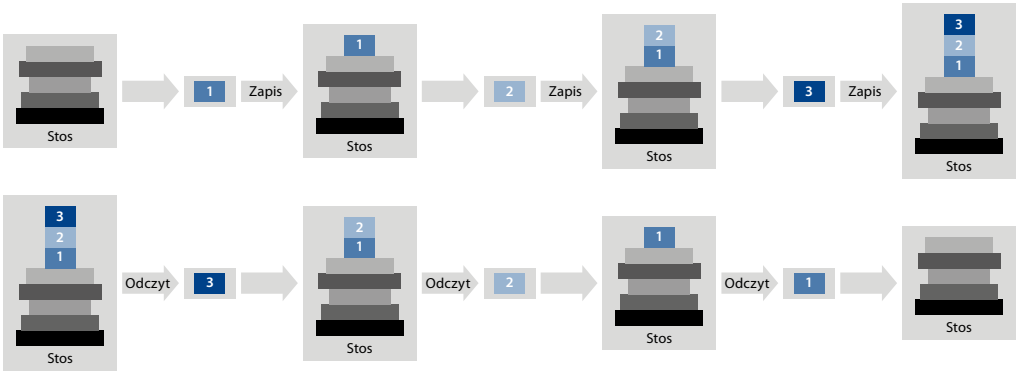
Kolejną strukturą jest stos, który oparty jest na zasadzie LIFO (ang. *Last In, First Out*) — pierwszy element, który dodamy do stosu, zostanie obsłużony na samym końcu, a ostatni zostanie obsłużony jako pierwszy. Oznacza to, że — inaczej niż w przypadku kolejek — dane umieszczone na stosie odczytujemy w kolejności odwrotnej do ich zapisu.

Przykład 3.17

Wyobraźmy sobie, że sprzątamy po przyjęciu. Zbieramy ze stołu brudne naczynia, bierzemy do ręki pierwszy talerz, na nim kładziemy kolejny i taki stos talerzy niesiemy do kuchni. Nie mamy zmywarki, więc musimy umyć wszystko sami. Bierzemy talerz z samej góry, myjemy go i odkładamy, po czym bierzemy kolejne, przechodząc stopniowo w dół stosu, aż wreszcie dotrzemy do talerza znajdującego się najniżej — tego, który zdjęliśmy ze stołu jako pierwszy. Tak właśnie działa obieg LIFO. Ostatni talerz, który zabraliśmy ze stołu, dołożyliśmy do stosu brudnych naczyń na samej górze, w związku z czym zdejmujemy go i myjemy w pierwszej kolejności. Pierwszy, od którego zaczęliśmy stos, znajduje się na samym dole i umyjemy go jako ostatni.

Najłatwiej zrozumieć różnicę między tymi dwoma strukturami na podstawie schematów (rysunki 3.10 i 3.11).

Rysunek 3.10. Odwzorowanie kolejności działań przy zapisie i odczycie danych — kolejka

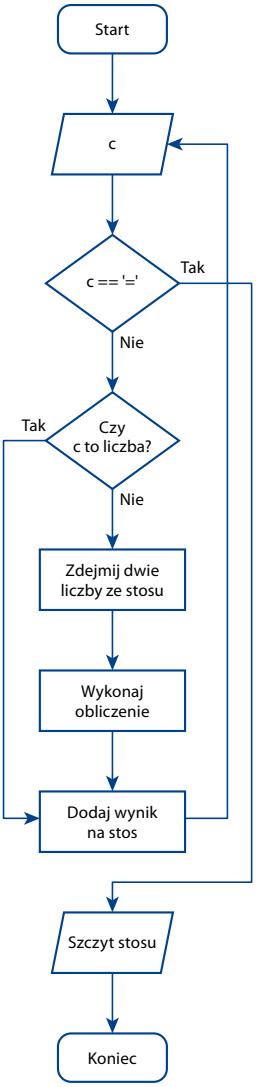


Rysunek 3.11. Odwzorowanie kolejności działań przy zapisie i odczycie danych — stos

Przykład 3.18

Strukturę stosu omówimy na podstawie notacji zwanej ONP, czyli odwrotna notacja polska (ang. RPN — *reverse Polish notation*). Jest to jeden ze sposobów zapisu wyrażeń arytmetycznych. Charakteryzuje się brakiem konieczności stosowania nawiasów. W klasycznym zapisie operatory stawiamy pomiędzy argumentami, np. 2 · 2. Dla bardziej złożonych obliczeń, np. 2 + 2 · 2, jeżeli chcemy określić inną niż domyślna (wynikająca z priorytetu operatorów) kolejność wykonywania działań, musimy użyć nawiasów, np. (2 + 2) · 2. W przypadku stosowania ONP nie mamy tego problemu, gdyż operator zapisujemy po argumentach, na których zostanie użyty, np. 2 2 ·, 2 2 2 · +, 2 2 · 2 +. Do utworzenia algorytmu wykonującego obliczenia za pomocą ONP zastosujemy stos. Wyrażenie czytamy od lewej do prawej; jeżeli pobraną wartością jest liczba, ląduje na stosie. W przypadku natrafienia na operator pobieramy ze stosu dwie liczby, wykonujemy obliczenie zgodnie z operatorem, a wynik odkładamy na stos. W momencie, w którym obsłużymy wszystkie dane wejściowe, ostateczny wynik będzie się znajdować na szczycie stosu.

Algorytm ten można zilustrować następującym schematem blokowym (rysunek 3.12):



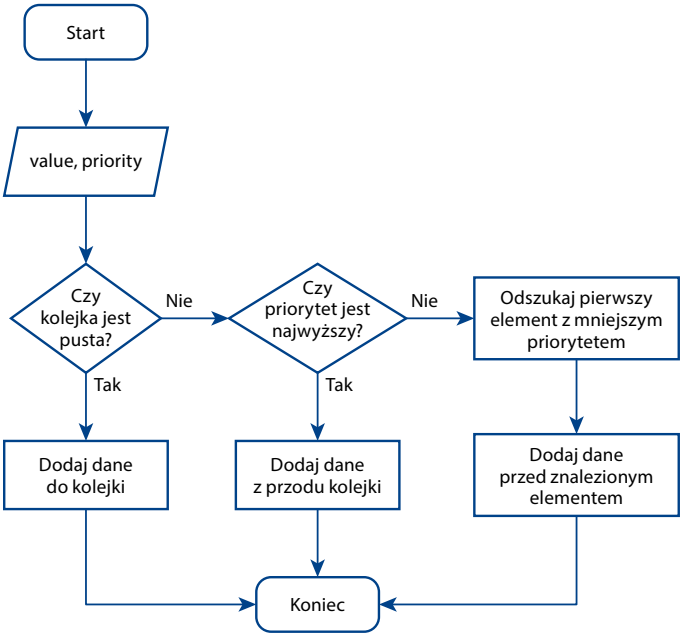
Rysunek 3.12. Przykładowy schemat blokowy algorytmu ONP

UWAGA

Implementacje stosu i kolejki zależą od języka programowania. W języku JavaScript do implementacji obu tych struktur możemy wykorzystać listę. W C++ można natomiast skorzystać z udostępnianych w bibliotece standardowej kolekcji wektora (`vector`) lub kolejki (`queue`), jak również kolejki priorytetowej (`priority_queue`).

Przykład 3.19

Zobaczmy, jak wygląda wyszukiwanie elementu w kolejce na przykładzie dodawania nowego elementu do kolejki priorytetowej. Najpierw utwórzmy schemat blokowy takiego działania (rysunek 3.13).



Rysunek 3.13. Schemat blokowy dodawania elementu do kolejki priorytetowej

Teraz zaimplementujemy rozwiązanie (listing 3.8).

Listing 3.8

Implementacja kolejki priorytetowej

```
class Element {
    constructor(value, priority) {
        this.value = value;
        this.priority = priority;
    }
}
```

```
class PriorityQueue {
    constructor() {
        this.elements = [];
    }
    pushOnQueue(element) {
        let contain = false;
        for (let i in this.elements) {
            if (this.elements[i].priority > element.priority) {
                this.elements.splice(i, 0, element);
                contain = true;
                break;
            }
        }
        if (!contain) {
            this.elements.push(element);
        }
    }
    display() {
        for (let i in this.elements) {
            console.log(this.elements[i].value);
        }
    }
}
let priorityQueue = new PriorityQueue();
priorityQueue.pushOnQueue(new Element("Janek", 3));
priorityQueue.pushOnQueue(new Element("Maja", 1));
priorityQueue.pushOnQueue(new Element("Kasia", 3));
priorityQueue.pushOnQueue(new Element("Ola", 1));
priorityQueue.pushOnQueue(new Element("Marcin", 2));
priorityQueue.display();
```

Zaimplementowaliśmy kolejkę priorytetową poprzez inicjalizację klasy `Element` z właściwościami `value` (wartość) oraz `priority` (priorytet). Następnie zaimplementowaliśmy metodę `pushOnQueue`, która służy do dodawania elementów do stworzonej struktury kolejki w postaci listy z obiektami typu `Element`. Kolejnym krokiem jest już implementacja rozwiązania na podstawie schematu blokowego.

UWAGA

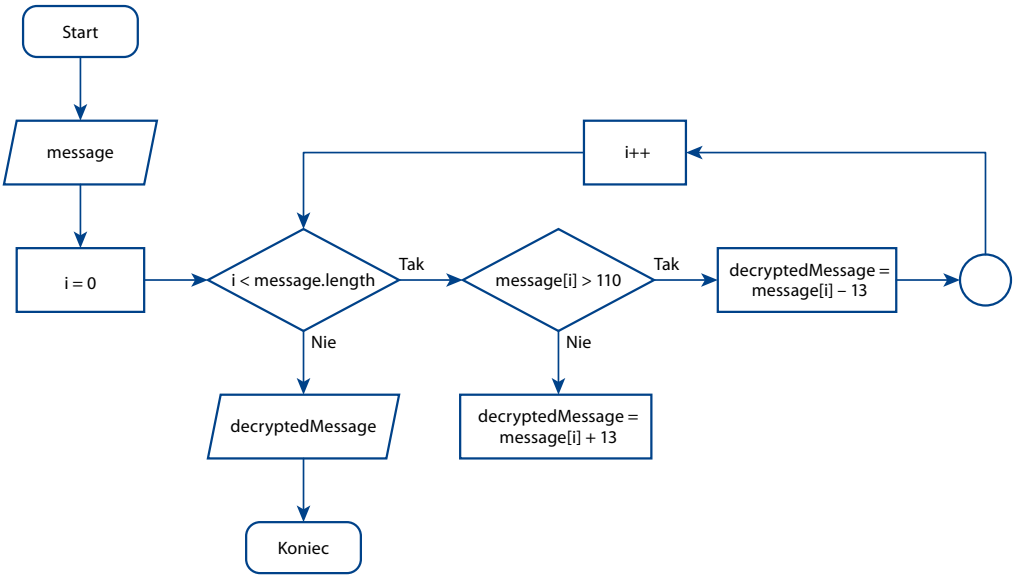
Rozwiązanie tego problemu jest o wiele prostsze w języku C++, jednak jego prezentacja wykracza poza ramy tego podręcznika.

Jak widać, samo dodanie elementu oparte jest na przeszukaniu tablicy w celu odnalezienia elementów o tym samym priorytecie. Dopiero po ich odnalezieniu można dodać element. Priorytet 1 jest najwyższy, natomiast 3 jest najniższy.

3.4.7. Algorytm szyfrujący ROT-13

Od czasu do czasu zachodzi potrzeba zaszyfrowania wiadomości. Z szyfrowaniem spotykamy się chociażby w przypadku podpisu elektronicznego, potrzebnego np. podczas logowania się do systemu bankowego. Jednym z podstawowych algorytmów szyfrujących jest ROT-13 (rysunek 3.14).

Zasada jego działania opiera się na zamianie każdego kolejnego znaku na symbol występujący 13 pozycji po nim. Dla klasycznego alfabetu łacińskiego (26 znaków) jest on funkcją odwrotną, co oznacza, że jest w stanie nie tylko zaszyfrować, ale też odszyfrować wiadomość.



Rysunek 3.14. Schemat działania algorytmu szyfrującego ROT-13 dla małych znaków

UWAGA

W centralnym punkcie algorytmu z rysunku 3.14 sformułowaliśmy warunek, w związku z którym `message[i]` ma być większe niż 110, co w ASCII odpowiada znakowi `'n'` w kodzie szesnastkowym `(message[i] > 'n')`. Jest to spowodowane tym, że litera *n* jest w połowie klasycznego 26-znakowego alfabetu łacińskiego.

Na podstawie zaprezentowanego wyżej schematu blokowego moglibyśmy utworzyć funkcję (listing 3.9):

Listing 3.9

Funkcja szyfrująca

```
function rot13(message) {
  let decryptedMessage = "";
  for(let i = 0; i < message.length; i += 1) {
    let code = message.charCodeAt(i);
    // zawrócenie od litery n na początek alfabetu
    if (code >= 110) {
      decryptedMessage += String.fromCharCode(code - 13);
    } else {
      decryptedMessage += String.fromCharCode(code + 13);
    }
  }
  return decryptedMessage;
}

let message = "hello";
console.log(rot13(message));
```

UWAGA

Wszystkie znaki wpisywane za pomocą klawiatury (lub innego urządzenia wejściowego) mają swoje odwzorowanie w liczbach. To odwzorowanie nazywamy kodem ASCII. Poniżej podano kody dla małej i wielkiej litery *a*, a także dla małej litery *n*:

a — 97

A — 65

n — 110

W przykładowej implementacji algorytmu ROT-13 w wierszu 6. utworzono warunek `if (code >= 110)`, gdyż trzeba zmapować małą literę *n* na kod ASCII. Następnie, skoro wiemy, że *n* jest w alfabecie 13. w kolejności, to w celu powrotu na początek alfabetu można odjąć 13 od kodu 110. Aby przejść na koniec alfabetu, można dodać 12 do 110.

3.4.8. Algorytmy heurystyczne

Temat algorytmów jest bardzo szeroki i dość skomplikowany. Należy pamiętać, że czasami rozwiązanie problemu wiąże się ze zbyt dużą, większą niż akceptowalna złożonością obliczeniową. Niekiedy wręcz jego znalezienie wydaje się niemożliwe. Wówczas z pomocą przychodzą algorytmy heurystyczne. Tego rodzaju algorytmy z reguły nie dają gwarancji znalezienia optymalnego rozwiązania, umożliwiają jednak odkrycie go w ogóle w sytuacjach, kiedy bez ich zastosowania byłoby to niemożliwe.

Przykład 3.20

Podstawowe rozwiązanie problemu komiwojażera pochodzi z grupy algorytmów heurystycznych.

Na czym polega problem komiwojażera?

Wyobraźmy sobie sprzedawcę podróżującego przez różne miasta. Codziennie wyrusza on z miasta, w którym mieszka, i dociera do wszystkich ośrodków na swojej drodze, odwiedzając każdy z nich dokładnie jeden raz. Na koniec wraca do domu. Oczywistym jest, że chce, aby jego podróż była jak najkrótsza, i to właśnie stanowi nasz problem, który musimy rozwiązać. Możemy zastosować prosty algorytm, który sprawdzi wszystkie drogi i zwróci tę optymalną. Niestety, rozwiązanie to obarczone jest wykładniczą złożonością obliczeniową, co ogranicza nas do analizy prostych modeli.

Temat algorytmów jest zgłębiany od stuleci przez największych filozofów i matematyków. Efekty ich pracy widać chociażby przy budowie linii warszawskiego metra czy w skróceniu czasu jazdy pociągu relacji Kraków – Kołobrzeg. W obu przypadkach prowadzono niejednokrotnie rozważania podobne do tych, które umożliwiają rozwiązanie wspomnianego wcześniej problemu komiwojażera. Warto też dobrze rozważyć wybór języka, w którym chcemy zaimplementować algorytm; nie zawsze ten, który jest nam najbliższy i który najczęściej wykorzystujemy, najlepiej się do tego nadaje.

3.5. Projektowanie klas (UML)

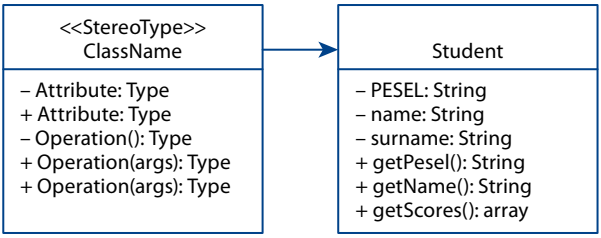
Do tej pory, gdy była mowa o tym, że trzeba utworzyć klasę o wskazanych właściwościach, używaliśmy jej opisu słownego.

W programowaniu rzadko stosuje się takie rozwiązanie. Ten zapis został zastąpiony przez **UML** (*Unified Modeling Language*). Mówiąc najprościej, zastąpiono tekst diagramem.

W tym zapisie stosuje się pewne stałe i powszechnie przyjęte zasady.

Przykład 3.21

Na rysunku 3.15 zaprezentowano zapis za pomocą diagramu UML klasy `Student`. Każdego reprezentanta tej klasy charakteryzują właściwości: PESEL, imię (`name`) i nazwisko (`surname`). Zdefiniowano dlań także trzy metody, pozwalające na pobranie numeru PESEL, imienia i tablicy z ocenami.



Rysunek 3.15. Przedstawienie klasy Student za pomocą diagramu UML

Klasa jest przedstawiona za pomocą tabelki. W pierwszej części (w nagłówku) umieszczona jest nazwa klasy, w drugiej mamy zmienne i atrybuty/właściwości (w kolumnie po prawej) oraz ich typy (w kolumnie po lewej). W tej drugiej sekcji przed każdym oznaczeniem stosuje się znaki:

- plus (+) — oznacza, że zmienna lub metoda jest publiczna;
- minus (–) — oznacza, że zmienna lub metoda jest prywatna.

UWAGA

Zmienna/metoda publiczna to taka, do której można uzyskać dostęp z dowolnego miejsca kodu. Do zmiennej/metody prywatnej można się odwołać bezpośrednio tylko z wnętrza klasy, w której została zadeklarowana/zdefiniowana, a także z klas, które z niej dziedziczą (problem dziedziczenia wykracza jednak poza zakres tego podręcznika). Z poziomu pozostałych klas można się do nich odwoływać za pomocą tzw. metod dostępowych (*getterów* — jak `getPesel()` czy `getName()` — i *setterów*).

Przykład 3.22

Gdybyśmy chcieli zlecić utworzenie klasy z przykładu 3.9, posługując się językiem naturalnym, moglibyśmy napisać:

Utwórz klasę `Student` o właściwościach `name` (imię) i `surname` (nazwisko). Obydwie właściwości powinny być typu tekstowego. Identyfikatorem obiektów w klasie niech będzie numer PESEL. Utwórz także metodę prywatną pobierającą ten numer oraz dwie metody publiczne, odpowiedzialne za:

- pobranie imienia studenta,
- zwrócenie jego ocen.

Prawda, że widać różnicę? Kiedy tylko przyzwyczaimy się do zapisu UML, jego stosowanie będzie naturalnym wyborem, ponieważ jest on prostszy i bardziej czytelny.

3.6. Wzorce projektowe

3.6.1. Czym są wzorce i do czego się przydają?

Podczas tworzenia kodu bardzo często napotykamy trudności, które są typowe — możemy być pewni, że mierzyło się z nimi już wielu programistów przed nami. Na szczęście wskutek tego wypracowano standardowe rozwiązania takich problemów, czyli właśnie wzorce projektowe.

Wzorzec projektowy ma na celu ułatwić pracę poprzez umożliwienie skorzystania z gotowych szablonów rozwiązań w spodziewanych, standardowych sytuacjach.

Przykład 3.23

Przenieśmy nasze rozważania o wzorcach projektowych do sfer znanych nam nieco lepiej. Zwróćmy uwagę na wznoszone dzisiaj budowle. Każdy budynek jest inny, ale większość z nich korzysta z tych samych schematów i norm dotyczących np. ilości światła w pomieszczeniu czy też dopuszczalnego poziomu hałasu w mieszkaniu. Będąc w dużym mieście, łatwo zauważymy, że kiedy obejdziemy niemal dowolny budynek stojący przy ruchliwej ulicy, natrafimy na miejsca, gdzie hałas jest bardzo mocno odczuwalny, podczas gdy inna część budynku będzie idealnie wyciszona i przygotowana w ten sposób pod część sypialnianą. Jest to spowodowane tym, że współczesne budynki są oparte na dobrze znanych architektom wzorcach projektowych.

W programowaniu też mówi się o funkcji architekta. Rola takiej osoby sprowadza się właśnie do *projektowania* aplikacji na podstawie z góry znanych wzorców projektowych. Wzorce stosowane w projektowaniu zostały bardzo dobrze opisane w książce *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku* autorstwa tzw. *Gangu Czworga*.

Książka ta została wydana ponad dwie dekady temu i do dziś rozwiązania powszechnych problemów programistycznych oparte są na jednym z 23 zawartych w niej wzorców. Warto o tym pamiętać i napotykając jakąś trudność, która wydaje się nie do przejścia, nie próbować wymyślać koła na nowo, lecz zajrzeć do tej klasycznej publikacji w celu sprawdzenia, czy nie ma już gotowego rozwiązania.

Dość długo mówimy już w tym rozdziale o wzorcach projektowych, nie podając ich definicji. Czas to zatem uczynić. Oto definicja autorstwa członków Gangu:

Wzorzec to rozwiązanie problemu w ramach pewnego kontekstu.

Spróbujmy ją wyjaśnić na przykładzie.

Przykład 3.24

Sytuacja: na wakacjach w Wielkiej Brytanii zaistniała potrzeba naładowania telefonu.

Kontekst: wtyczka kontaktowa i ładowarka do telefonu.

Problem: nie można włożyć ładowarki do kontaktu ze względu na to, że na Wyspach stosowany jest inny układ bolców we wtyczce.

Rozwiązanie: przejściówka/adapter.

Adapter zastosowany do rozwiązania tego problemu przyda się zresztą nie tylko w Wielkiej Brytanii, ale także w Chinach czy USA oraz we wszystkich krajach, w których stosuje się inny standard wtyczek. Dlatego właśnie można uznać go za rozwiązanie wzorcowe.

UWAGA

Aby stosować różnego rodzaju przejściówki, wcale nie trzeba wyjeżdżać poza kontynent. Trzeba podłączyć stary monitor z gniazdem VGA (D-SUB) do nowego laptopa, który ma już tylko nowe, cyfrowe wejścia (HDMI lub DisplayPort). Chcesz podłączyć słuchawki do telefonu, który nie ma wejścia typu minijack, a jedynie port micro USB? W obu tych przypadkach trzeba skorzystać z adaptera.

UWAGA

W programowaniu sytuacja, kiedy trzeba dostosować jeden system informatyczny do innego, zdarza się nader często. Dlatego jeden z wzorców opracowanych przez Gang Czworga to właśnie wzorzec adapter.

W książce GoF (ang. *Gang of Four* — Gang Czworga) każdy wzorzec projektowy ma:

1. Odpowiednio dobraną nazwę wzorca (sugerującą, jaki charakter będzie miało rozwiązanie, np. nazwa *adapter* dla wzorca *adaptującego* niepasujące do siebie elementy).
2. Opis problemu — kontekst oraz warunek (warunki) uzasadniające użycie wzorca.
3. Rozwiązanie — opis elementów, które wchodzą w skład rozwiązania, zależności pomiędzy nimi, ich odpowiedzialności oraz interakcji.
4. Konsekwencje — opis ewentualnych problemów związanych z zastosowaniem wzorca. Najczęściej jest to zmniejszenie wydajności w określonych sytuacjach. Ta sekcja pozwoli nam zdecydować, czy niepożądane skutki użycia wzorca nie są większe niż korzyści, jakie wzorzec daje.

UWAGA

Trzeba zdawać sobie sprawę z tego, że wzorzec projektowy nie jest gotowym, będącym rozwiązaniem problemu kodem, który możemy skopiować i wkleić do swojego projektu. To tak, jakbyśmy oczekiwali, że ubranie, które świetnie prezentuje się na koledze, będzie równie doskonale pasować nam. Oczywiście tak nie jest, nie powielajmy więc wzorców, nie zastanawiając się nawet, jak je dostosować do naszego problemu. Wzorzec jest tym, o czym mówi jego nazwa — *wzorem rozwiązania*, a nie rozwiązaniem.

Istnieją także gotowe implementacje wzorców. W ich przypadku liczba zmian, jakich będziemy musieli dokonać, jest zapewne mniejsza. Oczywiście takich implementacji jest bardzo dużo. Zanim wykorzystamy którąś z nich, zastanówmy się, dlaczego uważamy ją za lepszą od innych. Trzeba także pamiętać, że najlepiej korzystać ze źródeł z jak największą liczbą komentarzy. Nawet korzystając ze źródeł sprawdzonych i wiarygodnych, nie powinniśmy jednak po prostu przekopiowywać kodu bez próby jego zrozumienia!

3.6.2. Klasyfikacja wzorców

W książce Gangu Czworga zdefiniowano następujące grupy wzorców:

- **Strukturalne** — pomagają rozwiązać problemy związane z zarządzaniem strukturą obiektów oraz strukturami złożonymi z tych obiektów.
- **Behawioralne/czynnościowe/operacyjne** — porządkują zależności między obiektami i zarządzają komunikacją między nimi.
- **Konstrukcyjne/kreacyjne** — ułatwiają tworzenie obiektów poprzez delegowanie procesu tworzenia do innych klas.

3.6.3. Implementacja wzorca

Skupmy się teraz na procesie tworzenia własnej implementacji wzorca.

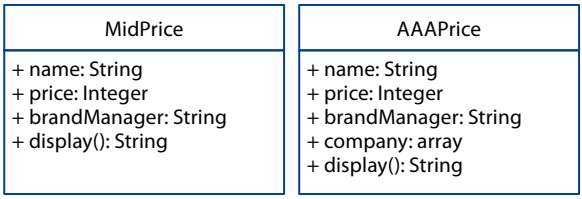
Przykład 3.25

Załóżmy, że mamy kod źródłowy do definiowania cen gier komputerowych. Od sprzedawcy wiemy, że istnieją dwa rodzaje cen gier komputerowych:

- cena sugerowana dla gier ze średniego segmentu (*midPrice*),
- cena sugerowana dla gier z segmentu wysokobudżetowego (*AAAPrice*).

Sprzedawca wie, że będzie rozwijał swój katalog gier, więc na pewno typy sugerowanych cen również będą ulegać zmianie.

Czas na implementację problemu według zaakceptowanego przez klienta modelu w notacji UML. Na razie spróbujmy przeprowadzić ją bez wykorzystania wzorca projektowego. Oto diagram klas do utworzenia (rysunek 3.16).



Rysunek 3.16. Diagram klas dla sklepu z grami

Oto kod, który napisaliśmy (listing 3.10):

Listing 3.10

Kod klas dla sklepu z grami

```
class MidPrice {
    name = 'mid';
    price = 50;
    brandManager = 'Maciej Nowak';
    display = function() {
        return console.log(`osoba odpowiedzialna za segment ${this.
name} to ${this.brandManager} (cena sugerowana to: ${this.price})`);
    }
}

class AAAPrice {
    name = 'aaa price';
    price = 250;
    brandManager = 'Marianna Srebrna';
    company = ['EA', 'Microsoft'];
    display = function() {
        return console.log(`osoba odpowiedzialna za segment ${this.
name} to ${this.brandManager} (cena sugerowana to: ${this.price})`);
    }
}
```

Teraz tworzymy funkcję, która ma na celu dodanie wszystkich kategorii cenowych zgodnie z segmentem sprzedażowym (listing 3.11).

Listing 3.11

Kod funkcji do dodawania cen

```
1 function games() {
2    let games = [];
```

```
3     games.push(new MidPrice());
4     games.push(new AAAPrice());
5
6     for (let i = 0; i < games.length; i++) {
7         console.log(games[i].display());
8     }
9 }
```

Wyjaśnijmy teraz, co się dzieje w poszczególnych liniach kodu.

1. Zdefiniowanie metody.

2. Deklaracja pustej tablicy z grami.

3. Dodanie do tablicy nowego obiektu `MidPrice`.

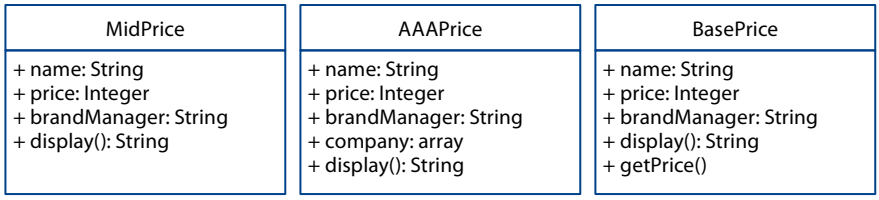
4. Dodanie do tablicy nowego obiektu `AAAPrice`.

5. Odstęp.

6. Utworzenie pętli o kroku jeden, przechodzącej po elementach tablicy.

7. Wypisanie na konsoli informacji zawartych w obydwóch klasach (`MidPrice` i `AAAPrice`) za pomocą metody `display`.

Wyobraźmy sobie teraz, że po napisaniu kodu przychodzi klient i mówi, że wprowadził do swojej kolekcji sprzedażowej gry z najniższego segmentu i chce dodać jeszcze *BasePrice*. Dodajemy zatem do naszego modelu UML jeszcze jedną klasę (rysunek 3.17).



Rysunek 3.17. Zmodyfikowany diagram klas

Przy tak opracowanym rozwiązaniu jedna rzecz może niepokoić. Jak już zostało wspomniane na samym początku tego rozdziału, w podrozdziale 3.2, nie powinniśmy powielać kodu (zasada DRY). Tu zaś mamy całkiem sporo zduplikowanych linijek. Co zrobić, aby tego uniknąć?

Właśnie w takich momentach na ratunek przychodzą wzorce projektowe.

Przykład 3.26

Analizę problemu należy rozpocząć od odpowiedzi na pytanie, z którego rodzaju wzorca powinniśmy skorzystać.

Na podstawie wymienionych w punkcie 3.6.2 krótkich charakterystyk grup wzorców możemy wywnioskować, że będziemy poszukiwać rozwiązania w grupie wzorców kreacyjnych. Dlaczego właśnie w tej?

Zadajmy sobie następujące pytania:

1. Czy problem dotyczy zależności między obiektami (wzorce behawioralne)? — Nie.
2. Czy problem dotyczy zależności między obiektami lub zarządzania obiektami (typ wzorca strukturalnego)? — Nie.
3. Czy problem dotyczy tworzenia nowego obiektu za każdym razem (typ wzorca kreacyjnego)? — Tak.

UWAGA

Przeglądając opisy poszczególnych wzorców z grupy wzorców kreacyjnych, trafiamy na następujący opis: „Określa interfejs do tworzenia obiektów, przy czym umożliwia podklasom wyznaczenie klasy danego obiektu. Metoda wytwórcza umożliwia klasom przekazanie procesu tworzenia egzemplarzy podklasom".

Jest to opis wzorca metoda wytwórcza (ang. *Factory Method*). Po uważnym, kilkakrotnym przeczytaniu zacytowanego zdania nasuwa się wniosek, że można wygenerować osobną klasę z metodą decydującą, na podstawie zadanego parametru, który typ instancji należy utworzyć.

Inna nazwa wzorca metoda wytwórcza to „konstruktor wirtualny" (ang. *Virtual Constructor*).

UWAGA

W przypadku źródeł dostępnych w internecie należy kierować się zasadą, że nie wszystko złoto, co się świeci. Nie zawsze atrakcyjny i przejrzysty interfejs idzie w parze z wysoką jakością przekazywanej wiedzy. Jeszcze raz podkreślamy, że warto sięgać tylko po sprawdzone i wiarygodne źródła informacji.

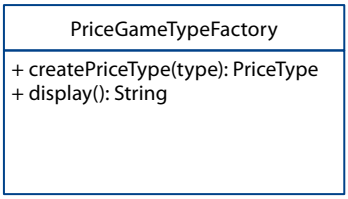
W praktyce na początku należy utworzyć klasę, która formalnie będzie nazywana fabryką. Zadania tej klasy to:

- wygenerowanie odpowiedniej instancji klasy według wskazanego typu,
- wyświetlenie danych dotyczących ceny (to ta część kodu jest zduplikowana).

Utwórzmy zatem klasę według schematu (rysunek 3.18). Na razie jednak nie będziemy pisać implementacji tworzenia instancji klas. Skupmy się na tym, by dobrze rozpisać klasę, implementacją zajmiemy się później (w kodzie brakujące elementy zastąpimy komentarzem).

UWAGA

Programiści, kiedy zostawiają sobie element kodu do napisania w późniejszym czasie, stosują do oznaczenia takich miejsc zapisaną w komentarzu sekwencję TODO (jeśli rozdzielimy to słowo po środkowej samogłosce „o", uzyskamy angielskie *to do*, czyli „do zrobienia").



Rysunek 3.18. Diagram UML klasy PriceGameTypeFactory utworzonej według wzorca metoda wytwórcza

Na listingu 3.12 przedstawiono kod, który powinniśmy napisać.

Listing 3.12

Definicja klasy PriceGameTypeFactory z pustymi metodami

```
class PriceGameTypeFactory {

    createGamePriceType = function(type) {
        let priceType;
        if (type === "baseprice") {
            // TODO stworzyć instancję BasePrice
        } else if (type === "midprice") {
            // TODO stworzyć instancję MidPrice
        } else if (type === "aaaprice") {
            // TODO stworzyć instancję AAAPrice
        }

        priceType.display = function() {
            // TODO metoda wyświetlająca komunikat
        }

        return priceType;
    }
}
```

Została utworzona klasa zawierająca jedną metodę, która w zależności od potrzeb zwróci konkretną instancję typu ceny.

Linijkę 3. możemy skrócić do równoznacznego zapisu:

```
createGamePriceType(type)
```

Linijkę 7. z listingu 3.12 należy zastąpić utworzeniem instancji klasy `BasePrice`. Uzyskamy to, wpisując:

```
priceType = new BasePrice()
```

Analogicznie postępujemy z pozostałymi instancjami do utworzenia.

Kolejnym krokiem jest „odchudzenie" już istniejących, wcześniej utworzonych klas (tych z listingu 3.10). Wyrzucamy z nich metodę `display`.

Skoro usunęliśmy metodę `display` z klas, to teraz trzeba ją gdzieś zaimplementować. Wcześniej stworzyliśmy sobie szkielet tej metody (listing 3.12, linijki 14.–16.). Ponieważ we wszystkich klasach ten kod był identyczny, wystarczy wkleić kod odpowiedzialny za wyświetlanie do linijki 15.

Co zatem wpisać w 15. linijce listingu 3.12? Metodę, którą usuwaliśmy z poszczególnych klas. Ciało tej metody to:

```
return console.log(`osoba odpowiedzialna za segment ${this.name}
to ${this.brandManager} (cena sugerowana to: ${this.price})`)
```

Tylko jedna rzecz wygląda inaczej niż poprzednio: w fabryce nie ma słowa kluczowego `this`, gdyż teraz za każdym razem tworzymy nową instancję *innej* klasy. Tym razem skorzystamy ze zmiennej, do której przypisywaliśmy nowo utworzone instancje, czyli `priceType` (listing 3.13).

Listing 3.13

Zmodyfikowany kod klasy PriceGameTypeFactory (uzupełniono metodę createGamePriceType oraz dodano metodę display)

```
class PriceGameTypeFactory {


    createGamePriceType = function(type) {
        let priceType;
        if (type === "baseprice") {
            priceType = new BasePrice();
        } else if (type === "midprice") {
            priceType = new MidPrice();
        } else if (type === "aaaprice") {
            priceType = new AAAPrice();
        }
```

```
        priceType.display = function() {
            return console.log(`osoba odpowiedzialna za segment ${this.
name} to ${this.brandManager} (cena sugerowana to: ${this.price})`);
        }
        return priceType;
    }
}
```

Teraz zmienimy definicję funkcji `gamesFactory`, tak jak pokazano na listingu 3.14.

Listing 3.14

Modyfikacja definicji funkcji gamesFactory

```
1  function gamesFactory() {
2      let games = [];
3      let gameFactory = new PriceGameTypeFactory();
4      let aaaPrice = gameFactory.createGamePriceType('aaaprice');
5
6      games.push(aaaPrice);
7      games.push(gameFactory.createGamePriceType('baseprice'));
8      games.push(gameFactory.createGamePriceType('midprice'));
9      games.push(gameFactory.createGamePriceType('baseprice'));
10
11     for (let i = 0; i < games.length; i += 1) {
12         console.log(games[i].display());
13     }
14     console.log(aaaPrice.publishers);
15 }
```

Przeanalizujmy wybrane linie kodu.

3. Tworzymy instancję klasy `PriceGameTypeFactory`.

4. Wykorzystując fabrykę, tworzymy instancję klasy `AAAPrice`.

6. Dodajemy instancję do tablicy.

12. Wykorzystujemy metodę `display` z klasy `PriceGameTypeFactory`.

3.7. Podsumowanie

Tworzenie wydajnej aplikacji o wysokiej jakości nie jest proste. Jeśli zapytamy architekta oprogramowania, ile czasu zajęło mu zdobycie niezbędnej wiedzy, zazwyczaj odpowie, że wciąż ją zdobywa. Dlaczego? Bo świat programowania to świat ciągłego kształcenia się. Bardzo często się okazuje, że rozwiązania stosowane w jednej firmie nie sprawdzają się w innej. Co więcej, technologia nie stoi w miejscu. Wciąż pojawiają się nowe frameworki, biblioteki, a nawet całe języki programowania. Musimy więc być przygotowani na nieustanne poszerzanie swojej wiedzy.

Niemniej zawsze trzeba zaczynać od opanowania podstaw. Bez tego po prostu nie uda się ruszyć z miejsca.

3.8. Zadania

Zadanie 3.1

Stwórz listę kroków potrzebnych do wysłania SMS-a.

Zadanie 3.2

Zaprezentuj za pomocą schematu blokowego listę kroków z zadania 3.1.

Zadanie 3.3

Zaprezentuj za pomocą schematu blokowego listę kroków potrzebnych do określenia, czy podana przez użytkownika liczba jest liczbą pierwszą.

Zadanie 3.4

Na podstawie istniejącego schematu blokowego (rysunek 3.6) napisz funkcję, która dla podanych przez użytkownika dwóch liczb wyznaczy ich największy wspólny dzielnik.

Zadanie 3.5

Na podstawie przedstawionego schematu blokowego (rysunek 3.8) napisz funkcję realizującą sortowanie bąbelkowe.

Zadanie 3.6

Napisz aplikację wyszukującą wskazany element w tablicy.

Zadanie 3.7

Zaprojektuj za pomocą diagramu UML rozwiązanie problemu sprzedaży warzyw w warzywniaku.

Zadanie 3.8

Zaprojektuj za pomocą diagramu UML rozwiązanie problemu sprzedaży komputera składanego ze wskazanych podzespołów.

Zadanie 3.9

Utwórz swój schemat blokowy opisujący wyciąganie mleka z lodówki. Jaka jest złożoność obliczeniowa tego schematu?

Zadanie 3.10

Zaimplementuj (na podstawie samodzielnie stworzonego schematu blokowego) metodę sprawdzającą, czy wskazana przez użytkownika liczba jest liczbą pierwszą. Jaka jest złożoność obliczeniowa tego algorytmu?

4

Testowanie oprogramowania

Współczesny świat opiera się na technologii komputerowej. Nie ma branży, która by funkcjonowała bez systemów informatycznych.

Zapewne większość z nas miała styczność z oprogramowaniem, które nie działało tak, jak powinno. Systemy są tworzone przez ludzi, a jak wiadomo, ludzie popełniają błędy.

Występujące w oprogramowaniu błędy prowadzą do awarii systemu. Każda awaria oznacza straty, zarówno finansowe, jak i wizerunkowe. Aby zminimalizować ryzyko jej wystąpienia, należy przeprowadzić testy wytwarzanego oprogramowania. Jest to szczególnie istotne, jeśli programista dba (a powinien!) o to, by użytkownicy mieli zaufanie do oferowanego przez niego produktu. Jeżeli jakość oferowanej usługi będzie niska, prawdopodobnie użytkownik nie dokona ponownego zakupu bądź nie poleci usługi znajomym. Dlatego nie wystarczy napisać kod aplikacji, by móc ją od razu udostępniać. Najpierw trzeba się upewnić, że jest ona wysokiej jakości i nie zawiera błędów. Uzyskamy to, testując oprogramowanie. Ten proces ma kluczowe znaczenie dla jakości produktu.

W kilku kolejnych podrozdziałach zostaną omówione podstawy testowania oprogramowania oraz dobre praktyki dotyczące zgłaszania błędów.

4.1. Siedem zasad testowania oprogramowania

Zanim przejdziemy do szczegółów, musimy poznać ogólne zasady dotyczące testowania. Jest ich siedem i są one solidnym fundamentem wiedzy zarówno dla początkującego, jak i dla doświadczonego testera. Warto, by znał je także programista — pomogą mu one tworzyć aplikacje wysokiej jakości. Niektóre z zasad mogą się wydawać oczywiste, jednak gdy się działa pod presją czasu, łatwo o nich zapomnieć. Niezależnie od charakteru projektu warto o nich pamiętać i do nich wracać.

1. Testowanie ujawnia usterki

Testowanie zmniejsza prawdopodobieństwo, że w oprogramowaniu pozostaną niezidentyfikowane defekty. Należy jednak pamiętać, że przeprowadzenie testów nie gwarantuje niezawodności aplikacji — nawet gdy żadne defekty nie zostaną znalezione, nie jest to dowód na poprawność oprogramowania. Niemal na pewno bowiem testy nie uwzględniły wszystkich możliwych przypadków. Jest to związane z tym, co głosi kolejna zasada.

2. Testowanie gruntowne jest niemożliwe

Przetestowanie wszystkich kombinacji danych wejściowych i warunków wstępnych jest możliwe tylko w prostych przypadkach. Gdybyśmy starali się przewidzieć i sprawdzić wszystkie możliwe przypadki, niepotrzebnie stracilibyśmy czas i być może nigdy nie zakończylibyśmy testowania. Dlatego zamiast dążyć do przetestowania całego systemu, powinniśmy ukierunkować wysiłki na analizę ryzyka (czyli na określenie prawdopodobieństwa wystąpienia poszczególnych błędów), dobór technik testowania (pod kątem najbardziej prawdopodobnych i najuciążliwszych typów błędów) i priorytetyzację (określenie, które błędy muszą zostać naprawione, a które — np. ze względu na niewielkie ryzyko ich wystąpienia lub nikłą uciążliwość dla użytkownika — są dopuszczalne).

3. Wczesne testowanie oszczędza czas i pieniądze

Czynności testowe powinny rozpoczynać się tak wcześnie, jak tylko to możliwe w przypadku danego oprogramowania. Im wcześniej zostanie wykryty błąd, tym niższy jest koszt jego naprawy.

UWAGA

Zwróć uwagę na to, ile kosztuje naprawa błędu na samym początku projektu (analiza), a ile już po jego wdrożeniu (tabela 4.1).

Tabela 4.1. Symulacja kosztu naprawy błędu na poszczególnych etapach wytwarzania oprogramowania

Etap	Koszt naprawy błędu
Analiza	0,2 MD*
Programowanie	0,5 MD
Testy modułowe	1 MD
Testy systemowe	3 MD
Po wdrożeniu	10 MD

* **MD (pracochłonność)** to ilość ciągłej nieprzerwanej pracy, jaką powinna wykonać jedna osoba, by skończyć zadanie. Jeśli zadanie musi wykonywać kilka osób, to sumuje się czas ich pracy, by uzyskać jeden wynik, tak jakby zadanie wykonywała jedna osoba. Pracochłonność mierzymy w dniach roboczych (ang. *man-day*).

Przykład 4.1

Wyobraź sobie, że piszesz program dla poważnej instytucji finansowej, np. dużego banku. Po ukończeniu pisania kodu nie wystarczyło Ci już czasu na rzetelne przetestowanie aplikacji, dlatego zamiast poprosić zleceniodawcę o zmianę terminu realizacji zlecenia, decydujesz się zaryzykować i przesłać program w obecnej postaci. Niestety zaraz po wdrożeniu produktu okazuje się, że niepoprawnie wylicza on raty kredytów. Kierownictwo banku jest — mówiąc oględnie — bardzo zdenerwowane. Kredytobiorcy dzwonią z reklamacjami. Sprawą zainteresowały się media. Żeby uniemożliwić zawieranie nowych umów z błędami w harmonogramach spłat kredytu, zdecydowano się zawiesić działanie systemu ratalnego. To oczywiście oznacza wymierne straty finansowe dla banku, nie wspominając nawet o utracie wizerunku instytucji. Ponieważ to Twój kod spowodował to zamieszanie, musisz go poprawić, tym razem jednak będziesz działać pod ogromną presją — kierownictwo banku żąda jak najszybszego usunięcia usterek, grożąc jednocześnie odstąpieniem od umowy i złożeniem pozwu sądowego. Ten przykład powinien uświadomić Ci, jak ważne jest testowanie produktu przed jego przekazaniem zamawiającemu.

4. Kumulowanie się błędów

Większość błędów znalezionych podczas testowania przed wypuszczeniem oprogramowania lub powodujących awarie produkcyjne znajduje się w małej liczbie modułów.

Możemy zaobserwować tutaj działanie tzw. zasady Pareto (80/20), czyli np.: na dziesięć testowanych funkcjonalności osiem zwykle działa prawidłowo, a dwie nieprawidłowo. Jeżeli jednak firma wprowadzi na rynek wadliwą aplikację, to zgodnie z zasadą Pareto 80% klientów zauważy te usterki i nie poleci produktu znajomym; może to nawet doprowadzić do odstąpienia od zakupu. Dlatego niezwykle ważne jest zlokalizowanie wadliwych modułów, które mogą być skupiskiem defektów aplikacji.

5. Paradoks pestycydów

Przy rozwijającej się aplikacji niezmieniane testy tracą z czasem zdolność do wykrywania defektów, podobnie jak pestycydy po pewnym czasie nie są zdolne do eliminowania szkodników. Ciągłe powtarzanie tych samych testów, bez zmieniania ich samych lub testowanych danych, prowadzi do sytuacji, w której przestają one w pewnym momencie wykrywać nowe błędy. Żeby przezwyciężyć paradoks pestycydów, przypadki testowe muszą być regularnie przeglądane i modyfikowane, podążając za rozwijaną aplikacją.

Przykład 4.2

Podczas testowania nowej strony internetowej dla firmy ubezpieczeniowej tester wyliczał składki ubezpieczenia OC, posługując się tylko jednym zestawem danych (tzn. za każdym razem, gdy przechodził do kalkulatora składki, wprowadzał te same dane). Jego testy nie wykazywały błędów. Dopiero podczas odbioru aplikacji przez klienta okazało się, że oprogramowanie zawiera takie usterki jak:

- brak walidacji pól „numer telefonu” i „adres mailowy” (w związku z czym można było np. wpisać mail bez znaku @ lub numer telefonu składający się z zaledwie dwóch cyfr);

- literówki w nazwach marek aut (przez które część użytkowników nie znajdowała na liście marek swoich samochodów);
- możliwość ubezpieczenia auta z datą startu z przeszłości (co jest niezgodne z obowiązującym prawem).

Gdyby testy odbywały się na zróżnicowanych danych, powyższe błędy zostałyby wykryte na początkowym etapie testów.

6. Testowanie zależy od kontekstu

W zupełnie inny sposób będzie testowany sklep internetowy niż system rezerwacji biletów czy aplikacja do zamawiania taksówki.

7. Przekonanie o braku błędów jest błędem

W świetle tego, co mówią zasady 1. i 2., nie należy oczekiwać, że jeśli oprogramowanie zostało gruntownie przetestowane i wyeliminowano znalezione dzięki temu błędy, na pewno jest już poprawne i będzie użyteczne dla korzystających z niego osób. Przede wszystkim wciąż może zawierać usterki, których nie udało się wykryć. Należy jednak także pamiętać, że nawet znalezienie i usunięcie wszystkich błędów na nic się nie zda, jeżeli system będzie nieintuicyjny i trudny w obsłudze albo nie będzie spełniał potrzeb oraz oczekiwań użytkownika.

Oczywiście nie znaczy to, że — skoro testowanie oprogramowania nie gwarantuje tego, że będzie ono bezbłędne i użyteczne — nie należy go przeprowadzać! Czas poświęcony na dobrze przemyślane i zaplanowane testowanie na pewno nie będzie czasem straconym.

4.2. Proces testowy według ISTQB

Znasz już ogólne zasady dotyczące testowania. Przejdźmy zatem do bardziej szczegółowych regulacji. Większość jest ustalanych przez **ISTQB**.

ISTQB (ang. *International Software Testing Qualifications Board*) jest instytucją wyznaczającą podstawowe standardy w dziedzinie testowania na skalę światową. Ta organizacja oferuje ścieżkę certyfikacyjną, która umożliwia zdobywanie certyfikatów potwierdzających uprawnienia dla testerów, analityków i test menedżerów. Jest uznawana na całym świecie w branży IT.

DEFINICJA

Zgodnie z definicją ISTQB **testowanie** to proces składający się z wszystkich czynności cyklu życia, zarówno statycznych (niezmiennych), jak i dynamicznych (zmieniających się w zależności od przekazanych do aplikacji danych). Jest skoncentrowany na planowaniu, przygotowaniu i ewaluacji oprogramowania oraz powiązanych produktów w celu określenia, czy spełniają one wyspecyfikowane wymagania, a także wykazania, że są one dopasowane do swoich celów i do wykrywania usterek.

Mówiąc prościej, testowanie oprogramowania to jeden z procesów zapewnienia jakości oprogramowania, który ma na celu weryfikację oraz walidację.

Ułożenie procesu, według którego będzie można testować oprogramowanie, jest kluczowym elementem każdej organizacji. Każdy proces jest dostosowany do struktury firmy, jej strategii, zasobów i metodyki, według której wytwarzane jest oprogramowanie.

Według ISTQB proces testowy składa się z następujących czynności:

1. Planowanie testów.
2. Monitorowanie testów i nadzór nad nimi.
3. Analiza testów.
4. Projektowanie testów.
5. Implementacja testów.
6. Wykonywanie testów.
7. Ukończenie testów.

W praktyce oznacza to, że tworząc projekt testowy, należy zaplanować kolejność prac i wziąć pod uwagę wszystkie czynności testowe, które będą realizowane. ISTQB nie narzuca organizacjom gotowego procesu, a jedynie pokazuje, na jakie grupy czynności warto zwrócić uwagę.

UWAGA

Nie wszystkie etapy będą miały odzwierciedlenie w realnych projektach. Niektóre czynności mogą występować równocześnie, nachodzić na siebie bądź zostać pominięte.

Przejdźmy teraz do przedstawienia poszczególnych procesów związanych z testowaniem.

4.2.1. Planowanie testów

Pierwszym etapem jest zaplanowanie wszystkich prac testowych w projekcie. Głównym dokumentem, w którym są przechowywane te informacje, jest plan testów. **Ten dokument jest aktualizowany przez cały cykl życia projektu.** Ma do niego wgląd cały zespół projektowy.

Opracowując plan testów, trzeba:

- określić zakres testów (wskazać, co będzie, a co nie będzie testowane);
- ustalić kryteria rozpoczęcia, zawieszenia i zakończenia testów (wskazać, co dokładnie musi się stać, żeby prace testowe przeszły do kolejnego etapu);
- określić cel testów (co chcemy osiągnąć poprzez testowanie);
- wskazać zasoby (skład zespołu projektowego, ale także sprzęt, który jest niezbędny do przeprowadzenia testów);
- wskazać środowiska, na których będą przeprowadzone testy;
- ustalić harmonogram (określić terminy, w których będą realizowane poszczególne prace testowe);

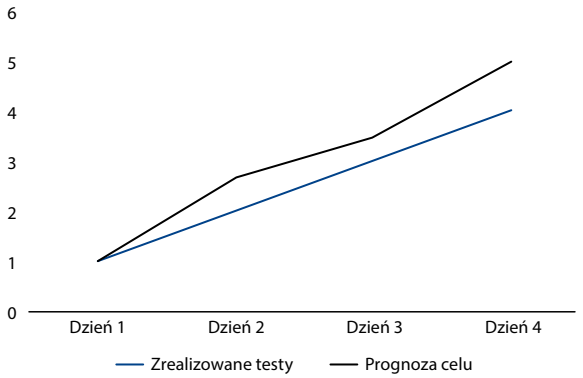
- zdefiniować ryzyka (zidentyfikować tzw. wąskie gardła, czyli obszary, etapy lub funkcjonalności, które mogą zagrozić terminowemu zakończeniu projektu);
- wskazać sposób bądź narzędzie, za pomocą którego będą rejestrowane i komunikowane zgłaszane błędy.

To tylko skrócona lista elementów planu testu. Ze względu na jego znaczenie oraz stopień skomplikowania będziemy jeszcze o nim mówić w rozdziale dotyczącym dokumentacji testowej.

4.2.2. Monitorowanie testów i nadzór nad nimi

Istotnym elementem prac jest ciągłe nadzorowanie przebiegu testów. Wszyscy członkowie zespołu projektowego oraz sponsorzy (czyli osoby, z których budżetów realizowane są projekty) są informowani o aktualnym stanie wykonawczym prac w odniesieniu do założonych celów.

W praktyce często stosowaną formą raportowania stanu prac jest wykres liniowy — jedna linia wskazuje aktualny stan zrealizowanych testów, druga pokazuje prognozę celu (co już powinno być wykonane) (rysunek 4.1).



Rysunek 4.1. Przykładowy wykres przedstawiający status zrealizowanych testów

Ważne, aby forma przekazywania raportu z przebiegu testów była zrozumiała dla wszystkich osób z zespołu projektowego.

4.2.3. Analiza testów

Jest to grupa czynności, których wykonanie zabiera najwięcej czasu, a ich wyniki są kluczowe dla dalszych etapów procesu testowego. Zespół testowy wykonuje analizę, dzięki której określa:

- podstawę testów;
- dane testowe, jakie będą potrzebne do ich wykonania;
- pokrycie, czyli warunki testowe (zespół wskazuje, co będzie testowane).

Członkowie zespołu korzystają głównie ze specyfikacji wymagań (ten dokument zostanie przedstawiony w rozdziale 5.), z dokumentacji technicznej lub założeń biznesowych.

UWAGA

Często już na etapie analizy zostają wychwycone błędy. Zgodnie z tym, co głosi znana Ci już zasada 3. testowania (o wczesnym testowaniu), przekłada się to na zmniejszenie kosztów poniesionych w związku z realizacją projektu. Na tym etapie można wyeliminować wszelkiego rodzaju:

- niejednoznaczności,
- pominięcia,
- niespójności,
- nieścisłości,
- sprzeczności,
- nadmiarowe (zbędne) instrukcje.

Powtórzmy raz jeszcze: im wcześniej wykryty błąd, tym niższy koszt jego naprawy.

4.2.4. Projektowanie testów

Gdy znane są już wnioski z etapu analizy, można przystąpić do projektowania testów. Na tym etapie należy przekształcić warunki testowe w przypadki testowe (wysokiego lub niskiego poziomu).

DEFINICJA

Warunek testowy to element lub zdarzenie modułu lub systemu, który może być zweryfikowany przez jeden lub więcej przypadków testowych, np. funkcja, transakcja, cecha, atrybut jakości lub element struktury. Zatem warunek testowy można uznać za artefakt nadrzędny w odniesieniu do przypadku testowego.

Mówiąc prostym językiem, warunek testowy określa, co powinno zostać sprawdzone podczas testów, a co za tym idzie, także co powinno zostać pokryte przypadkami testowymi.

DEFINICJA

Norma ISO 29119 określa **przypadek testowy** jako zestaw warunków wstępnych, danych wejściowych, akcji (w stosownych przypadkach), oczekiwanych rezultatów i warunków końcowych, opracowany na podstawie warunków testowych.

W praktyce taki przypadek testowy to instrukcja dla użytkownika, która mówi, jakie kroki należy wykonać i jakie są oczekiwane rezultaty wykonanych kroków.

Szczegółowa budowa przypadku testowego, zarówno wysokiego, jak i niskiego poziomu, będzie opisana w rozdziale 5.

Wszelkie **testalia**, czyli dokumentacja (przypadki testowe, historyjki użytkownika itp.) i skrypty wytworzone podczas trwania procesu testowego, powinny być przechowywane w narzędziu, które zapewnia wersjonowanie (historię edycji dokumentu) i śledzenie historii wykonania.

Warto korzystać z narzędzi wspomagających proces testowy, np. Jira czy TestLink (pierwsze z nich będzie omawiane w dalszej części rozdziału).

4.2.5. Implementacja testów

Na etapie implementacji (czyli wdrożenia, wprowadzenia) następuje sprawdzenie, czy posiada się wszystko, co pozwoli na skuteczne przeprowadzenie testów. Należy określić, jakie parametry powinno mieć środowisko testowe (np. z jakimi systemami ma być połączone, jaka ma być baza danych, jakie konta użytkowników powinny być utworzone), utworzyć skrypty automatyczne (jeżeli mają zastosowanie), uporządkować zestawy testów, wykonać priorytetyzację (zadecydować, które zadania bądź czynności będą wykonywane w pierwszej kolejności) itd.

UWAGA

Aby lepiej zrozumieć pojęcie priorytetyzacji, wykonaj zadanie 4.1.

4.2.6. Wykonanie testów

Podczas tego etapu wykonane zostają wszystkie zaplanowane testy, zarówno ręczne, jak i automatyczne (jeżeli projekt zakłada automatyzację). Jeżeli zostanie wychwycone niepożądane zachowanie systemu, należy zgłosić błąd (według wytycznych opisanych w planie testów).

Poprzez wykonanie testów należy też rozumieć zarejestrowanie prawidłowego wyniku testów (np. pozytywny, negatywny, zablokowany). Wszelka dokumentacja powstała w trakcie wykonania testów powinna być archiwizowana. W organizacjach o charakterze finansowym, bankowym czy instytucjach z sektora publicznego często są przeprowadzane kontrole (audyty) z przebiegu wytwarzania oprogramowania.

4.2.7. Ukończenie testów

Czynności zamykające testy mają miejsce w momencie osiągnięcia zamierzonego celu, jakim może być np. zakończenie 80% przypadków testowych z wynikiem pozytywnym, przekazanie oprogramowania dla grupy użytkowników, wydanie produktu do masowej sprzedaży.

Należy sprawdzić, czy dane w raportach są zgodne ze stanem faktycznym (np. analizując rejestr defektów). Następnie trzeba opracować końcowy raport z wykonanych przypadków testowych i przekazać go zespołowi projektowemu.

Jeżeli wytworzona aplikacja będzie udostępniona nowej grupie użytkowników, warto sporządzić instrukcję korzystania z aplikacji (tzw. instrukcję użytkownika/*manual*). Ważne, aby taki dokument zawierał zrzuty ekranów wraz z opisem kroków, które należy wykonać.

W praktyce często na etapie ukończenia testów organizowane są spotkania, na których zespół podsumowuje swoje działania i wyznacza obszary wymagające poprawy przy okazji dalszych prac.

4.3. Poziomy testów

Znasz już ogólne zasady dotyczące przeprowadzania testów. W poprzednim podrozdziale omówiliśmy zalecane przez ISTQB etapy, z których składać się powinien proces testowania. Na każdym z tych etapów wykonuje się pewne specyficzne dla niego czynności. Tworzą one tzw. **poziomy testów**.

DEFINICJA

Poziomy testów to określone czynności testowe wykonywane na danym etapie wytwarzania oprogramowania.

Na każdym poziomie testów określa się kolejno następujące zagadnienia — nazwijmy je obszarami testowania:

- przedmiot testów (to, co jest przedmiotem testowania);
- cel (to, co chcemy osiągnąć);
- podstawę testów (to, na czym będziemy testować);
- typowe defekty (nieprawidłowości, jakich szukamy na danym poziomie);
- preferowane środowisko testowe (warunki, jakie muszą być spełnione, aby wykonane testy były miarodajne).

Ważne jest, aby ustalać wyżej wymienione elementy w takiej właśnie kolejności — trudno będzie wskazać cel testów, jeśli nie jesteśmy świadomi, co jest ich przedmiotem. Oprócz tego musimy zdawać sobie sprawę z ich specyfiki.

W tym podrozdziale poznasz cztery poziomy testów wraz z charakterystycznymi dla nich cechami:

- testy modułowe (testy jednostkowe),
- testy integracyjne,
- testy systemowe,
- testy akceptacyjne.

4.3.1. Testy modułowe

Jak sama nazwa wskazuje, testy modułowe nie dotyczą całej aplikacji, lecz skupiają się na jej poszczególnych elementach, które można przetestować oddzielnie, w oderwaniu od reszty aplikacji — na modułach.

Przykład 4.3

Wyobraź sobie pojedyncze wagony pociągu. Zanim zostaną podłączone do lokomotywy, muszą zostać sprawdzone. Każdy taki wagon to moduł. Aby pociąg dotarł na czas do punktu docelowego, wszystkie wagony muszą być sprawne.

Ćwiczenie 4.1

Spróbuj określić obszary testowania — przedmiot, cel, podstawę, typowe defekty i preferowane środowisko testowe dla modułów, którymi są wagony pociągu.

W przypadku testowania oprogramowania na poziomie testów modułowych wymienione wcześniej obszary testowania zazwyczaj kształtują się następująco.

Przedmiot: moduły, jednostki, komponenty aplikacji, moduły baz danych.

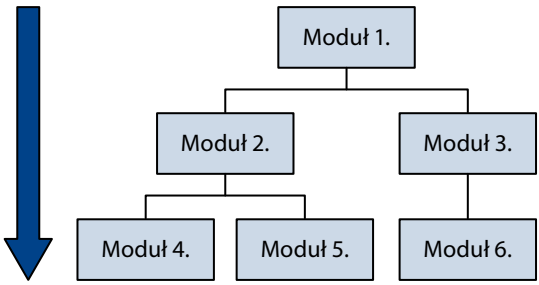
Cel: wykrycie defektów w module, zablokowanie przedostania się defektów do powiązanych modułów.

Podstawa: kod, projekt szczegółowy, specyfikacja techniczna modułu.

Typowe defekty: niepoprawna logika, martwy kod, niepoprawna funkcjonalność.

Preferowane środowisko testowe: programistyczne.

Istnieje kilka wypracowanych podejść do testów modułowych. Obok prezentujemy przykładowe podejście ***top-down*** zastosowane do testów modułowych (rysunek 4.2).



Rysunek 4.2.
Podejście typu top-down do testów modułowych

***Top-down* (od góry do dołu)**

Moduły znajdujące się na najwyższym poziomie są testowane jako pierwsze. Moduły będące w hierarchii poniżej są zastępowane/symulowane przez zaślepki. Testowane moduły są używane do testowania niżej położonych komponentów. Proces testowy jest kontynuowany do momentu przetestowania komponentów znajdujących się na najniższym poziomie.

4.3.2. Testy integracyjne

Testy integracyjne są ukierunkowane na znajdowanie błędów w interfejsach na styku poszczególnych zintegrowanych elementów — głównie usterek dotyczących wymiany danych pomiędzy nimi.

Przykład 4.4

Kontynuując nasz przykład z pociągiem, możemy założyć, że przeprowadziliśmy już testy pojedynczych modułów — wagonów. Wszystkie testy przebiegły prawidłowo. Czy to oznacza, że pociąg może już ruszać w drogę? Oczywiście nie. Teraz musimy sprawdzić, czy wagony są trwale i bezpiecznie ze sobą połączone, w szczególności — czy pierwszy z nich jest poprawnie podłączony do lokomotywy. W ten sposób badamy ich integrację.

Ćwiczenie 4.2

Spróbuj określić obszary testowania umożliwiające sprawdzenie integracji modułów pociągu — wagonów.

Przykład 4.5

Innym dobrym przykładem testowania zintegrowanych elementów jest sprawdzanie poprawności przeprowadzenia transakcji finansowych.

Co się dzieje, kiedy wykonujesz przelew bankowy? Z Twojego konta odejmowana jest pewna kwota pieniędzy. Ale to nie wszystko. Przecież ta suma nie może tak po prostu zniknąć. Musi się pojawić na innym koncie — tym, na które dokonujesz przelewu. Aby mieć pewność, że transakcja została przeprowadzona poprawnie, nie wystarczy więc sprawdzić stan konta, z którego wykonany został przelew. Trzeba też się upewnić, że pieniądze dotarły na wskazany rachunek docelowy.

Spróbujmy określić, jak mogłyby wyglądać obszary testowania w przypadku testów integracyjnych.

Przedmiot: podsystemy, infrastruktura, interfejsy, interfejsy programowania aplikacji (ang. *Application Programming Interface*, API).

Cel: budowanie zaufania do interfejsów, wykrycie błędów występujących w poszczególnych modułach, szczególnie na ich styku.

Podstawa: projekt oprogramowania, przepływy pracy (tzn. sposób przepływu informacji pomiędzy rozmaitymi obiektami biorącymi udział w jej przetwarzaniu), przypadki użycia (czyli interakcje z całym systemem lub jego podsystemem, prowadzące do pewnego konkretnego rezultatu), specyfikacja wymagań.

Typowe defekty: błędy w komunikacji pomiędzy modułami, niezgodność interfejsów, niepoprawne bądź brakujące dane.

Preferowane środowisko testowe: niewielka ilość zintegrowanych modułów (im mniej, tym lepiej).

Niepoprawna integracja modułów aplikacji jest często przyczyną błędów. Wystarczy, że dane generowane w jednym module nie docierają do drugiego lub docierają doń zniekształcone. Co zrobić, by zminimalizować ryzyko takich usterek? Pomocne w tym może być stosowanie **podstawowych zasad integracji**:

- poszczególne moduły/systemy powinny być integrowane tylko raz;
- kolejność i terminy integracji kolejnych elementów powinny być udokumentowane;
- należy integrować możliwie małą liczbę modułów w jednym czasie;
- nie powinno się integrować więcej niż dwóch systemów naraz;
- powinno się dążyć do minimalizacji nakładów związanych z budową zaślepek i sterowników.

DEFINICJA

Zaślepka to szkieletowa albo specjalna implementacja modułu używana podczas produkcji lub testowania innego modułu, który tę zaślepkę wywołuje bądź jest w inny sposób od niej zależny.

DEFINICJA

Sterownik to program lub narzędzie testowe używane do uruchamiania oprogramowania w celu wykonania zestawu przypadków testowych.

4.3.3. Testy systemowe

Testy systemowe skupiają się na całościowym działaniu systemu z uwzględnieniem każdej funkcjonalności.

Przykład 4.6

Zbadaliśmy już wagony, upewniliśmy się także, że połączenia pomiędzy nimi są poprawne, trwałe i bezpieczne. To jednak wciąż za wcześnie, by wpuścić do pociągu

pasażerów. Najpierw powinniśmy przeprowadzić jazdę próbną, by sprawdzić, jak cały pociąg zachowuje się na torach. Jak reaguje na zakręty, szczególnie te bardziej ostre? Czy nie ma ryzyka wykolejenia? Jaka jest jego droga hamowania przy różnych prędkościach?

Ćwiczenie 4.3

Jak zdefiniować obszary testowania pod kątem sprawdzenia poprawności działania pociągu jako całości?

W przypadku testów systemowych aplikacji obszary testowania mogłyby wyglądać następująco.

Przedmiot: aplikacje, systemy operacyjne, system podlegający testowaniu.

Cel: działanie całego systemu, walidacja zgodności zachowań funkcjonalnych i niefunkcjonalnych systemu z projektem i specyfikacją wymagań, budowanie zaufania do jakości produktu.

Podstawa: specyfikacja wymagań, przypadki użycia, historyjki użytkownika.

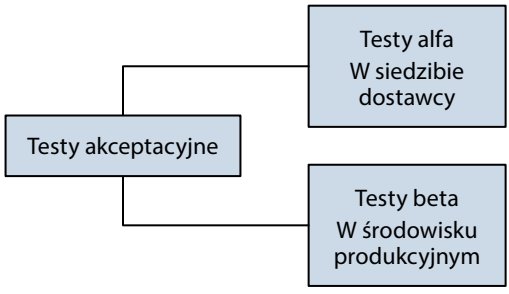
Typowe defekty: niepoprawne lub nieoczekiwane zachowania funkcjonalne lub niefunkcjonalne systemu, problemy z prawidłowym i kompletnym wykonywaniem całościowych zadań funkcjonalnych, problemy z prawidłowym działaniem systemu w środowisku (środowiskach) testów systemowych, niezgodność działania systemu z opisami zawartymi w instrukcji użytkownika.

Preferowane środowisko testowe: w pełni zintegrowane środowisko testowe. Powinno odzwierciedlać docelowe środowisko produkcyjne (takie, w którym użytkownicy będą korzystać z systemu).

4.3.4. Testy akceptacyjne

Testy akceptacyjne, podobnie jak systemowe, skupiają się na całościowym działaniu systemu z uwzględnieniem każdej funkcjonalności. Odgrywają one decydującą rolę w kwestii wdrożenia rozwiązania na docelowe środowisko produkcyjne. Testy akceptacyjne zostały podzielone na dwie kategorie: alfa i beta (rysunek 4.3). Czynniki wpływające na podział to:

- miejsce wykonywania testów,
- środowisko testów,
- profil działalności testerów.



Rysunek 4.3.
Podział testów akceptacyjnych

Testowanie akceptacyjne alfa to symulowane lub rzeczywiste testy produkcyjne przeprowadzane u producenta bez

udziału wytwórców oprogramowania. Testy alfa przeprowadza się w środowisku jak najbardziej zbliżonym do środowiska docelowego, przed wypuszczeniem produktu na rynek, w chwili gdy jest on już dostatecznie stabilny, zazwyczaj w siedzibie producenta oprogramowania. Testy te wykonywane są przez potencjalnych użytkowników, a nie przez twórców produktu.

Testowanie akceptacyjne beta to testy produkcyjne przeprowadzane w środowisku niezwiązanym z twórcami oprogramowania. Testy beta wykonywane są głównie w przypadku systemów na dużą skalę, gdy nie ma jednego konkretnego odbiorcy oprogramowania, a liczba ewentualnych środowisk docelowych jest duża.

Wykonywane jest przez potencjalnych użytkowników, nie przez twórców produktu.

Z reguły po pomyślnym przeprowadzeniu testów akceptacyjnych beta dochodzi do akceptacji (stąd nazwa) produktu i finalizacji zlecenia.

Poniżej wymieniamy przykładowe obszary testowania oprogramowania dla testów akceptacyjnych.

Przedmiot: aplikacje, systemy operacyjne, system podlegający testowaniu.

Cel: walidacja i weryfikacja zgodności zachowań funkcjonalnych i niefunkcjonalnych systemu z projektem i specyfikacją wymagań, budowanie zaufania do jakości produktu.

Podstawa: specyfikacja wymagań, procesy biznesowe, przypadki użycia, historyjki użytkownika, wymogi prawne.

Typowe defekty: niepoprawnie zaimplementowane reguły biznesowe, niepoprawne lub nieoczekiwane zachowania funkcjonalne lub niefunkcjonalne systemu, niezgodność działania systemu z opisami zawartymi w instrukcji użytkownika, niedostateczna wydajność, słabe zabezpieczenia.

Preferowane środowisko testowe: w pełni zintegrowane środowisko testowe. Powinno odzwierciedlać docelowe środowisko produkcyjne.

4.4. Typy testów

W poprzednim podrozdziale omówiliśmy poziomy testowania zależne od tego, czego dotyczą testy — podstawowych modułów (testy modułowe), ich integracji (testy integracyjne), całości aplikacji czy systemu (testy systemowe) czy wreszcie zachowania systemu w środowisku, w którym będzie pracował na co dzień — u zamawiającego oprogramowanie klienta, na jego komputerach, w połączeniu z jego bazami danych (testy akceptacyjne). Obecnie zajmiemy się typami testów.

Typy testów wyznaczane są przez grupę dynamicznych czynności testowych, które mają wspólny określony cel. Do takich celów należą m.in.:

- przetestowanie danej funkcjonalności;
- przetestowanie wybranych właściwości, takich jak użyteczność, niezawodność, przenaszalność;
- wykonanie testów architektury systemu;
- weryfikacja naprawionych defektów (testy potwierdzające);
- poszukiwanie niezamierzonych zmian (testy regresji).

DEFINICJA

Re-test (powtórzenie testu, test potwierdzający) to powtórne wykonanie testu, którego pierwotne wykonanie wykazało awarię (nieprawidłowe działanie). Celem re-testu jest sprawdzenie, czy defekt będący przyczyną awarii został usunięty.

DEFINICJA

Testy regresywne (regresji) to ponowne przetestowanie uprzednio testowanego programu po dokonaniu w nim modyfikacji, w celu upewnienia się, że w wyniku zmian nie powstały nowe błędy.

W poniższych punktach przedstawimy typy testów oraz wyjaśnimy podstawowe techniki, które na co dzień są stosowane w zespołach testowych.

4.4.1. Testy funkcjonalne

Testy funkcjonalne polegają na wykonaniu testów systemu, dzięki którym sprawdzamy zgodność wykonanej zmiany z wymaganiami.

Wymagania funkcjonalne mogą być udokumentowane w:

- specyfikacji wymagań funkcjonalnych,
- historyjkach użytkownika,
- przypadkach użycia.

Dopuszczalny jest również tryb nieudokumentowany, ale warto zaznaczyć, że sprawdza się on tylko w zespołach o wysoko rozwiniętej umiejętności komunikacji. Testy funkcjonalne należy wykonywać na wszystkich poziomach testów, z uwzględnieniem podstawy i przedmiotu każdego z nich. Testy funkcjonalne mogą być wspierane przez wyspecjalizowane w tym celu narzędzia, umożliwiające automatyzację testów — mówimy w takim przypadku o testach automatycznych.

Testy automatyczne pozwalają skutecznie przyspieszyć cykl testowania i poprawić jakość dostarczanego oprogramowania. Dzięki objęciu automatyzacją testów regresyjnych działy jakości są w stanie wykazać wysokie wyniki przy niewielkim obciążeniu osób testujących. W praktyce oznacza to zwiększenie stopnia pokrycia testami dzięki rozszerzeniu automatyzacji o te części aplikacji, które mogły nie być dokładnie testowane w poprzednich wersjach.

Automatyzacja testów pozwala na:

- wykonanie szybkich testów w celu potwierdzenia, że nowo dodana funkcjonalność nie wywołała błędów w niezmienianej części systemu;
- wykonanie testów po naprawieniu błędu występującego w czasie pracy aplikacji w środowisku testowym;
- sprawdzenie stabilności i działania systemu w trybie 24/7 (nieprzerwanego działania 24 godziny przez 7 dni w tygodniu).

UWAGA

Dlaczego w obecnych czasach tak wiele firm dąży do automatyzacji testów?

Kierują się kosztami, jakie ponoszą na testy regresji (im więcej systemów w firmie, tym bardziej czasochłonne są testy regresji).

Testy białoskrzynkowe i czarnoskrzynkowe

W zapewnianiu jakości oprogramowania kluczowe znaczenie ma dobre przygotowanie i zaplanowanie testów. Niezbędna jest do tego wiedza o technikach testowania. Rozróżnienie poniższych definicji pomoże Ci odpowiednio dopasować testy do prac w danym projekcie. Inaczej będą planowane prace do projektu, w którym masz do dyspozycji pełną dokumentację projektową oraz testera z wiedzą techniczną, a inaczej w projekcie bez pełnej dokumentacji oraz z testerami biznesowymi (użytkownikami aplikacji, bez wiedzy na temat tworzenia programów komputerowych).

DEFINICJA

Testy czarnoskrzynkowe (funkcjonalne) to technika projektowania przypadków testowych, w której te przypadki są projektowane bez zaglądania w wewnętrzne mechanizmy działania modułu (bez znajomości kodu). Mogą być realizowane przez użytkowników aplikacji, którzy nie muszą posiadać (i najczęściej nie posiadają) wiedzy technicznej. Ludzie ci, użytkując aplikację na co dzień, mogą być jednak nieocenionym źródłem informacji na temat tego, czy rzeczywiście działa ona niezawodnie, czy jest intuicyjna i czy ma wszystkie potrzebne im funkcje.

DEFINICJA

Testy białoskrzynkowe (strukturalne, oparte na strukturze) to technika projektowania przypadków testowych na podstawie analizy struktury modułu. Najczęściej wykonywane przez doświadczonych testerów lub programistów. Wymagają wiedzy technicznej — znajomości składni języka, w którym napisano program, umiejętności tworzenia i analizy kodu, obeznania z dobrymi zasadami programistycznymi itp.

4.4.2. Testy niefunkcjonalne

Testy niefunkcjonalne, jak sama nazwa sugeruje, mają za zadanie sprawdzić wszystkie atrybuty niefunkcjonalne, czyli:

- wydajność (testy wydajności, obciążeniowe i przeciążające),
- użyteczność,
- niezawodność,
- efektywność,
- przenaszalność,
- zdolność wprowadzania zmian w przyszłości.

WSKAZÓWKA

Testowanie niefunkcjonalne można przeprowadzać na wszystkich poziomach testów.

W tabeli 4.2 znajduje się zestawienie cech testów odpowiadających za sprawdzenie atrybutu wydajności.

Tabela 4.2. Cechy testów wydajnościowych, przeciążeniowych i obciążeniowych

Testy wydajnościowe	Testy przeciążeniowe	Testy obciążeniowe
• Badanie czasu odpowiedzi krytycznych dla biznesu funkcji systemu • Porównywanie czasu odpowiedzi przejścia jednego i wielu użytkowników przez aplikację • Kryterium testów jest sprawdzenie, czy poszczególne akcje wykonywane są przez aplikację w akceptowalnym czasie	• Badanie zachowania aplikacji w przypadku jednoczesnego korzystania z niej przez zbyt wielu użytkowników, z wykorzystaniem zbyt wielu danych, przy malejących zasobach systemowych • Test, czy system „zawiedzie” w oczekiwany sposób • Wyszukiwanie defektów w aplikacji działającej w trybie awaryjnym • Sprawdzanie konsekwencji utraty danych po awarii wywołanej nadmiernym obciążeniem	• Badanie zachowania aplikacji przy jednoczesnym korzystaniu z niej przez dużą, ale dopuszczalną (w przeciwieństwie do testów przeciążeniowych) liczbę użytkowników, przy dużej liczbie przeprowadzanych transakcji czy operacji • Utrzymanie takiego stanu przez określony w scenariuszu czas • Sprawdzenie, jak wiele zapytań (ang. *requests*) jest w stanie obsłużyć system w określonym przedziale czasu

4.5. Dobre praktyki w zgłaszaniu błędów za pomocą narzędzi

Jednym z celów testowania jest znajdowanie defektów (błędów). Kluczowy jest sposób ich zgłoszenia. Jeżeli mamy do dyspozycji dedykowane narzędzie, to naszym obowiązkiem jest dbanie o to, aby wszystkie zgłoszenia były rejestrowane. Dzięki pełnemu rejestrowi błędów możemy mierzyć jakość wytwarzanego oprogramowania.

DEFINICJA

Defekt (błąd) jest też czasem nazywany usterką i jest jednym z najczęściej używanych przez testera pojęć w codziennej pracy. Defekt to wada modułu lub systemu, która może spowodować, że moduł lub system nie wykona zakładanej czynności. Ta wada najczęściej wynika z popełnionej przez programistę pomyłki, polegającej np. na użyciu niepoprawnego polecenia lub wyrażenia bądź niewłaściwej definicji danych. Ostatecznym skutkiem jest awaria oprogramowania (rysunek 4.4).

Rysunek 4.4. Mechanizm powstawania awarii

WSKAZÓWKA

Oto obszary, w których najczęściej możemy znaleźć błędy:

- skomplikowany kod, rozbudowana funkcjonalność (np. wykorzystywana w kilku systemach);
- nowo wprowadzone zmiany (regresja);
- zmiany, podczas wprowadzania których programista pracował pod presją czasu.

Przy rejestrowaniu zgłoszenia każde narzędzie wymaga różnego zestawu informacji. W zgłoszeniu takim powinny się znaleźć:

- nazwa — powinna w krótki i zwięzły sposób, hasłowo informować o tym, czego dotyczy błąd;
- unikatowy identyfikator (ID) — przydzielany przez system;
- opis problemu — powinien zawierać dokładny zapis (krok po kroku) wykonywanych operacji, które spowodowały błąd, oraz opis błędnego działania skonfrontowany z poprawnym, oczekiwanym zachowaniem;
- załączniki — zrzut ekranu, arkusz, dokument, link itp.;
- login bądź imię i nazwisko osoby zgłaszającej;

- priorytet — informujący o tym, jak ważny jest błąd i jak szybko powinien zostać naprawiony;
- waga — informująca o tym, jaki wpływ na inne funkcjonalności ma raportowany błąd;
- środowisko, w którym wykryto błąd (przeglądarka, rola użytkownika, link do środowiska testowego).

W dalszej części rozdziału zostały przedstawione trzy narzędzia, które są wykorzystywane w komercyjnych projektach. Każde z nich jest cały czas rozbudowywane i wzbogacane o kolejne funkcje. Omówimy ich istotne cechy pod kątem zgłaszania błędów.

Jira

Jira jest obecnie najpopularniejszym narzędziem wykorzystywanym komercyjnie. Swoją popularność zawdzięcza przede wszystkim elastyczności oraz możliwości dostosowania ekranów do potrzeb projektu. Producent programu opracowuje i udostępnia na bieżąco wiele dodatków, które można w pełni integrować z tym narzędziem.

Na rysunku 4.5 widać przykład zadania do realizacji zaplanowanego za pomocą narzędzia Jira.

Rysunek 4.5. Widok na zadania w Jira

Najpopularniejsze pola, z jakimi możesz się spotkać w Jira podczas zgłaszania błędu, to:

- ***Project*** — nazwa projektu w Jira — czasem się zdarza, że mamy kilka do wyboru, jeśli pracujemy w paru projektach jednocześnie;
- ***Issue Type*** — typ zgłoszenia, np. *Bug* — błąd, *Improvement* — zgłoszenie dotyczące nowych zmian w systemie, *Documentation* — zgłoszenie dotyczące tworzenia nowej dokumentacji w projekcie;

- ***Summary*** — tytuł zgłoszenia;
- ***Priority*** — priorytet; rozróżniamy następujące priorytety zgłoszeń:
  - ***Blocker*** — np. aplikacja nie działa, blokuje się kluczowy proces i nie ma sposobu na obejście problemu;
  - ***Critical*** — np. kluczowy proces nie działa, ale jest obejście (nieergonomiczne) problemu ze wsparciem IT;
  - ***Major*** — np. błąd powodujący spowolnienie procesu, a jednocześnie błąd, dla którego istnieje obejście;
  - ***Minor*** — np. błąd, który nie ma wpływu na działanie procesu biznesowego, ale jego naprawa poprawi ergonomię pracy;
  - ***Trivial*** — np. mało znaczący, łatwy do naprawienia błąd, jak zły napis lub literówka;
- ***Affect/Fix Version*** — oznaczenie wersji systemu, w której znaleziono błąd, lub tej, w której zadanie ma zostać zrealizowane;
- ***Assignee*** — pole określające osobę, która ma się zająć realizacją zadania (domyślnie uzupełnione jest wartością *Automatic* — w takim przypadku zadanie automatycznie zostanie przypisane do osoby wskazanej w konfiguracji projektu w Jira; często jest to *Project Owner*);
- ***Environment*** — to pole wskazuje, w którym środowisku testowym zaobserwowane zostało zdarzenie, którego dotyczy zgłoszenie;
- ***Description*** — pole pozwalające na umieszczenie w nim pełnego opisu zgłoszenia;
- ***Original Estimate*** — umożliwia określenie czasu na realizację zadania;
- ***Attachment*** — daje możliwość dodania załączników do zgłoszenia.

Bugzilla

Kolejne warte wspomnienia narzędzie to Bugzilla. Pomimo dość prostego interfejsu wciąż chętnie się je wykorzystuje w projektach. Jest darmowym i otwartym oprogramowaniem służącym do raportowania błędów. Może być stosowane do aplikacji komercyjnych.

Poniżej zamieszczamy zrzut ekranu prezentujący przykładowy błąd zgłoszony w Bugzilli (rysunek 4.6). Nie ma potrzeby zagłębiać się w szczegóły na nim widoczne. Większość pól występuje również w Jira i została już omówiona.

Podczas raportowania błędów mamy możliwość zgłaszania własnych sugestii związanych z rozwojem oprogramowania (ang. *enhancement* — ulepszenia i porady).

Rysunek 4.6. Widok na szczegóły zadania w Bugzilli

Trello

Ostatnie z prezentowanych narzędzi to Trello. Jest „najmłodszym" narzędziem z tej grupy. Ma bardzo szerokie zastosowanie i co najważniejsze — daje użytkownikowi całkowitą dowolność. Nie ma zdefiniowanych pól. Niezbędne informacje można wprowadzać na dodawanych w razie potrzeby kartach. Wystarczy znać zbiór dobrych praktyk, aby narzędzie w niczym nie odbiegało od Jira.

Poniżej zamieszczamy zrzut ekranu aplikacji Trello (rysunek 4.7). Ponieważ nie wymusza ona na użytkowniku wypełnienia określonych pól, cały ciężar przekazania informacji o błędzie spoczywa na zgłaszającym.

Rysunek 4.7. Widok na zadania w aplikacji Trello

4.6. Zadania

Zadanie 4.1

Część I (10 min)

Wyobraź sobie, że przydzielono Cię do nowego projektu. Twoim zadaniem będzie przetestowanie nowej poczty elektronicznej. Wypisz podstawowe funkcjonalności, które Twoim zdaniem muszą zostać sprawdzone.

Część II (5 min)

Poinformowano Cię, że ze względu na przedłużający się czas wytwarzania oprogramowania, aby nie przekroczyć terminu oddania aplikacji klientowi, harmonogram testów zostanie znacznie skrócony. Zmusza Cię to do wybrania tylko trzech kluczowych funkcjonalności, które będą testowane. Wskaż trzy wybrane funkcjonalności programu pocztowego.

Zadanie 4.2

Wymień siedem zasad testowania i omów dwie z nich bardziej szczegółowo.

Zadanie 4.3

Wymień grupy czynności, z których składa się proces testowy.

Zadanie 4.4

Z czego składa się plan testów?

Zadanie 4.5

Z czego składa się przypadek testowy?

Zadanie 4.6

Wskaż różnice pomiędzy testami funkcjonalnymi a niefunkcjonalnymi.

Zadanie 4.7

Jakie są podstawowe cele testów akceptacyjnych?

Zadanie 4.8

Podaj różnicę pomiędzy testami regresji a re-testem.

Zadanie 4.9

Jaka jest różnica pomiędzy testami czarnoskrzynkowymi a białoskrzynkowymi?

Zadanie 4.10

Czym różnią się testy przeciążeniowe od testów obciążeniowych?

Zadanie 4.11

Podaj definicję defektu.

Zadanie 4.12

Jakie informacje powinno zawierać poprawne zgłoszenie błędu?

5

Tworzenie dokumentacji testowej

Dokumentacja wytworzona podczas procesu testowego jest niezbędna do ewentualnego odtworzenia danego etapu testów czy przekazania wiedzy nowym członkom zespołu. Stanowi też swego rodzaju środek komunikacji — nie zawsze cały zespół projektowy przebywa w jednej siedzibie. Wszelkie ustalenia i zmiany są umieszczane we wskazanych lokalizacjach (takich jak miejsce na dysku firmowym lub dedykowane narzędzie, do którego prowadzi wskazany link). Dokumentacja powinna być tworzona według wzorców przyjętych i obowiązujących w danej organizacji.

Przykład 5.1

Wyobraź sobie, że przystępujesz do projektu w trakcie jego trwania jako tester bądź programista. Nikt nie ma czasu wprowadzać Cię w każdy aspekt czynności, które będziesz wykonywać. Twoim podstawowym źródłem wiedzy będzie wszelkiego rodzaju dokumentacja. To dzięki niej będziesz wiedzieć, jak zgłaszać błędy oraz do jakich narzędzi będziesz potrzebować dostępów.

W tym rozdziale poznasz przykładową dokumentację testową, zasady jej tworzenia, jej budowę oraz najważniejsze elementy.

5.1. Plan testów

Jak już zostało powiedziane w rozdziale 4., plan testów (ang. *test plan*) jest tworzony na etapie planowania procesu testowego (ang. *test process*).

Ten dokument jest aktywny przez cały czas trwania projektu. Powinien być każdorazowo modyfikowany i uzupełniany o nowe informacje. Wszystkie osoby zaangażowane w projekt muszą mieć dostęp do dokumentu.

WSKAZÓWKA

Nie należy się spodziewać, że ze względu na pewien stały szkielet plan testów będzie wyglądał tak samo w różnych organizacjach czy nawet w różnych projektach w obrębie jednej organizacji. Przeciwnie, zazwyczaj da się wskazać istotne niejednolitości w konstrukcji planu. Wynika to z odmiennego podejścia do projektów, specyficznych wymogów prawnych dla niektórych branż, a nawet ze struktury organizacyjnej firmy.

Poniżej zaprezentowano schemat przykładowego planu testów, który może być wykorzystywany w komercyjnych projektach.

Plan testów projektu [Nazwa]

I Cel dokumentu

W tym punkcie należy opisać cel stworzenia dokumentu, a także wskazać projekt, którego dotyczy.

II Przedmiot i zakres

Należy wskazać, co jest celem testów, oraz wymienić elementy, które będą testowane. W tym punkcie może się znaleźć np. opis informujący o tym, że zostaną sprawdzone interfejsy i nowe funkcjonalności, procesy biznesowe; można także zamieścić informację o tym, czy przeprowadzone będą testy regresji (jeśli tak, należy podać ich zakres).

III Terminologia

Zawiera wykaz definicji pojęć użytych przy tworzeniu dokumentu, np.:

DEFINICJA

Smoke testy (nazywane również testami dymnymi) — podzbiór wszystkich zdefiniowanych/zaplanowanych przypadków testowych, które pokrywają główne funkcjonalności modułu lub systemu. Mają na celu potwierdzenie, że działają kluczowe funkcjonalności programu, bez zagłębiania się w szczegóły.

Testy eksploracyjne (ang. *exploration tests*) — testy oprogramowania wykonywane bez wykorzystania przypadków testowych. Mają jasno zdefiniowany obszar weryfikacyjny oraz trwają określony kwant czasu (np. od jednej do czterech godzin). Są uzupełnieniem testów opartych na scenariuszach, natomiast nie mogą ich zastępować.

Raport o postępie testów (ang. *test progress report*) — dokument zawierający podsumowanie aktywności testowych i osiągniętych wyników, tworzony regularnie, by zaraportować postęp prac testowych w stosunku do założeń, a także przedstawiający ryzyka i alternatywy wymagające podjęcia decyzji zarządczych.

UWAGA

Zwróć uwagę, że często osoby biorące udział w projekcie nie mają wiedzy z zakresu wytwarzania oprogramowania, dlatego warto tłumaczyć nawet te pojęcia, które dla Ciebie są oczywiste.

Zachowaj kolejność alfabetyczną — dzięki temu osoba czytająca dokumentację szybciej odnajdzie poszukiwany termin.

IV Obowiązki, odpowiedzialność i uprawnienia

Zawiera opis ról wraz z zakresem obowiązków przydzielonych do osób, które biorą udział w tworzeniu specyfikacji funkcjonalnej zmiany informatycznej.

W ramach poszczególnych etapów testów w naszym przykładowym planie do realizacji zadań przydzielone zostały osoby wymienione w tabeli 5.1.

Tabela 5.1. Przydział osób do realizacji testów na poszczególnych etapach

ID	Imię i nazwisko	Rola/odpowiedzialność/zakres zadań
1.	Anna Anonimowa	Project manager — zarządzanie projektem, kontakt z dostawcą zewnętrznym oprogramowania
2.	Krzysztof Null	Analityk biznesowy (ang. *business analyst*) — analiza błędów, przygotowanie danych do testów
3.	Jan Statystyczny	Koordynator testów (ang. *test coordinator*) — zarządzanie testami, informowanie zespołu o niedostępności środowiska

V Opis postępowania

1. Informacje wstępne

a) Dokumenty powiązane

W tabeli 5.2 należy wskazać wszystkie dokumenty, które są związane z projektem (harmonogram, zestawienie wymagań dotyczących danych, zestawienie wymagań dotyczących środowiska itp.).

Tabela 5.2. Wykaz dokumentów dotyczących projektu

Id	Dokument (symbol, nazwa)	Data ostatniej modyfikacji	Wersja dokumentu	Autor dokumentu
1.	Specyfikacja wymagań funkcjonalnych	01.03.2020	v.2	Anna Anonimowa
2.	Plik z danymi do importu	07.03.2020	v.3	Krzysztof Null
3.	Harmonogram prac	09.03.2020	v.4	Anna Anonimowa

b) Założenia i ograniczenia

Należy wymienić, jakie typy testów będą przeprowadzone, np. weryfikacji gotowości kodu, konfiguracji oprogramowania, migracji danych, regresji, akceptacyjne, po wdrożeniu na produkcję (*smoke testy*). Trzeba też opisać, jak będą realizowane zależności od innych projektów.

Należy także wskazać na powiązania z komponentami, które mogą mieć wpływ na testowany projekt, np. poprzez wspólne środowisko testowe.

- Zakres testów

Należy wskazać zakres testów zgodnie z dokumentacją projektową i założeniami projektu.

- Rodzaje przeprowadzanych testów (przykładowe — tabela 5.3).

Tabela 5.3. Rodzaje przeprowadzanych testów

Rodzaj testów	Planowane (TAK/NIE)	Uwagi
Smoke testy	TAK	Zespoły operacyjne wyznaczyły trzy osoby, które przeprowadzą testy w siedzibie firmy
Testy jednostkowe (ang. *unit tests*)	NIE	Testy będą wykonane po stronie dostawcy oprogramowania
Testy akceptacyjne (ang. *user acceptance tests*, UAT)	TAK	
Testy funkcjonalne (ang. *functional tests*)	TAK	
Testy integracyjne (ang. *integration tests*)	TAK	Należy skontaktować się z dostawcą oprogramowania i ustalić datę testów
Testy regresywne (ang. *regression tests*)	TAK	
Testy systemowe (ang. *system tests*)	TAK	

c) Dokumentacja realizowania testów

Potwierdzenia wyników testów zostaną umieszczone w narzędziu **TestLink** w postaci zrzutów ekranów ilustrujących poszczególne kroki scenariusza uzupełnionych tam, gdzie to możliwe, o identyfikator odpowiadający wykonywanemu testowi (numer zamówienia, identyfikator klienta itp.). Do każdego scenariusza zakończonego ze statusem *Zablokowany* (ang. *blocked*) zostanie dołączony e-mail lub inny dokument uzasadniający wykluczenie scenariusza z zakresu testów.

d) Potrzeby szkoleniowe

Przykładowo warsztaty z architektami i kierownikami projektów. Dotyczy to wszystkich projektów w ramach ścieżki wdrożeniowej. Oprócz tego warto przeprowadzić warsztaty dla pracowników obsługi klienta, na których to warsztatach zaprezentujemy im nowe funkcje aplikacji, nowe ekrany itp.

e) Kryteria rozpoczęcia, zawieszenia i zakończenia testów

Należy opisać kryteria rozpoczęcia, zawieszenia i zakończenia testów.

Przykład 5.2

Kryteria rozpoczęcia testów: przygotowane środowisko testowe, zanonimizowana baza danych (dane osobowe są przekształcone, niemożliwe do odczytu pierwotnej wartości).

Kryteria zawieszenia testów: niedostępność środowiska testowego przypadająca na okres dłuższy niż jeden dzień roboczy, brak osoby do wykonania testów akceptacyjnych.

Kryteria zakończenia testów: brak błędów krytycznych, wykonanie 100% przypadków testowych.

2. Ryzyka

Należy opisać potencjalne ryzyka:

a) projektowe — niedostępność środowiska, opóźnienie realizacji testów wynikające z nieterminowej realizacji działań zaplanowanych w harmonogramie (np. niedostarczony na czas kod aplikacji), złej jakości dane testowe, niedostępność — wynikająca z urlopów, zwolnień itp. — osób kluczowych dla testów;

b) produktowe — złej jakości kod, nieaktualna dokumentacja, luki analityczne, zmieniające się wymagania itp.;

c) bezpieczeństwo informacji — wyciek danych osobowych, możliwość wejścia do panelu klienta bez podania loginu i hasła, brak autoryzacji dostępu do danych wrażliwych.

3. Wymagania dotyczące danych, środowiska i narzędzi testowych

a) wymagania dotyczące danych testowych — dane klientów posiadających przynajmniej dwa aktywne produkty, dane zanonimizowane itp.;

b) wymagania dotyczące środowiska testowego — środowisko powinno być zintegrowane z systemem głównym, baza danych musi być odświeżona na dzień 02.03.2020, wymagana możliwość zmiany daty systemowej itp.;

c) wymagania dotyczące narzędzi testowych — dostępność Jira i TestLink dla osób wyznaczonych do testów, generator danych itp.

4. Harmonogram testów (tabela 5.4)

Tabela 5.4. Harmonogram testów

Nazwa zadania	Data rozpoczęcia	Data zakończenia	Odpowiedzialny
Testy akceptacyjne	10.03.2020	20.03.2020	Jan Statystyczny — koordynator testów
Smoke testy	22.03.2020	22.03.2020	Anna Anonimowa — project manager

5. Raportowanie i komunikacja

a) Produkty dostarczane w wyniku testów (testalia) — tabela 5.5

Tabela 5.5. Produkty testów i osoby za nie odpowiedzialne

Nazwa produktu	Osoba odpowiedzialna
Raport z testów akceptacyjnych	Koordynator testów (Jan Statystyczny)
Raport defektów (ang. *defect report*)	Koordynator testów (Jan Statystyczny)

b) Plan raportowania i komunikacji

Należy opisać sposób raportowania oraz ustalony proces komunikacji, np. „Realizacja testów będzie na bieżąco raportowana w narzędziu TestLink".

c) Zespół projektowy (tabela 5.6)

Tabela 5.6. Skład zespołu projektowego

ID	Imię i nazwisko	Rola	Sposób komunikacji
1.	Anna Anonimowa	Project manager	Poczta elektroniczna, telekonferencja MS Teams Cykliczne spotkania (każda środa o godzinie 10:00)
2.	Krzysztof Null	Analityk biznesowy	Poczta elektroniczna, telekonferencja MS Teams Cykliczne spotkania (każda środa o godzinie 10:00)
3.	Jan Statystyczny	Koordynator testów	Poczta elektroniczna, telekonferencja MS Teams Cykliczne spotkania (każda środa o godzinie 10:00)
4.	Wojciech Defekt	Programista	Poczta elektroniczna, telekonferencja MS Teams Cykliczne spotkania (każda środa o godzinie 10:00)

6. Załączniki (tabela 5.7)

Tabela 5.7. Załączniki do planu testów

Id	Dokument (symbol, nazwa)	Odnośnik do rozdziału planu	Data ostatniej modyfikacji	Wersja dokumentu
1.	Przypadki testowe zapisane w pliku *.xls*	5.2	02.03.2020	v.2

5.2. Scenariusze testowe

W dobie ciągłego monitorowania wykonywanej pracy oraz pomiaru jej jakości scenariusze testowe są pewnego rodzaju dowodem, że podczas testów system zachował się w określony sposób. Ta wiedza jest niezbędna dla wielu grup:

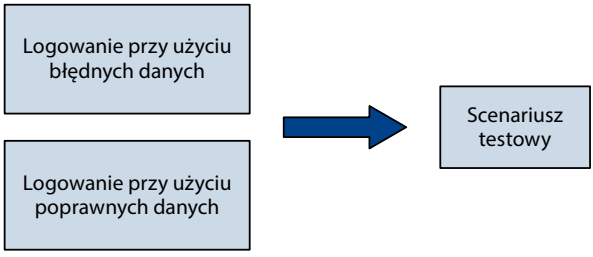
- programistów,
- testerów,
- zamawiających rozwiązania systemowe.

Na początku wyjaśnimy, czym jest scenariusz testowy, a następnie przedstawimy różnicę pomiędzy scenariuszem testowym a przypadkiem testowym.

DEFINICJA

Scenariusz testowy (ang. *test scenario*) — zbiór przypadków testowych, czyli kroków służących do sprawdzenia danej funkcjonalności systemu.

Zaprezentowany na rysunku 5.1 przykład obrazuje budowę scenariusza testowego dla funkcjonalności znanej każdemu z nas.



Rysunek 5.1. Schemat budowy scenariusza testowego

Przykład 5.3

Funkcjonalność: logowanie do systemu.

W tej funkcjonalności należy utworzyć osobne przypadki dla poprawnego i niepoprawnego logowania (tzn. przy użyciu poprawnych i błędnych danych).

Scenariusze testowe mogą być wykonywane przez osoby, które nie miały styczności z pracą w dziale IT. Podczas testów akceptacyjnych często o wykonanie pewnych czynności testowych proszone są osoby pracujące w działach operacyjnych (np. obsługa klienta, sprzedaż), które nie mają wiedzy o projekcie ani umiejętności technicznych. Im dokładniejszy scenariusz, tym mniej pytań od takich osób. Jeżeli testy będą wykonywane przez osoby z wewnątrz organizacji, możemy sobie pozwolić na przypadki wysokiego poziomu. Zaoszczędzamy wtedy czas na szczegółowe rozpisywanie każdego z kroków.

Przykład 5.4

W tabeli 5.8 prezentujemy przypadki niskiego i wysokiego poziomu dla naszego przykładu dotyczącego logowania.

Tabela 5.8. Przypadki niskiego i wysokiego poziomu

Nazwa	Przypadek niskiego poziomu	Przypadek wysokiego poziomu
Tytuł	**Logowanie przy użyciu poprawnych danych**	**Poprawne logowanie**
Cel testu	**Celem testu jest sprawdzenie poprawności działania mechanizmu logowania**	**Weryfikacja mechanizmu logowania**
Warunki wstępne	1. Podano prawidłowy login i hasło dla użytkownika testowego 2. Aplikacja została uruchomiona i znajdujemy się na ekranie logowania	1. Podany jest prawidłowy login i hasło dla użytkownika testowego 2. Aplikacja została uruchomiona i znajdujemy się na ekranie logowania
Kroki do wykonania	1. Wprowadź nieprawidłowy login i prawidłowe hasło, po czym kliknij *Zaloguj* 2. Wprowadź prawidłowy login i nie wprowadzaj hasła; kliknij *Zaloguj* 3. Wprowadź prawidłowy login i nieprawidłowe hasło, po czym kliknij *Zaloguj* 4. Wprowadź prawidłowy login i prawidłowe hasło, po czym kliknij *Zaloguj*	Zaloguj się do systemu (zastosuj różne warianty prawidłowych/nieprawidłowych danych)

→

Nazwa	Przypadek niskiego poziomu	Przypadek wysokiego poziomu
Oczekiwany wynik	1. Pojawia się komunikat o wprowadzonym nieistniejącym loginie, wyświetlana jest także prośba o ponowne wpisanie loginu 2. Pojawia się komunikat informujący o brakujących danych; po jego wyświetleniu powinno nastąpić przejście do pola, w którym wprowadza się hasło 3. Pojawia się komunikat informujący o podaniu nieprawidłowego hasła, zostaje także wyświetlona prośba o ponowne wpisanie hasła 4. Użytkownik zostaje zalogowany do systemu	1. Błędny login, logowanie nie powiodło się 2. Logowanie nie powiodło się. Wprowadź hasło 3. Błędne hasło, logowanie nie powiodło się 4. Następuje prawidłowe zalogowanie do aplikacji
Środowisko	Próby logowania należy wykonać w: • środowisko Web — przeglądarki Chrome, Edge, Opera • środowisko mobilne — Android, iOS	Próby logowania należy wykonać w środowiskach dedykowanych aplikacji

Warto zwrócić uwagę na to, że przypadki wysokiego poziomu są bardziej ogólne i w przeciwieństwie do przypadków niskiego poziomu nie wymieniają szczegółowo wszystkich sytuacji, jakie mogą nastąpić podczas próby logowania. Jest to spowodowane tym, że takie przypadki są realizowane przez osoby, które posiadają niezbędną wiedzę i są świadome czynności, jakie trzeba przeprowadzić, aby pomyślnie zakończyć testy.

5.2.1. Tworzenie scenariuszy testowych w aplikacji TestLink — instrukcja użytkownika

W tym punkcie zostanie zaprezentowane krok po kroku, jak tworzyć scenariusze w aplikacji TestLink. Oczywiście — zależnie od używanej wersji aplikacji — ekrany mogą się nieznacznie różnić od tych, które tu zamieszczono; jednak kluczowe pola będą takie same. Opis będzie mieć charakter instrukcji użytkownika. W takiej formie często prezentuje się docelowej grupie użytkowników nową funkcjonalność lub całą aplikację.

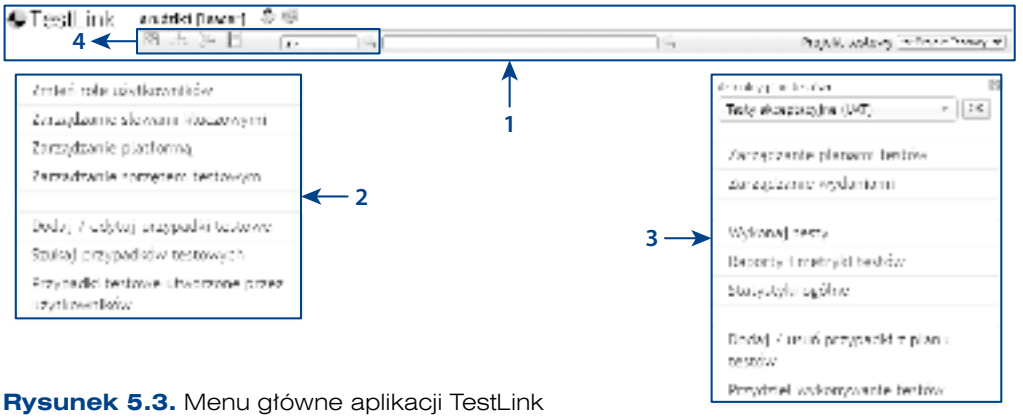
5.2.1.1. Zaloguj się do aplikacji

Po uruchomieniu aplikacji TestLink (oczywiście musimy mieć ją wcześniej zainstalowaną) pojawi się ekran logowania (widoczny na rysunku 5.2). Aby zalogować się do niej, należy podać login oraz hasło użytkownika utworzone wcześniej przez administratora.

Rysunek 5.2. Ekran logowania do aplikacji TestLink

5.2.1.2. Menu główne

Użytkownik po zalogowaniu się zostanie przeniesiony na stronę główną narzędzia TestLink (rysunek 5.3).



Rysunek 5.3. Menu główne aplikacji TestLink

Okno składa się z czterech głównych części:

- menu górne (1),
- menu lewe (2),
- menu prawe (3),
- panel użytkownika (4).

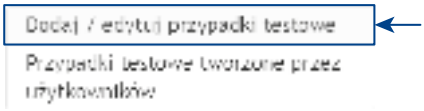
Opcje wyświetlane w *menu górnym* (1) uzależnione są od praw przydzielonych zalogowanemu użytkownikowi, są jednak dostępne niezależnie od wybranej sekcji i projektu. Możemy tam zobaczyć aktualnie zalogowanego użytkownika, przyciski *Ustawienia konta* i *Wyloguj*. Poniżej (z prawej strony) wyświetlana jest lista z aktualnie wybranym projektem testowym (na rysunku jest to *_tst:Projekt Testowy*). Po kliknięciu na strzałce z prawej strony listy rozwinie się pasek z aktualnymi projektami.

Menu lewe (2) pozwala tworzyć przypadki, zarządzać projektami, użytkownikami, rolami i wymaganiami. Przechowuje listę plików z dokumentacją.

Menu prawe (3) wyświetla aktualny plan testów i pozwala na jego zmianę. Z tego poziomu możesz zarządzać projektami testów, wybranym planem testów oraz samymi testami.

5.2.1.3. Specyfikacja testowa — Dodaj/edytuj przypadki testowe

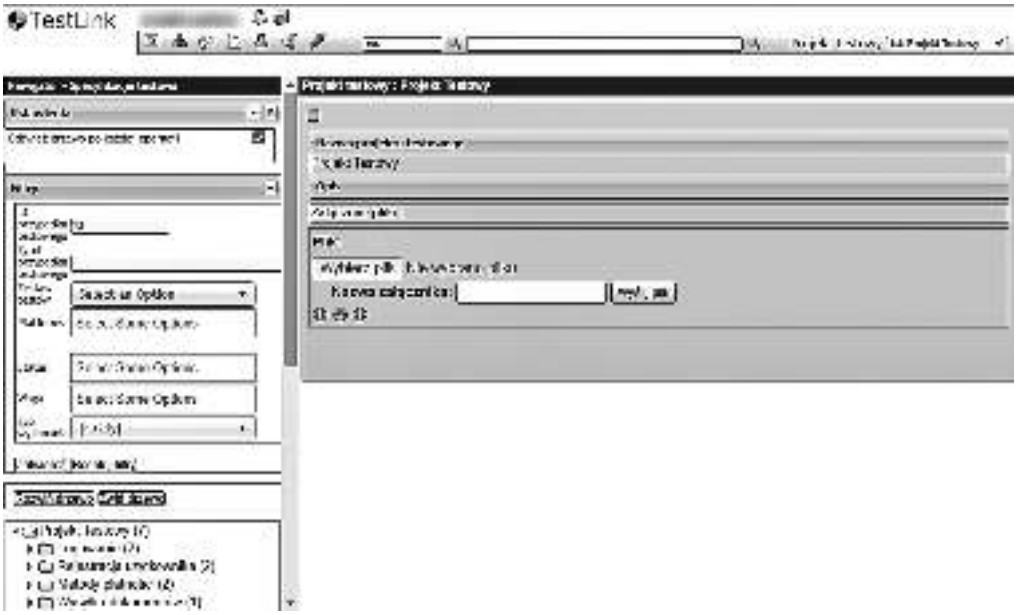
Po kliknięciu w tym miejscu możesz zarówno tworzyć nowe przypadki, jak i edytować już istniejące (rysunek 5.4).



Rysunek 5.4. Dodaj/edytuj przypadki testowe

5.2.1.4. Wybór zestawu testów

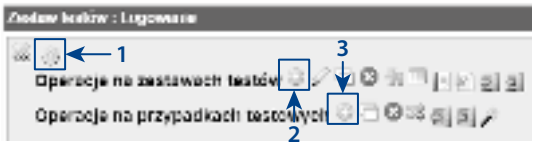
Wybierz folder, w którym chcesz utworzyć nowe przypadki testowe — drzewo folderów znajdziesz po lewej stronie ekranu (rysunek 5.5).



Rysunek 5.5. Zestawy testów

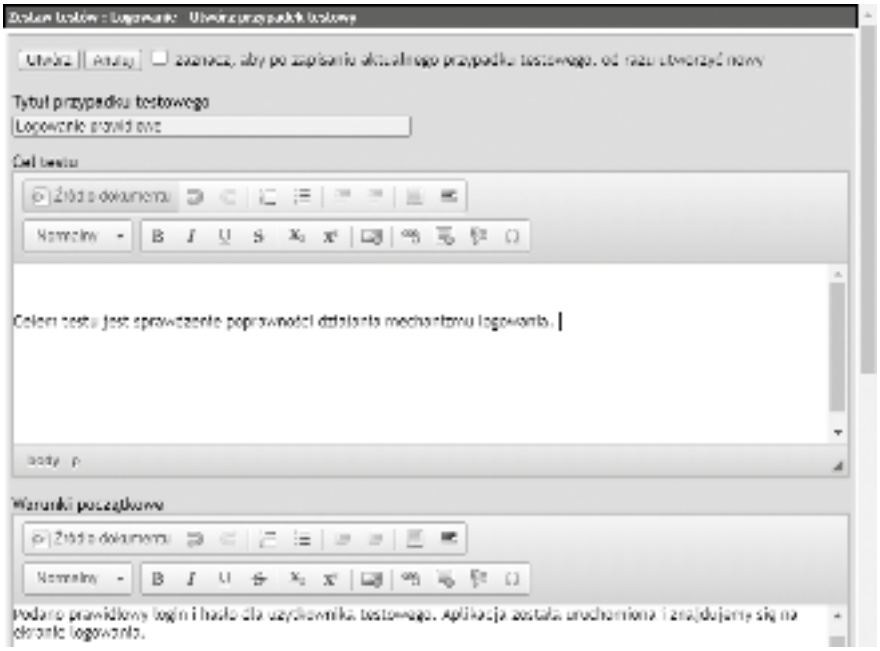
5.2.1.5. Wybór operacji

W prawej części ekranu wybierz ikonę *Ustawienia* (1). Masz przed sobą dwa wiersze możliwych do wykonania operacji (rysunek 5.6) — na zestawach testów (2) i na przypadkach testowych (3).



Rysunek 5.6. Dostępne operacje na zestawach i przypadkach testowych

Wybierz operację dodawania przypadku testowego (2). Pojawi się ekran przypadku testowego (rysunek 5.7).



Rysunek 5.7. Tworzenie przypadku testowego

Następnie wypełnij pola:

- *Tytuł przypadku testowego* (nazwa musi jasno wskazywać, co jest przedmiotem testu dla tego przypadku).
- *Cel testu* (informacja o tym, co chcesz osiągnąć, wykonując ten przypadek).
- *Warunki początkowe* (jakie warunki muszą być spełnione, abyś mógł przystąpić do testu).

Kliknij przycisk *Utwórz*. Następnie wybierz *Utwórz krok* i wprowadź treść przypadku.

UWAGA

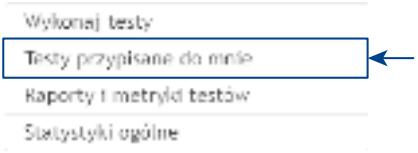
Zastanów się, czy będziesz pisać przypadek wysokiego, czy niskiego poziomu. Pamiętaj, że osoba, która ma wykonywać zaprojektowany przez Ciebie przypadek, powinna wiedzieć, co ma robić, na podstawie opisanych przez Ciebie kroków.

5.2.2. Wykonanie scenariuszy testowych — instrukcja użytkownika

Zakładając, że scenariusze testowe zostały utworzone, przejdziemy teraz do wykonywania testów. Zanim to jednak nastąpi, osoba zarządzająca testami, która ma odpowiednie uprawnienia w narzędziu TestLink, przydzieli Ci konkretne przypadki.

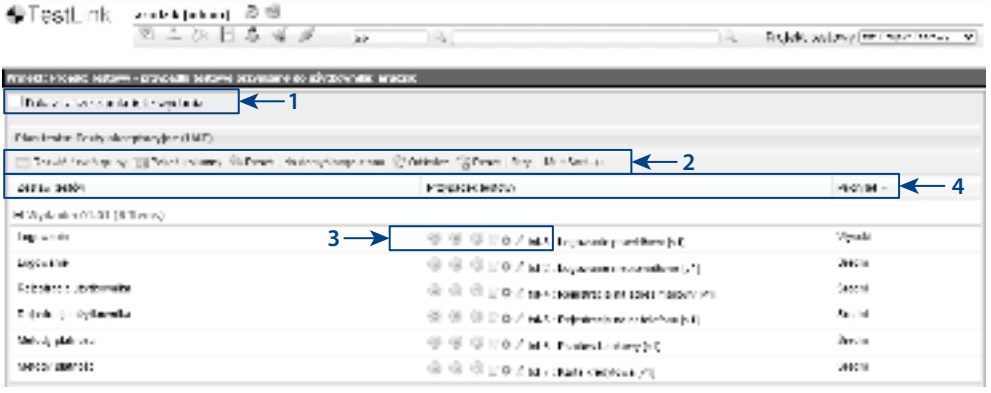
5.2.2.1. Testy przypisane do mnie

W prawym menu wybierz opcję wskazaną na rysunku 5.8.



Rysunek 5.8. Uruchomienie ekranu z listą przydzielonych przypadków

Teraz jesteś na ekranie z listą przypadków testowych do wykonania (rysunek 5.9).



Rysunek 5.9. Dostępne operacje do przeprowadzenia na przypadkach testowych

Zaznaczenie opcji *Pokaż także zamknięte wydania* (1) spowoduje wyświetlenie zestawów testów dla zamkniętych już planów testów.

Na rysunku 5.9 numerem 2 oznaczono *Panel zarządzania*. Składa się on z opcji:

- *Rozwiń/zwiń grupy* — rozwija/zwija zestawy testów;
- *Pokaż kolumny* — powoduje wyświetlenie wszystkich kolumn w tabeli;
- *Resetuj do domyślnego stanu* — resetuje do domyślnego stanu;
- *Odśwież* — odświeża tabelę;
- *Resetuj filtry* — resetuje filtry zawężające zakres wyświetlanych danych;
- *MultiSort* — sortuje po wielu kolumnach jednocześnie; aby użyć tej opcji, złap i przeciągnij wybrane nagłówki na pasek narzędzi.

Panel przypadku testowego (oznaczony na rysunku 5.9 numerem 3) pozwala zarządzać przypadkiem testowym. Oto dostępne na nim opcje (kolejno od lewej):

- kliknij, aby ustawić przypadek testowy na pozytywny (zakończony z pozytywnym wynikiem);
- kliknij, aby ustawić przypadek testowy na negatywny (taki, w przypadku którego nie udało się uzyskać pozytywnego efektu);
- kliknij, aby ustawić przypadek testowy na zablokowany (taki, którego realizacja jest w danym momencie z jakiegoś powodu niemożliwa);
- kliknij, aby wyświetlić historię przypadku testowego (zobaczyć przebieg zmian i uwagi dotyczące jego wykonania);
- kliknij, aby wejść w specyfikację przypadku testowego.

Numerem 4 na rysunku 5.9 oznaczono mechanizm umożliwiający filtrowanie listy, pozwalający na jej sortowanie według kryteriów:

- *Zestaw testów*,
- *Przypadek testowy*,
- *Priorytet* (właśnie tę opcję widać na rysunku),
- *Status*,
- *Przypadek testowy przypisany od wskazanej daty*.

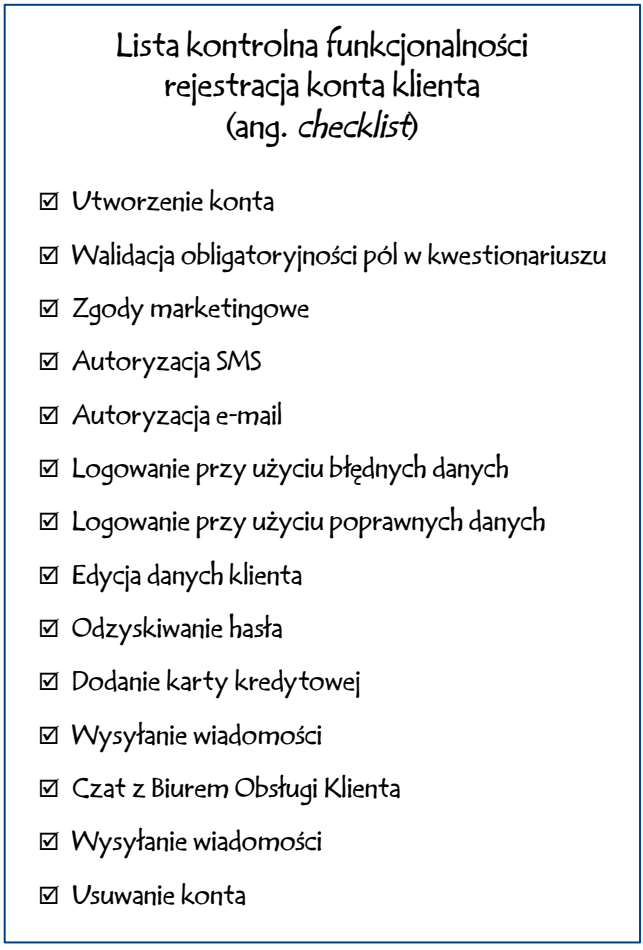
5.3. Lista kontrolna

Lista kontrolna (ang. *checklist*) to narzędzie umożliwiające kontrolę poprawności i/lub ocenę stopnia ukończenia danego przedsięwzięcia. Lista składa się z serii pytań lub zagadnień dotyczących projektu. Skuteczność listy zależy od jej złożoności. Im bardziej szczegółowe są zagadnienia lub pytania, tym większa jest jej skuteczność.

DEFINICJA

W praktyce często odchodzi się od szczegółowych list kontrolnych ze względu na brak czasu. Niekiedy firma programistyczna realizuje nawet kilkadziesiąt projektów w jednym czasie, a te same osoby są zaangażowane w kilka z nich. Nie jest to dobra praktyka. Zapamiętaj, że czas poświęcony na sporządzenie szczegółowej listy uchroni Cię przed zbędnymi i czasochłonnymi pracami na późniejszym etapie projektu.

Na rysunku 5.10 zamieszczono przykład listy kontrolnej do mechanizmu rejestracji konta klienta. Oznaczenia przy poszczególnych polach wskazują na to, że wszystkie przypadki zostały zrealizowane.



Rysunek 5.10. Przykładowa lista kontrolna

5.4. Rejestr ryzyk

Rejestr ryzyk jest dokumentem wytwarzanym we wczesnym stadium projektu, zazwyczaj dostarczają go osoby odpowiedzialne za zarządzanie projektem, tzw. kierownicy projektu.

Rejestr ma głównie na celu zwrócenie uwagi wszystkich zaangażowanych osób na czynniki, które mogą być zagrożeniem dla terminowej realizacji projektu, jak również dla jakości wytwarzanego produktu. Dokument pozwala przygotować się na potencjalne ryzyka oraz wcześnie na nie reagować. Przykładowy rejestr pokazano w tabeli 5.9.

Tabela 5.9. Przykładowa budowa rejestru ryzyk

ID ryzyka	Autor	Opis ryzyka	Data zgłoszenia	Kategoria ryzyka	Wpływ ryzyka	Priorytet	Podjęte działania
1.	Jakub Typowy	Nieaktualna specyfikacja wymagań funkcjonalnych	10.03.2020	Produktowe	Duży	Wysoki	Spotkanie z pomysłodawcami i ustalenie odpowiedzialności
2.	Anna Anonimowa	Realizacja projektu w sezonie urlopowym	05.03.2020	Projektowe	Duży	Wysoki	Wyznaczenie obszarów, w których należy zapewnić dostępność pracowników w określonym czasie

5.5. Raport błędów

W czasie testów zgłaszanych jest wiele błędów — niektóre z nich są naprawiane od razu, inne zostają przełożone do realizacji w kolejnych wersjach programu. Raport błędów jest narzędziem o charakterze informacyjnym — pomaga ocenić jakość wytwarzanego rozwiązania i czas naprawy przez zespół programistów.

Taki raport powinien zawierać niezbędne informacje (tabela 5.10), które dadzą zainteresowanym pełny obraz aktualnej sytuacji, bez konieczności logowania się do narzędzia zarządzającego błędami, np. Jira.

Tabela 5.10. Przykładowa budowa raportu błędów

ID defektu	Status	Aplikacja	Priorytet	Komponent	Data zgłoszenia	Osoba przypisana	Planowana data naprawy
ID-10 Błąd przy próbie logowania	Otwarty	XXZ	Wysoki	222	04.03.2020	Wojciech Defekt	10.03.2020
ID-23 Error 500 przy próbie uruchomienia aplikacji	Do wyjaśnienia	XXZ	Wysoki	222	05.03.2020	Adam Naprawczy	13.03.2020

→

ID defektu	Status	Aplikacja	Priorytet	Komponent	Data zgłoszenia	Osoba przypisana	Planowana data naprawy
ID-25 Błąd w nazwie pola „Metoda płatności”	Otwarty	YYZ	Średni	111	05.03.2020	Wojciech Defekt	06.03.2020

Opcjonalnie można dodać datę spodziewanej naprawy błędów, jednak w praktyce wygląda to tak, że data realizacji nie jest stałą, pewną wartością. Dlatego podawana jest zazwyczaj tylko przy defektach krytycznych.

5.6. Raport testów

Poniżej zaprezentowano wzorzec przykładowego raportu. Uwzględniono pola najczęściej występujące w takim dokumencie, jednak warto mieć na uwadze, że w zależności od projektu, metodyki czy wytwarzanego oprogramowania raport może zawierać inny zestaw informacji.

I Cel dokumentu

W tym punkcie należy opisać krótko cel raportu końcowego z testów. Przykładowo:

Niniejszy raport końcowy testów akceptacyjnych powstał w celu poinformowania wszystkich zaangażowanych w projekt o statusie wykonanych prac, a także o jakości wytworzonego oprogramowania.

Dzięki przedstawionym danym osoby decyzyjne mogą wydać rekomendacje do wdrożenia rozwiązania w środowisku produkcyjnym (docelowym).

II Przedmiot i zakres dokumentu

Ten dokument jest raportem końcowym dla testów akceptacyjnych projektu XYZ.

Dokument obejmuje:

- Podsumowanie wykonania scenariuszy testowych (tabela 5.11).

Tabela 5.11. Zestawienie wykonanych scenariuszy testowych

Osoba	Liczba przydzielonych scenariuszy	Liczba wykonanych scenariuszy
Tester 1	234	234
Tester 2	98	98
Tester 3	113	113
Tester 4	24	24
Razem	**469**	**469**

- Podsumowanie zgłoszonych błędów. Oprócz tabelarycznej prezentacji ich liczby — przykład stanowi tabela 5.12 — warto zamieścić także listę wszystkich błędów

zawierającą nr ID błędu wraz z tytułem, dane autora zgłoszenia, informację o statusie oraz wskazanie osoby, do której zgłoszenie jest przypisane.

Tabela 5.12. Zestawienie zgłoszonych błędów

	Liczba błędów o wysokim priorytecie	Liczba błędów o średnim priorytecie	Liczba błędów o niskim priorytecie	Razem
Liczba zgłoszonych błędów	80	115	150	**345**
Liczba błędów ze statusem „Otwarty”	0	7	14	**21**
Liczba błędów ze statusem „Zamknięty”	80	100	132	**312**
Liczba błędów ze statusem „Do wyjaśnienia”	0	8	4	**12**

- Wyniki testów (tabela 5.13).

Tabela 5.13. Zestawienie wyników testów

Scenariusze testowe wykonane z wynikiem POZYTYWNY	Scenariusze testowe wykonane z wynikiem NEGATYWNY	Scenariusze testowe wykonane z wynikiem ZABLOKOWANY	Razem
395	30	44	**469**

- Zakres testów.

W ramach testów dla projektu (NAZWA) zweryfikowano:

(Należy podać listę wszystkich sprawdzonych funkcjonalności lub tytuły wykonanych scenariuszy testowych).

III Terminologia

Zawiera wykaz skrótów i definicje używanych pojęć.

IV Obowiązki, odpowiedzialność i uprawnienia

Zawiera opis obowiązków przypisanych do stanowisk, które biorą udział w tworzeniu raportu końcowego z testów.

V Opis postępowania

1. Dokumenty powiązane z *Raportem końcowym z testów*

Przykładowo: plan testów, dokumentacja projektowa, wymagania biznesowe (tabela 5.14).

Tabela 5.14. Wykaz dokumentów powiązanych z raportem

ID	Dokument (symbol, nazwa)	Data modyfikacji	Wersja dokumentu	Autor dokumentu

2. Podsumowanie wyników testów

Przykładowo: statystyki realizacji, przedstawione opisowo lub za pomocą wykresów, zgłoszone błędy.

Warto wskazać funkcjonalności, które były najbardziej podatne na defekty. Przydatną informacją jest również średni czas realizacji zgłoszenia z podziałem na priorytety.

a) Zespół realizujący testy

Należy wpisać wszystkich uczestników testu z podziałem na role albo odnieść się do listy z planu testów.

b) Czynniki blokujące postęp testów

Należy wskazać czynniki blokujące testy, np. niedostępność środowiska, brak zasobów ludzkich, braki sprzętu o określonych parametrach.

c) Rekomendacja dotycząca wdrożenia

Należy wskazać informacje o testach scenariuszy, które zostały zakończone z wynikiem pozytywnym. Przypadki ze statusem *Zablokowany* nie powodują żadnego ryzyka dla wdrożenia projektu. Wynikiem zakończonych testów UAT (akceptacyjnych) jest pozytywna rekomendacja dotycząca wdrożenia funkcjonalności na środowisko produkcyjne, potwierdzona przez zamawiającego.

3. Szczegółowe wyniki testów

Informacje zawarte w tym punkcie powinny dotyczyć statusów wykonania przypadków testowych oraz statusów defektów. Można je zawrzeć w tabeli, w której podajemy defekty wraz ze statusami. Oprócz tego zamieszczamy:

- listę otwartych defektów i nierozwiązanych problemów;
- ocenę kryteriów zakończenia testów zaczerpniętą z planu testów; może ona mieć formę tabeli (tabela 5.15);

Tabela 5.15. Ocena kryteriów zakończenia testów

Kryterium	Czy spełnione?	Uwagi
Zakończenie ze statusem *Pozytywny* 100% przypadków testowych o priorytecie *Wysoki*	TAK/NIE	
Zakończenie ze statusem *Pozytywny* 65% przypadków testowych o priorytecie *Średni*	TAK/NIE	
Brak otwartych defektów krytycznych (priorytet *Wysoki*)	TAK/NIE	

- ryzyka

  (należy wskazać takie ryzyka, jakie były podane w planie testów — tabela 5.16).

Tabela 5.16. Opis rodzajów ryzyka, które pojawiły się w planie testów

ID	Opis ryzyka	Wykonane działania zapobiegawcze	Bieżący status ryzyka

4. Lista wytworzonych produktów ze wskazaniem linku

- Liczba przypadków testowych w *<tu link do narzędzia>*
- Postęp wykonania przypadków testowych w *<tu link do narzędzia>*
- Raporty dzienne w *<tu link do narzędzia>*
- Rejestr defektów w *<tu link do narzędzia>*
- Inne (wszelkiego rodzaju raporty wygenerowane na potrzeby projektu, np. raport aktywności dziennej osób testujących).

5. Załączniki

Należy załączyć np. plan testów, harmonogram, realizacje scenariuszy, przykładowy raport.

5.7. Zadania

Zadanie 5.1

Wymień elementy, z których składa się plan testów.

Zadanie 5.2

Podaj różnicę pomiędzy scenariuszem testowym a przypadkiem testowym.

Zadanie 5.3

Jak często powinien być aktualizowany plan testów?

Zadanie 5.4

Utwórz scenariusze testowe do dowolnie wybranej przez siebie strony internetowej, następnie przekaż je innej osobie do wykonania. Po wykonaniu zadania sprawdźcie razem wyniki testów. Upewnij się, czy Twoje opisy kroków zostały poprawnie zinterpretowane.

Zadanie 5.5

Wyobraź sobie, że otwierasz sklep internetowy. Stwórz własną listę kontrolną, a następnie przekaż ją do sprawdzenia innej osobie. Zwróć uwagę, ile czynności zostało pominiętych.

Zadanie 5.6

Wykonaj testy wybranej przez siebie strony internetowej, a następnie sporządź raport błędów. Pamiętaj o zachowaniu dobrych praktyk przy tworzeniu zgłoszenia.

6

Metodologie prowadzenia projektu

Obecnie technologie rozwijają się w bardzo dynamicznym tempie. Wiele projektów jest realizowanych z powodzeniem, lecz niemal tyle samo kończy się porażką. Dlaczego tak się dzieje?

Każdy z czytelników potrafiłby zapewne podać kilka powodów, jednak główne przyczyny są dwie: czynnik ludzki i pieniądze. Odpowiednie zarządzanie projektem to odpowiednie zarządzanie zespołem. Pieniądze pomagają w naszych działaniach lub ograniczają je.

Dlatego bardzo ważny jest dobór odpowiedniej metodologii prowadzenia projektu, czyli zbioru zasad i dobrych praktyk, według których zespół projektowy będzie realizował swoje zadania. Zarówno dla testera, jak i programisty wiedza o stosowanym modelu, w którym wytwarzane jest oprogramowanie, jest niezbędna do właściwego planowania swojej pracy.

W podręczniku zawarto ogólne wzory metodologii prowadzenia projektu. Jednak warto wiedzieć, że świat IT rzadko przyjmuje i wdraża je bez dostosowania ich pod kątem konkretnej organizacji. Niemniej najpierw trzeba poznać podstawy, aby opierając się na nich, móc później wyrobić sobie własne zdanie o każdej metodologii.

W niniejszym rozdziale poznasz metodologie wykorzystywane najczęściej w prowadzeniu projektów. Ze względu na obszerność tematu w tym podręczniku zostały zamieszczone najistotniejsze informacje oraz przykłady.

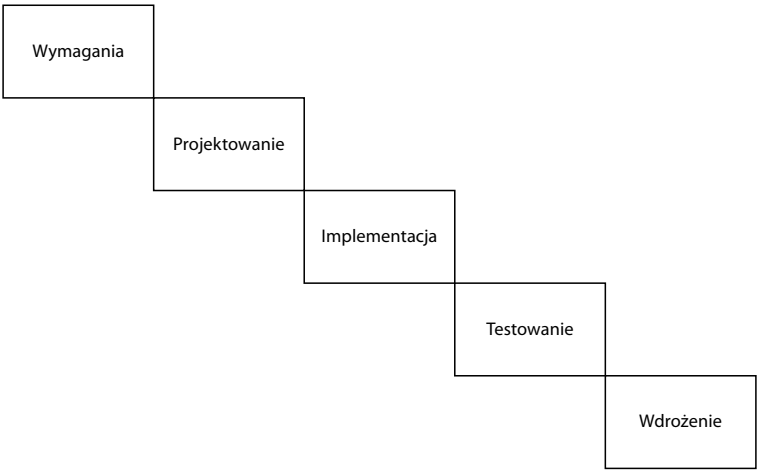
6.1. Model kaskadowy

Model kaskadowy (ang. *waterfall*) jest modelem sekwencyjnym, który składa się z etapów następujących kolejno po sobie. Potocznie mówi się, że to model tradycyjny — czyli przewidywalny, oparty na przejrzystych zasadach. Może być realizowany przez wiele zespołów, które działają niezależnie od siebie, przez co jest narażony na błędy komunikacyjne.

Skąd się wzięła nazwa „kaskadowy"?

Chodzi o sekwencyjność kolejnych faz projektu. Etap pierwszy musi zostać formalnie ukończony, aby można było przejść do realizacji kolejnego. Po zakończeniu jednej fazy schodzimy „w dół" do kolejnej, co doskonale widać na rysunku 6.1. To właśnie przez ten schemat model zyskał nazwę „kaskadowy".

Rysunek 6.1. Schemat modelu kaskadowego



UWAGA

Spójrz na rysunek i sprawdź, na którym etapie odbywa się implementacja i testowanie oprogramowania.

Zastanów się, czy taka kolejność realizacji tych etapów jest zagrożeniem dla powodzenia projektu.

W modelu kaskadowym kolejność etapów jest ważna. Niedopuszczalne jest realizowanie kilku etapów równolegle czy też pomijanie niektórych z nich. Formalne zakończenie każdej fazy to przede wszystkim sprawdzenie i upewnienie się, że wszystkie zaplanowane prace zostały wykonane. Jeżeli w jednym z etapów wystąpiły błędy, których naprawa jest zbyt skomplikowana i pociąga za sobą zmiany w wymaganiach, to zawsze jest możliwy powrót do wcześniejszych faz. Oczywiście takie cofanie się niesie za sobą ryzyko opóźnienia realizacji projektu czy zwiększenia kosztów, ale jest dopuszczalne i w razie konieczności stosowane.

Zamieszczona poniżej tabela 6.1, prezentująca zestawienie etapów z przypisanymi do nich czynnościami, pozwoli lepiej zrozumieć takie podejście.

Tabela 6.1. Etapy modelu kaskadowego z zakresem czynności

Etap	Zakres czynności
Wymagania	Zbieranie wymagań od klienta
	Wykonanie ich analizy
	Koncentracja na potrzebach klienta
Projektowanie	Projektowanie architektury rozwiązania na podstawie przedstawionych wymagań
Implementacja	Tworzenie oprogramowania (kodu) w częściach
	Łączenie części wytworzonego oprogramowania w zintegrowany system
Testowanie	Projektowanie i wykonywanie scenariuszy testowych
	Zgłaszanie błędów
	Tworzenie raportu testów i raportu błędów
Wdrożenie	Wdrożenie rozwiązania na docelowe środowisko

Przykład 6.1

Wyobraź sobie, że w fazie implementacji wykryto wiele niezgodności z istniejącym już systemem głównym. Zgłoszono dużo błędów i uwag do dokumentacji. Okazało się, że uprawnienia, które miały zostać zaimplementowane, wykluczają wiele istniejących już ról w systemie. Wszelkie prace wytwórcze na tym etapie zostały wstrzymane. Projekt wrócił do fazy wymagań, czego efektem było ich zmodyfikowanie. Takie działania, które zmuszają zespoły do wstrzymania prac bądź cofnięcia się do wcześniejszych etapów, wiążą się z ogromnymi kosztami dla klienta. Czy można było uniknąć takiej sytuacji?

Oczywiście. Nie należy bagatelizować tworzenia, zbierania i analizy wymagań. Gdyby dokumentacja była szczegółowa, a wymagania przemyślane i uwzględniające istniejące już rozwiązania systemowe, to taka sytuacja by się nie zdarzyła.

UWAGA

Zalety modelu kaskadowego:

- zdefiniowane etapy procesu;
- łatwość w planowaniu i zarządzaniu;
- wyraźny podział odpowiedzialności.

UWAGA

Cykl życia projektu trwa od kontrolowanego startu (zbieranie wymagań) do kontrolowanego zamknięcia (wdrożenie). Wszystkie etapy są zaplanowane pod kątem zakresu wraz z estymacją pracochłonności.

Szacowanie w modelu tradycyjnym odbywa się przy wykorzystaniu co najmniej jednej z metod przedstawionych poniżej.

- Metoda delficka (inaczej panel ekspertów) bazuje na doświadczeniu osób, które już zetknęły się z podobnym problemem lub mają odpowiednią wiedzę, przydatną do jego rozwiązania. Zastosowanie metody polega w praktyce na przedstawieniu problemu ekspertom.

  Istnieją dwa możliwe sposoby postępowania w obrębie tej metody. Możemy:

  – analizować uzyskane od ekspertów opinie i zadawać im na tej podstawie kolejne, bardziej szczegółowe pytania;

  – pozwolić na krytyczne opinie o ich ekspertyzach osobom o podobnej wiedzy i podobnym doświadczeniu.

Przykład 6.2

Metodę delficką stosujemy w gruncie rzeczy na co dzień, pytając o poradę nauczyciela lub udając się do lekarza w celu uzyskania diagnozy na podstawie opisanych objawów. W obu tych przypadkach nie jest wskazane opieranie się na opinii osób, które nie są ekspertami. Zazwyczaj także — z wyjątkiem bardziej skomplikowanych sytuacji — nie ma potrzeby odwoływania się do zdania większej liczby specjalistów.

UWAGA

- Analiza punktów funkcyjnych (FPA, ang. *Function Point Analysis*) szacuje pracochłonność prac nad różnymi elementami programu, takimi jak:

  – zewnętrzne typy wejścia — to metody dokonywania zmian w wewnętrznych danych systemu;

  – zewnętrzne typy wyjścia — to metody reprezentacji danych przechowywanych przez system (np. wydruki komputerowe);

  – logiczne wewnętrzne typy plików — to pliki używane wewnętrznie przez system;

  – zewnętrzne typy interfejsów plików — to metody wymiany danych między innymi systemami informatycznymi (programy umożliwiające integrację z innymi systemami);

  – zewnętrzne typy zapytań — to metody odczytu danych z systemu niepowodujące ich modyfikacji (np. zapytanie w języku SQL).

→

UWAGA cd.

Celem analizy jest znalezienie wszystkich elementów analizowanego systemu informatycznego, które należą do opisanych powyżej kategorii. Następnie każdemu z nich przypisuje się stopień złożoności; może być ona niska, średnia lub wysoka. Kolejną czynnością jest przypisanie każdemu elementowi liczby punktów odpowiadających jego kategorii i złożoności od 3 do 15, gdzie 3 oznacza niski wpływ na system, a 15 — najwyższy. Wartości przedstawia poniższa tabela 6.2.

Tabela 6.2. Przykładowe zestawienie kategorii dla punktów funkcyjnych

Kategoria	Złożoność		
	Niska	Średnia	Wysoka
Zewnętrzne typy wejścia	3	4	6
Zewnętrzne typy wyjścia	4	5	7
Logiczne wewnętrzne typy plików	7	10	15
Zewnętrzne typy interfejsów plików	5	7	10
Zewnętrzne typy zapytań	3	5	6

Liczba punktów funkcyjnych jest wyznaczana na podstawie sumy wag elementów systemu.

Zazwyczaj model kaskadowy pozwala na wytworzenie oprogramowania zawierającego pełny zestaw funkcjonalności. Realizacja takiego projektu może trwać miesiące, a nawet lata. Dlatego model ten wybierany jest często przy projektach długoterminowych, z jasno określonymi wymaganiami, zależnymi od regulacji prawnych. Nacisk na tworzenie i przechowywanie dokumentacji należy do głównych przyczyn wyboru.

6.1.1. Model prototypowy

Model prototypowy jest przeznaczony dla projektu, w którym klient i zespół chcą przygotować prostą, szybką implementację rozwiązania lub pewnej funkcjonalności. Ma to na celu weryfikację danego rozwiązania w praktyce oraz umożliwia klientowi sprawdzenie, czy zespół poprawnie rozumie jego wizję.

Model ten składa się z następujących etapów:

1. Ogólne określenie wymagań.
2. Budowa prototypu.
3. Weryfikacja prototypu przez klienta/użytkownika końcowego.
4. Wyczerpujące określenie wymagań.
5. Realizacja pełnego systemu zgodnie z modelem kaskadowym.

Głównym celem realizacji prototypu jest szczegółowe określenie wymagań, tj. wykrycie i uniknięcie:

- nieporozumień pomiędzy klientem a twórcami systemu;
- brakujących funkcji;
- trudnych, wręcz niewykonalnych usług;
- braków w specyfikacji wymagań.

WAŻNE

Prototyp nie jest częścią przyszłego pełnego systemu. Jest raczej zapowiedzią tego, co pełny system będzie zawierać. W momencie akceptacji prototypu przez klienta zespół deweloperski przystępuje do prac nad pełnym systemem.

Zalety budowy prototypu to:

- możliwość szybkiej demonstracji pracującej wersji systemu;
- możliwość przeprowadzenia szkoleń dla użytkowników, zanim zbudowany zostanie pełny system.

6.2. Metodyki zwinne

Alternatywnym rozwiązaniem dla tych, którzy nie odnajdują się w uporządkowanym, mocno formalnym modelu kaskadowym, są właśnie metodyki zwinne. Wymagają one jednak dużej samoorganizacji zespołu i koncentracji wielu kompetencji.

Początki metodyk zwinnych wiążą się z ogłoszeniem Manifestu Agile w 2001 roku w Stanach Zjednoczonych. W ośrodku narciarskim zebrała się grupa siedemnastu osób, które reprezentowały różne podejścia i metody budowania systemów informatycznych. Pomimo tak dużej liczby uczestników spotkania okazało się, że są oni zgodni co do podstawowych filarów, które posłużyły jako baza Manifestu.

UWAGA

Oto treść Manifestu Agile.

Wytwarzając oprogramowanie i pomagając innym w tym zakresie, odkrywa się lepsze sposoby wykonywania tej pracy. W wyniku tych doświadczeń przedkłada się:

- **ludzi i interakcje** ponad procesy i narzędzia,
- **działające oprogramowanie** ponad obszerną dokumentację,
- **współpracę z klientem** ponad formalne ustalenia,
- **reagowanie na zmiany** ponad podążanie za planem.

Według Manifestu Agile zazwyczaj docenia się to, co wymieniono po prawej stronie tego zestawienia, jednak bardziej powinno się cenić to, co wymieniono po lewej.

Rzadko wspomina się o dwunastu zasadach Agile, które powstały na skutek publikacji Manifestu. To bardzo ważne wskazówki dla każdego, kto chce wdrożyć metodyki zwinne w swojej pracy. Oto te zasady:

1. Najwyższy priorytet ma dla nas zadowolenie klienta dzięki wczesnemu i ciągłemu wdrażaniu wartościowego oprogramowania.
2. Bądźcie gotowi na zmiany wymagań nawet na późnym etapie jego rozwoju. Procesy zwinne wykorzystują zmiany dla zapewnienia klientowi konkurencyjności.
3. Dostarczajcie funkcjonujące oprogramowanie często, w kilkutygodniowych lub kilkumiesięcznych odstępach. Im częściej, tym lepiej.
4. Zespoły biznesowe i deweloperskie muszą ściśle ze sobą współpracować w codziennej pracy przez cały czas trwania projektu.
5. Twórzcie projekty wokół zmotywowanych ludzi. Zapewnijcie im potrzebne środowisko oraz wsparcie i zaufajcie, że wykonają powierzone zadanie.
6. Najbardziej efektywnym i wydajnym sposobem przekazywania informacji zespołowi deweloperskiemu i wewnątrz niego jest rozmowa twarzą w twarz.
7. Działające oprogramowanie jest podstawową miarą postępu.
8. Procesy zwinne umożliwiają zrównoważony rozwój. Sponsorzy, deweloperzy oraz użytkownicy powinni być w stanie utrzymywać równe tempo pracy.
9. Ciągłe skupienie na technicznej doskonałości i dobrym projektowaniu zwiększa zwinność.
10. Prostota — sztuka minimalizowania ilości koniecznej pracy — jest kluczowa.
11. Najlepsze rozwiązania architektoniczne, wymagania i projekty pochodzą od samoorganizujących się zespołów.
12. W regularnych odstępach czasu zespół analizuje możliwości poprawy swojej wydajności, a następnie dostraja i dostosowuje swoje działania do wyciągniętych wniosków.

UWAGA

Zalety Agile:

- elastyczne podejście do zakresu zmian i ich akceptacji;
- koncentracja na dostarczanej wartości zamiast na zakresie prac;
- stosowanie krótkich iteracji oraz weryfikacji po każdej z nich;
- wzmacnianie samodzielności i odpowiedzialności zespołu.

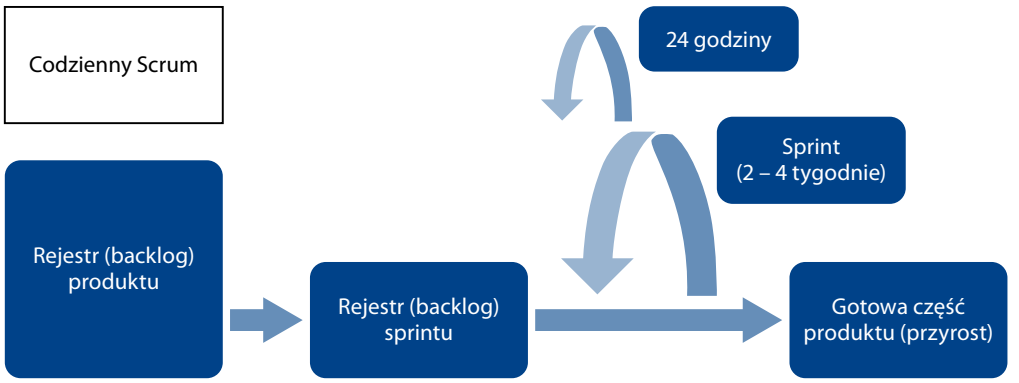
Szacowanie w metodykach zwinnych odbywa się za pomocą następujących technik:

- Poker planistyczny (ang. *Planning Poker*) — zwinna technika oparta na konsensusie, do szacowania używamy kart do gry. Żeby rozpocząć sesję planowania pokera, właściciel produktu lub klient czyta historyjkę użytkownika. Każdy estymator (uczestnik gry) trzyma talię kart pokera planistycznego o wartościach takich jak 0, 1, 2, 3, 5, 8, 13, 20, 40 i 100. Wartości te reprezentują liczbę punktów historii (ang. *Story Points*), w których zespół ocenia historyjkę. Estymatorzy omawiają tę funkcję, w razie potrzeby zadając pytania właścicielowi produktu. Gdy funkcja zostanie w pełni omówiona, każdy z estymatorów prywatnie wybiera jedną kartę, aby przedstawić swoje oszacowanie. Wszystkie karty są następnie okazywane w tym samym czasie. Jeśli wszyscy estymatorzy wybiorą tę samą wartość, staje się to oszacowaniem. Jeśli nie, estymatorzy omawiają swoje szacunki.
- Triangulacja (ang. *Triangulation*) — oszacowanie rozmiaru elementu na podstawie zestawienia go z dwoma innymi elementami.
- *Estimation Board* — przypisywanie elementów do kategorii, w której znajdują się elementy tej samej wielkości.

6.2.1. Scrum

Scrum to metodyka o charakterze iteracyjnym oraz przyrostowym, zaliczana do zwinnych metod pracy zgodnych z Manifestem Agile. Jest to taki sposób prowadzenia projektu, którego celem jest osiągnięcie zamierzonego rezultatu i panowanie nad zmianami. Idealnie wpasowuje się w młode, małe organizacje typu start-up (rozpoczynające swoją działalność).

Na podstawie rysunku 6.2 omówione zostaną teraz narzędzia, praktyki oraz sposób zarządzania projektem realizowanym w Scrumie.



Rysunek 6.2. Zarządzanie projektem w Scrumie

Codzienny (ang. *daily*) Scrum

Jak sama nazwa wskazuje, jest to codzienne spotkanie, trwające maksymalnie do 15 minut, na którym uczestnicy zespołu mówią, co udało im się zrobić wczoraj, co chcą zrobić dzisiaj i jakie widzą przeszkody. Odbywa się ono w trybie *stand-up* (na stojąco).

Sprint

Scrum dzieli projekt na iteracje (wydania cykliczne) o stałej długości (w praktyce iteracja trwa od 2 do 4 tygodni).

UWAGA

Długość iteracji nie ulega zmianie. Jeżeli przyjęto, że sprint trwa 4 tygodnie, to ten zakres czasu jest ostateczny.

Jeżeli zadanie nie może być ukończone w czasie sprintu, odpowiednie cechy produktu są dzielone i zadanie wraca do backlogu produktu. Technika ta nazywa się ograniczaniem czasowym (ang. *timeboxing*).

Przyrost produktu

Wynikiem każdego sprintu jest produkt, który potencjalnie można wdrożyć (nazywany przyrostem).

Rejestr (ang. *backlog*) produktu

Właściciel produktu zarządza spriorytetyzowaną listą planowanych pozycji do zrealizowania (nazywaną backlogiem produktu). Zadania produktu zmieniają się od sprintu do sprintu (nazywamy to udoskonalaniem backlogu produktu).

Rejestr (ang. *backlog*) sprintu

Na początku każdego sprintu zespół scrumowy wybiera z backlogu produktu zbiór pozycji o najwyższym priorytecie (nazwa tego zbioru to *backlog sprintu*). Ponieważ to zespół scrumowy, a nie właściciel produktu wybiera pozycje do realizacji podczas sprintu, wybór nawiązuje raczej do zasady pobierania (według której członek zespołu przydziela sobie zadania sam) niż do zasady wpychania pracy (zadania przydzielane są odgórnie i realizowane według nadanego priorytetu).

Definicja ukończenia (ang. *Definition of Done*)

By się upewnić, że na końcu sprintu uzyskany zostanie produkt nadający się do wdrożenia (sprawny „kawałek" produktu), zespół scrumowy przedyskutowuje i określa odpowiednie kryteria ukończenia sprintu. Dyskusja pogłębia zrozumienie przez zespół poszczególnych pozycji z backlogu i wymagań dotyczących produktu.

Raportowanie

Status sprintu jest raportowany i uaktualniany codziennie w czasie spotkań nazywanych codziennym Scrumem, dzięki czemu zawartość i postęp bieżącego sprintu — a także wyniki testów — są dostępne dla zespołu scrumowego, osób zarządzających i innych zainteresowanych. Na przykład status sprintu może być pokazywany na białej tablicy suchościeralnej.

6.2.1.1. Role w metodyce Scrum

Zgodnie z metodyką Scrum im mniej ról, tym lepiej. Łatwiej wtedy o utrzymanie wysokiego poziomu komunikacji, łatwiej o szybkie podejmowanie decyzji i reagowanie na zmiany.

Scrum wyróżnia trzy role, które wchodzą w skład zespołu scrumowego (ang. *Scrum Team*):

1. Scrum Master (nie ma właściwego tłumaczenia w języku polskim)

   Dba o odpowiednie stosowanie i implementację reguł scrumowych. Jeżeli wystąpią naruszenia reguł, to Scrum Master sprawdza, co jest przyczyną takich działań, i stara się je usuwać z procesu. Eliminuje wszelkiego rodzaju przeszkody, zarówno te związane z zasobami, jak i te, które uniemożliwiają zespołowi stosowanie właściwych reguł i praktyk.

UWAGA

Scrum Master nie jest liderem zespołu, a trenerem (ang. *coach*). Daje tylko wskazówki i doradza zespołowi, aby wybrał właściwą drogę, technologię czy zastosował właściwą metodę. Nie deleguje pracy, nie planuje projektu, a raczej dba o to, aby zespół spełniał wszystkie kryteria, by osiągnąć określone przez siebie cele.

2. Zespół wytwórczy/deweloperski (ang. *Development Team*)

   Zespół wytwórczy może liczyć od 3 do 9 osób (oczywiście jest to tylko sugestia), które razem wytwarzają i testują produkt. Nie ma lidera zespołu, więc wszystkie decyzje podejmowane są wspólnie. Scrum nie dostarcza wytycznych, jak powinno być wykonywane testowanie w tego typu projekcie.

3. Właściciel produktu (ang. *Product Owner*)

   Właściciel produktu reprezentuje użytkownika (przygotowuje historyjki użytkowników) oraz tworzy priorytety w backlogu produktu, zarządza nimi i określa ich wartość. Kontroluje też finansową stronę projektu (ROI). Właściciel produktu nie jest liderem zespołu.

DEFINICJA

Zwrot kosztów z inwestycji (ROI) — podstawowy wskaźnik rentowności, wykorzystywany do mierzenia efektywności danego działania. Wzór na obliczenie ROI jest prosty:

ROI % = (przychód – koszt inwestycji) / koszt inwestycji · 100

Wskaźnik ROI daje odpowiedź na pytanie, czy dany projekt przyniesie firmie wymierną korzyść w określonym czasie. Im wyższy wskaźnik (ROI), tym lepiej.

6.2.1.2. Historyjki użytkownika

Słabej jakości specyfikacja — główny powód niepowodzenia projektu — może powstawać w wyniku braku zrozumienia przez klienta własnych potrzeb, braku globalnej wizji systemu, nadmiernych lub sprzecznych właściwości oraz innych błędów w przepływie informacji.

W środowisku zwinnym historyjki użytkownika pisane są po to, by uchwycić wymagania z perspektywy programistów, testerów i przedstawicieli biznesu. W przeciwieństwie do modelu sekwencyjnego, gdzie wspólna wizja właściwości jest osiągana przez przeglądy formalne po utworzeniu wymagań, w zwinnym rozwoju oprogramowania wizja ta jest osiągana przez częste przeglądy nieformalne podczas pisania wymagań.

DEFINICJA

Dobrze stworzona **historyjka użytkownika** spełnia warunki modelu **INVEST**, to znaczy, że jest:

- **Independent** — niezależna;
- **Negotiable** — negocjowalna;
- **Valuable** — wartościowa;
- **Estimable** — dająca się oszacować (można określić niezbędny czas i zasoby);
- **Sized Appropriately** — odpowiedniej wielkości;
- **Testable** — testowalna.

Historyjki użytkownika (ang. *User Stories*) mają na celu skupienie się na realnym użytkowniku systemu i na zaspokojeniu jego potrzeb. Zmuszają autora do zastanowienia się nad tym, jakie rozwiązanie będzie najlepsze dla użytkownika.

Każda historyjka powinna zawierać kryteria akceptacji dla właściwości wymienionych w schemacie INVEST. Te kryteria powinny być definiowane we współpracy pomiędzy przedstawicielami biznesu, programistami i testerami. Dają one programistom i testerom lepszy wgląd w wizję właściwości, którą reprezentanci biznesu będą sprawdzać. Zespół zwinny uważa zadanie za ukończone, gdy zbiór kryteriów akceptacyjnych jest spełniony.

UWAGA

Koncepcja 3C (autorstwa Rona Jeffriesa) mówi, z jakich części powinna składać się historyjka. Są to:

- Karta (ang. *Card*) lub fiszka, na której zapisano historyjkę. Stanowią one fizyczny odnośnik do wymagania.
- Dyskusja (ang. *Conversation*) na temat szczegółów.
- Potwierdzenie (ang. *Confirmation*) dzięki testom akceptacyjnym, które potwierdzają implementację z punktu widzenia biznesu.

Zalety korzystania z historyjek:

- mniej dokumentacji — więcej spotkań i rozmów;
- najważniejszy jest cel użytkownika, a nie sam system;
- nie narzucają ustalania szczegółów z góry.

Historyjki użytkownika muszą obejmować zarówno właściwości funkcjonalne, jak i niefunkcjonalne (przykład 6.3).

Przykład 6.3

Właściciel produktu zdefiniował wymagania dotyczące aplikacji sklepu internetowego. Poniżej przykład jednej z historyjek użytkownika.

Użytkownik z rolą „Sprzedawca" chce, żeby po wejściu w zakładkę „Zamówienia" stan zamówień o statusie „Nieopłacone" wyświetlał się na czerwonym tle (właściwość niefunkcjonalna, związana z wyglądem pewnego elementu programu, a nie z funkcją, jaką pełni).

Użytkownik niezarejestrowany chce mieć możliwość złożenia zamówienia w sklepie internetowym (właściwość funkcjonalna).

UWAGA

Zalety metodyki Scrum:

- komunikacja ponad dokumentacją;
- przejrzystość działań (krótkie sprinty z ustalonym zakresem prac);
- szybkie reagowanie na zmiany wymagań.

6.2.2. Kanban

DEFINICJA

Kanban jest metodyką wywodzącą się z Toyota Production System (metody zarządzania produkcją stosowanej przez wiele firm w Japonii oraz na całym świecie). Nazwa „kanban" w języku japońskim oznacza tablicę informacyjną. Jej celem jest wizualizacja i optymalizacja przepływu pracy w łańcuchu wartości dodanej (ang. *a value-added chain*).

Kanban składa się z pięciu zasad:

- Zobrazuj przepływ pracy.
- Ogranicz WIP (ang. *Work in Progress*).
- Zarządzaj przepływem.
- Ustal wyraźne zasady.
- Wspólnie ulepszaj.

Prezentowane podejście nie przewiduje iteracji (powtarzania), a jedynie ciągły proces.

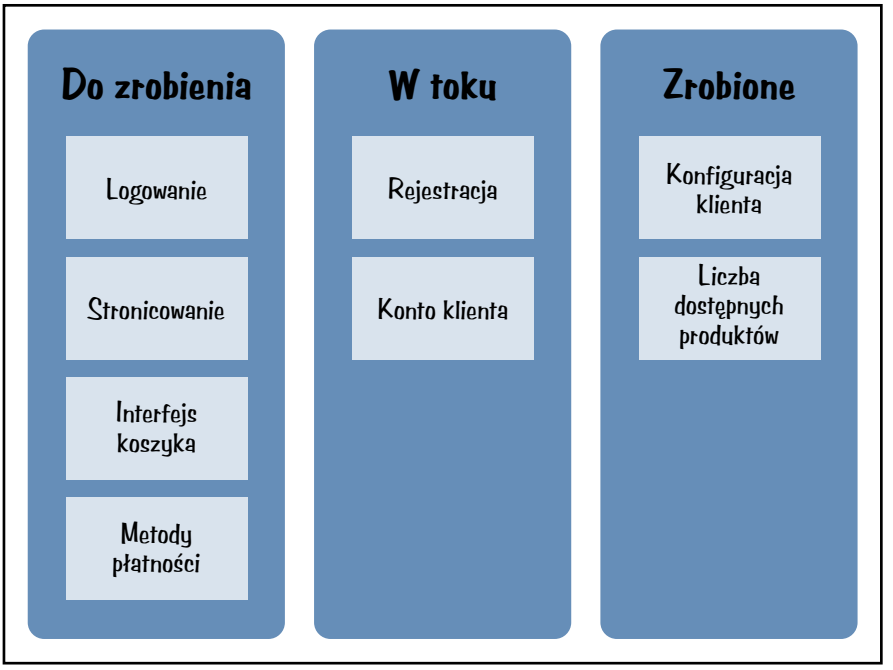
Kanban wykorzystuje trzy narzędzia, którymi są:

1. **Tablica kanbanowa** (rysunek 6.3). Łańcuch wartości, którym zarządzamy, jest wizualizowany na tablicy kanbanowej. Każda kolumna pokazuje etap procesu — zbiór powiązanych czynności, które powinny zostać zrealizowane, lub zadania, które będą przetwarzane, są symbolizowane przez kolorowe kartki. Kartki są przesuwane od lewej do prawej strony przez kolejne kolumny tablicy, odpowiadające następującym po sobie etapom procesu.

Przykład 6.4

Załóżmy, że pracujemy nad aplikacją sklepu internetowego. Tablica kanbanowa projektu mogłaby wyglądać tak jak ta pokazana na rysunku 6.3. Jak widać, zespół pracuje obecnie nad modułami odpowiedzialnymi za rejestrację oraz konto klienta. Zadania już wykonane to opracowanie konfiguracji konta klienta i części aplikacji przechowującej, aktualizującej i udostępniającej informacje o liczbie dostępnych produktów. Wciąż pozostało całkiem sporo do zrobienia.

2. **Limit pracy w toku** (ang. *Work in Progress*). Liczba równolegle wykonywanych zadań jest ściśle ograniczona. Jest to kontrolowane przez ustalenie maksymalnej dozwolonej liczby kartek na danym etapie lub globalnie na tablicy. Gdy na danym etapie na tablicy pojawia się wolne miejsce, pracownik przesuwa kartkę z wcześniejszego etapu.
3. **Czas realizacji** (ang. *Lead Time*). Metodyki Kanban używa się do optymalizacji ciągłego przepływu zadań przez minimalizację (średniego) czasu realizacji dla całego łańcucha wartości. Kanban wykazuje pewne podobieństwo do Scruma. W obu podejściach wizualizacja aktywnych zadań (np. na dostępnej dla wszystkich tablicy suchościeralnej) zapewnia przejrzystość zawartości i postępu zadań. Zadania



Rysunek 6.3. Przykładowa tablica kanbanowa

jeszcze nieprzydzielone czekają w zbiorze zadań do wykonania; w momencie gdy zwalnia się miejsce na kolejnym etapie, są one odpowiednio przesuwane na tablicy kanbanowej.

Zarówno iteracje, jak i sprinty są opcjonalne w Kanbanie.

Proces w Kanbanie pozwala na wydawanie produktów raczej jeden po drugim niż jako część wydania. Ograniczenia czasowe (ang. *timeboxing*), jako mechanizm synchronizujący, stają się tym samym opcjonalne, inaczej niż w Scrumie, gdzie synchronizuje się wszystkie zadania w ramach sprintu. Zespoły stosujące metodykę Kanban mierzą czas realizacji (średni czas od momentu zgłoszenia żądania pracy do momentu jej zakończenia) i optymalizują swoje procesy, aby skrócić czas wykonywania zadań.

UWAGA

Zalety metodyki Kanban:

- wizualizacja przebiegu prac;
- łatwa weryfikacja optymalizacji procesów.

6.2.3. Scrumban

Scrumban jest hybrydą uzyskaną z połączenia struktury Scrum z metodami Kanbana. Jest rekomendowaną metodyką dla zespołów, które chcą przejść od formy pracy właściwej dla Scruma do Kanbana. Obecnie jest to najpopularniejsza metodyka zwinna. Więcej informacji o niej znajdziesz w rozdziale 7.

UWAGA

Jakie elementy Scruma można tu odnaleźć?

- Planowanie — iteracje w regularnych odstępach czasu, synchronizowane z przeglądami i retrospektywami.
- Przejrzystość — proces musi być zrozumiały dla wszystkich zaangażowanych.
- Inspekcja — scrumowe artefakty muszą być często przeglądane w celu wykrycia niepożądanych rozbieżności.
- Adaptacja — w momencie zauważenia rozbieżności proces musi zostać natychmiast skorygowany.

Elementy zaczerpnięte z Kanbana wpływają na ulepszenie procesu, dodają do metody Scrumban wizualizację i więcej wskaźników wartości.

UWAGA

Jakie elementy Kanbana można tu odnaleźć?

- Ciągły system i przepływ pracy.
- Nowe wymagania mogą być dodane zawsze, o ile w kolejce zadań (WIP) istnieje wolne miejsce.
- Wyraźne limity liczby elementów w toku w dowolnym momencie.
- Poszczególne role nie są jasno określone.
- Krótkie terminy realizacji — nacisk na analizę i planowanie.
- Korzystanie z buforów procesów i diagramów przepływu, aby ujawnić słabości procesu i zidentyfikować możliwości poprawy.

6.3. Zestawienie metodyk pod kątem różnic

W tabeli 6.3 zestawiono różnice pomiędzy przedstawionymi metodykami. Wybrano najistotniejsze czynniki wpływające na definiowanie metodyk. Pamiętaj jednak, że w praktyce stosowanie poszczególnych metod w firmach wygląda różnie i może odbiegać od przedstawionego tutaj wzorca.

Tabela 6.3. Zestawienie porównawcze metodyk

	Waterfall	Scrum	Kanban	Scrumban
Metodyka zwinna		✓	✓	✓
Szczegółowa dokumentacja wymagań	✓			
Szybko zmieniające się środowiska testowe		✓		✓
Role kierownicze	✓			
Przypadki testowe	✓	✓		
Małe zespoły		✓		✓
Wdrażanie kilku funkcjonalności w jednym wydaniu	✓			
Dopuszczalna zmiana zakresu zadań	✓		✓	✓
Dopuszczalna zmiana terminu wdrożenia	✓		✓	
Sprinty		✓		✓
Ograniczenia co do realizowanej liczby zadań		✓	✓	✓
Wizualizacja zadań		✓	✓	✓

6.4. Zadania

Zadanie 6.1

Jakie są wady testowania w modelu kaskadowym?

Zadanie 6.2

W jakich projektach sprawdzi się dobrze model kaskadowy?

Zadanie 6.3

Jakie role występują w zespole scrumowym?

Zadanie 6.4

Kto jest liderem zespołu scrumowego?

Zadanie 6.5

Wymień części, z jakich powinna się składać historyjka użytkownika (według koncepcji 3C).

Zadanie 6.6

Wymień wartości, o których mówi Manifest Agile.

Zadanie 6.7

Przygotuj własną listę Kanban w narzędziu Trello. Zaplanuj w niej wszystkie zadania związane ze szkołą na najbliższe 30 dni.

Zadanie 6.8

Jakie elementy Kanbana znajdziesz w metodyce Scrumban?

7

Od pomysłu po wdrożenie — praktyczne zastosowanie zdobytej wiedzy

Realizacja projektu od początku do końca to praca długa i wymagająca zaangażowania. Niezależnie od czasu trwania projektu liczba przypisanych do niego pracowników może przekraczać nawet sto osób! Dlatego ważne jest, aby każdy członek zespołu projektowego znał swoją rolę, swój zakres obowiązków i odpowiedzialności. Duży nacisk kładzie się na komunikację i przestrzeganie dobrych praktyk. Ze względu na dynamiczne tempo pracy zakłada się, że członkowie zespołu znają podstawowe zasady realizacji zadań w ramach swoich kompetencji. Wprowadzenie nowych członków do zespołu polega na krótkim zapoznaniu ich ze współpracownikami i pokazaniu, na którym etapie projektu jesteśmy. Bardzo ważne jest jak najszybsze wdrożenie nowych pracowników. W związku z tym nie należy lekceważyć wcześniejszego zapoznania się z teorią — jest ona niezbędna do płynnego przejścia do praktyki.

Celem niniejszego rozdziału jest wyjaśnienie ról poszczególnych osób i symulacja przejścia procesu tworzenia oprogramowania, tak by jak najlepiej oddawała rzeczywisty przypadek.

7.1. Etap pierwszy — pomysł i zapytanie ofertowe

Pani Ania, pracująca w prywatnym przedszkolu „Skrzat", badała za pomocą ankiet, jakich zajęć brakuje w placówce (ankietę wypełniali rodzice). Ankietowani wskazali na zajęcia językowe, pojawiły się też sugestie, że dzieci powinny zdobywać taką umiejętność jak orientacja w terenie. Pani Ania pomyślała, że świetnym rozwiązaniem byłoby wprowadzenie lekcji językowych w formie zajęć interaktywnych.

Wychowawczyni zgłosiła dyrekcji przedszkola zapotrzebowanie na zakup usługi w firmie informatycznej. Dyrekcja poprosiła o sprecyzowanie, co dokładnie takie oprogramowanie miałoby robić i na jakich urządzeniach działać, po czym zadeklarowała, że na podstawie uzyskanych informacji wyśle zapytania ofertowe do firm. Dodatkowo pomysł został przedstawiony na zebraniu rady rodziców. Oczywiście pojawiły się głosy, że budżet można przeznaczyć na inne aktywności, np. na dodatkowe zajęcia z integracji sensorycznej, lekcje tańca czy wyjścia do kina — jednak dyrekcja mocno broniła pomysłu.

Jak myślisz, dlaczego?

CIEKAWOSTKA

Otóż raz wydane pieniądze w przypadku programu komputerowego dają placówce narzędzie, z którego można będzie korzystać przez najbliższe lata. Co prawda oprogramowanie będzie wymagało aktualizacji czy dodania nowych funkcji, jednak koszty konserwacji i rozwoju nie będą już tak duże jak te, które pociąga za sobą zbudowanie aplikacji.

Kilkoro rodziców twierdziło, że jest to marnotrawienie budżetu przedszkola, ale przeważyły głosy za tym, by zanim pomysł zostanie odrzucony, zebrać propozycje i kosztorysy od firm, które deklarowały, że podejmą się zadania.

Po ostatecznej akceptacji przez dyrekcję przedszkola pani Ania przygotowała prezentację opisującą jej pomysł oraz wymagania dotyczące aplikacji. Ze względu na brak wykształcenia programistycznego nie zawarła w jej treści szczegółów dotyczących technologii bądź budżetu. Prezentacja została wysłana do dyrektora przedszkola. Przełożony zaś przekazał uzyskane od wychowawczyni informacje firmom, które mogłyby zrealizować projekt.

Oto treść e-maila wysłanego do trzech firm poleconych przez rodziców:

„Kontaktuję się z Państwem w imieniu Przedszkola Niepublicznego »Skrzat«. W związku z planowaną modyfikacją zajęć dodatkowych chcielibyśmy dostać od Państwa ofertę na opracowanie pomysłu oraz zbudowanie przeznaczonej dla dzieci aplikacji do nauki (poprzez zabawę) języka obcego. Program musi być prosty i intuicyjny oraz zrealizowany tak, by dziecko nie musiało prosić opiekuna o pomoc. Aplikacja ma wykorzystywać słownictwo w językach angielskim i polskim. Zaproponowany sposób nauki i zabawy musi przybrać postać pojedynczych lekcji, które będą wraz z dziećmi realizowane

w trakcie zajęć przedszkolnych. Celem edukacyjnym — oprócz nauki języka — ma być również orientacja w terenie. Proszę mieć na uwadze, że dzieci nie potrafią jeszcze dobrze czytać, dlatego ważne jest, aby słówka były nie tylko wyświetlane na ekranie, ale także odczytywane przyjaznym, spokojnym głosem".

UWAGA

Dlaczego to nie pani Ania wysłała e-mail do firm, choć pomysł był jej i sama wiedziała najlepiej, co chce uzyskać, miała też zgodę przełożonego na realizację projektu? E-maile od przełożonego wyglądają bardziej profesjonalnie, a firmy chętniej angażują się w przygotowanie oferty w odpowiedzi na zapytanie ofertowe skierowane przez pracownika wysokiej rangi.

Na tym etapie projektu podstawowym zadaniem jest ustalenie szczegółów takich jak budżet, technologie czy rozdysponowanie zadań w zespole.

Spójrzmy teraz na problem z perspektywy pracowników firmy, która otrzymała e-mail od dyrektora przedszkola.

Jak łatwo się domyślić, uznano, że opis z e-maila nie jest wystarczający, aby przygotować nawet wstępną wycenę. Zespół przygotował więc listę pytań:

1. Dla jakich urządzeń przeznaczona jest aplikacja?
2. Ilu maksymalnie użytkowników będzie z niej korzystać w tym samym czasie?
3. Ile ma trwać jedna lekcja?
4. Jaki jest poziom trudności?
5. Czy aplikacja ma działać w trybie offline?
6. Czy użytkownicy będą się logować do platformy?

UWAGA

Często klienci nie są świadomi, jak zrealizować to, czego potrzebują, dlatego w odpowiedzi na zapytanie ofertowe poza pytaniami o szczegóły warto zamieścić kilka sposobów rozwiązania problemu — najlepiej popartych przykładami wcześniej zrealizowanych projektów.

UWAGA

Ważne jest, byśmy pamiętali, że przy tworzeniu aplikacji z logowaniem powinniśmy rozważyć następujące podstawowe kwestie dotyczące zabezpieczeń:

1. Czy dostęp do stron można uzyskać tylko po zalogowaniu? Jeśli tak, to czy do wszystkich, czy tylko niektórych?

→

UWAGA cd.

2. Czy planowany system zabezpieczeń jest na tyle dobry, że jego złamanie nie jest proste? [Warto w tym miejscu dodać, że praktycznie nie ma systemów, które nie mogłyby być złamane. Trzeba stosować takie zabezpieczenia, by nakład sił i środków do ich złamania był zbyt duży, żeby się to opłacało potencjalnym hakerom. Lista najpopularniejszych ataków jest co roku uaktualniana przez organizację OWASP (Open Web Application Security Project)].
3. Czy login i hasło w bazie danych są zapisane w postaci zaszyfrowanej? Jeśli tak, to jakich algorytmów szyfrujących użyto?
4. Czy dostęp do bazy danych mają wszyscy zalogowani użytkownicy, czy tylko ci, dla których ustawiono określone role?

Po otrzymaniu odpowiedzi od klienta uznano, że nie potrzeba żadnych elementów uwierzytelniających, bo ze względu na wiek użytkowników i na to, że nie wszyscy potrafią jeszcze pisać i czytać, system logowania mógłby wręcz przeszkadzać w korzystaniu z aplikacji.

Kiedy członkowie zespołu znają już bardziej szczegółowo wymagania klienta, uruchamiane są liczne procesy w działach, które pozyskują projekty dla firmy.

UWAGA

Na tym etapie, aby móc przedstawić klientowi wstępny, ale dobrze poparty zakresem prac kosztorys, często rozpisuje się zadania na poszczególne role w docelowym zespole.

Zadania dla poszczególnych osób uczestniczących w projekcie wyznaczone zostaną dopiero po podpisaniu umowy lub zakończeniu przetargu.

Nawet w przypadku aplikacji darmowych kalkulowane są koszty utrzymania serwera. Potrzeba dużo czasu na przeprowadzenie konsultacji z innymi zainteresowanymi (najlepiej z grupy docelowej — użytkowników).

UWAGA

Jeśli wybierzesz karierę programisty, często będziesz się spotykać ze sformułowaniem: „Aplikacje mają działać w architekturze klient-serwer”. Warto wyjaśnić, co ono znaczy.

DEFINICJA

Klient-serwer (ang. *client/server*) to architektura systemu komputerowego, w szczególności oprogramowania, umożliwiająca podział zadań. Serwer zapewnia usługi dla klientów (zazwyczaj aplikacji zainstalowanych na komputerach użytkowników; mogą być nimi nawet zwykłe przeglądarki internetowe) zgłaszających do serwera żądania obsługi (ang. *service request*). Główną zaletą architektury klient-serwer jest to, że przez sieć są udostępniane jedynie konkretne odpowiedzi na konkretne pytania, a nie całe pliki czy programy do wykonania lokalnego, jak to się odbywało w dawnych architekturach serwerowych.

7.2. Etap drugi — oferta

Po skompletowaniu dokumentów i odpowiedzi na niezbędne pytania została przygotowana oferta projektowa. Oferta na realizację projektu na ogół składa się z wyceny i dokumentów analitycznych, w których zawarty jest dokładny opis, jak ma być zrealizowana aplikacja pod względem technicznym i merytorycznym. Niezbędna jest też wizualizacja.

Przykład 7.1

W tabeli 7.1 przedstawiono przykładową wycenę prac. Wspomniana w rozdziale 4. liczba MD, określająca pracochłonność, jest na każdym etapie mnożona przez liczbę pracowników realizujących zadanie; na tej podstawie wyznacza się kwotę do zapłaty. Robi się to w ten sposób, że całkowitą liczbę MD (widać ją w ostatnim wierszu tabeli) mnoży się przez wynegocjowaną stawkę za dzień pracy. W negocjacjach biznesowych na wysokim poziomie nie rozdziela się stawki programisty od stawki analityka czy testera. Zatem końcowa kwota projektu wynosiłaby 307 · wynegocjowana stawka.

Tabela 7.1. Przykładowa wycena prac

Planowane prace	Liczba MD	Liczba testerów/ programistów	Suma MD z uwzględnieniem liczby testerów/programistów
Weryfikacja scenariuszy	5	3	15
Testy akceptacyjne	15	6	90
Testy eksploracyjne	3	6	18
Testy regresji	7	6	42
Zapoznanie się z dokumentacją/aplikacjami	5	6	30
Uzupełnienie scenariuszy (jeżeli będzie taka konieczność)	3	3	9

Planowane prace	Liczba MD	Liczba testerów/ programistów	Suma MD z uwzględnieniem liczby testerów/programistów
Tworzenie dokumentacji (raporty, podsumowania)	3	1	3
Testy wydajnościowe	2	2	4
Testy bezpieczeństwa	3	2	6
Prace programistyczne	30	3	90
Liczba MD	76		307

UWAGA

Decydującymi czynnikami, które wpłyną na wybór oferty, są:

- budżet,
- skład zespołu projektowego (czy są dostępne osoby z odpowiednim doświadczeniem),
- wstępny projekt wizualny w postaci makiety.

Często na tym etapie zapadają decyzje o wstrzymaniu projektu — najczęstszym powodem jest zbyt mały budżet lub zbyt krótki termin realizacji zlecenia (klient oczekuje szybkiego zakończenia prac, nie biorąc pod uwagę czasu poświęconego na analizę rozwiązania).

Na podstawie złożonych ofert władze przedszkola podjęły decyzję o wyborze firmy wykonawczej.

Firma, która wygrała konkurs, jest przedsiębiorstwem z kilkuletnim doświadczeniem w realizacji projektów informatycznych. W jej portfolio znajduje się m.in. aplikacja do gry miejskiej oparta na geocachingu (uczestnik takiej gry na podstawie lokalizacji GPS dowiaduje się np., że w danym miejscu jest ukryty przedmiot).

Wybrany wykonawca zaproponował stworzenie prostej gry polegającej na wyjściu z labiryntu dzięki sterowaniu postacią za pomocą słów po angielsku.

Prace nad większością projektów firmy wykonywano, opierając się na Agile, ze względu na większe zadowolenie klienta, który na spotkaniach podsumowujących mógł zobaczyć efekty prac. W taki sposób będzie realizowana także aplikacja dla przedszkola.

Na czele zespołu technicznego stoi *tech team leader* (osoba o bardzo wysokich kompetencjach technicznych oraz miękkich, potrafiąca kierować grupą. Nie należy mylić tej roli z rolą architekta, który ma za zadanie odpowiednio zaprojektować aplikację pod kątem rozwiązań technicznych) z wieloletnim stażem. W skład zespołu wchodzi dwóch juniorów (programistów z niewielkim stażem) i jeden mid (mający już pewne

doświadczenie). Wspiera ich analityk z wieloletnim doświadczeniem. Do dyspozycji mają również zespół testerów z koordynatorem testów.

Siedziba firmy znajduje się w innym mieście niż przedszkole.

Zdecydowano, że projekt będzie miał postać labiryntu, z którego będzie można wyjść za pomocą komend tekstowych w języku angielskim.

Na spotkaniu z dostawcą usług zostały ustalone następujące kwestie, które później opisano w dokumentacji technicznej i merytorycznej:

1. Celem współpracy jest gotowy produkt — aplikacja webowa o nazwie Labirynt, w której przedszkolaki będą wpisywać słowa w języku angielskim, wybierając kierunek poruszania się.
2. Architekt zarekomendował, aby napisać aplikację webową w języku JavaScript, z wykorzystaniem kodu HTML. Dzięki temu aplikacja będzie mogła być uruchomiona na różnych platformach.
3. Ustalono skład stałego zespołu projektowego (tabela 7.2). Oczywiście na poszczególnych etapach projektu będą się pojawiać role spoza poniższego zestawienia. Odgrywający te role będą wykonywać swoje prace w węższym zakresie czasowym niż ci, którzy wchodzą w skład stałego zespołu.

Tabela 7.2. Zestawienie ról i czynności

Rola	Zakres czynności
Project Manager	Podział zadań pomiędzy członków zespołu, organizacja czasu pracy swojego i podwładnych, zakup produktów i usług niezbędnych do realizacji projektu (np. oprogramowania, licencji, sprzętu), komunikowanie się z klientami i członkami zespołu, kontrola jakości wykonywanych zadań projektowych, zarządzanie budżetem projektu, ocena ryzyka związanego z realizacją projektu.
Product Owner	Przygotowanie wizji produktu, przeprowadzanie badań rynkowych w grupie wiekowej zbliżonej do grupy docelowej, zarządzanie backlogiem produktu, porządkowanie elementów backlogu.
Tester (dwie osoby)	Przygotowanie danych testowych niezbędnych do przeprowadzenia testów oprogramowania, przeprowadzanie testów na podstawie przypadków testowych oraz przeprowadzanie testów eksploracyjnych, pielęgnacja repozytorium testów regresji, zgłaszanie błędów i retestowanie przygotowanych poprawek błędów.
Programista (dwie osoby)	Przygotowanie rozwiązania zgodnego z wymaganiami, implementacja testów jednostkowych, debugowanie błędów i ich poprawienie.
Analityk	Kompletowanie i analiza wymagań, tworzenie dokumentacji związanej z analizą, przegląd i akceptacja przypadków testowych i planu testów, przegląd i analiza zgłoszeń z testów dotyczących kwestii nieobjętych analizą, a mających istotny wpływ na wytwarzane oprogramowanie. Zaprojektowanie systemu zgodnie z wymaganiami.

1. W wyniku ustaleń wspólnie podjęto decyzję o rezygnacji z roli koordynatora testów ze względu na niską szczegółowość aplikacji, jak również ograniczony budżet przedszkola.
2. Wybór narzędzi — wybrano narzędzia darmowe, tj. Trello (do rozpisania zadań dla programistów i testerów) oraz TestLink (do tworzenia przypadków testowych). Dodatkowo Jira pełni funkcję narzędzia do zarządzania projektem — korzysta z niej Project Manager i na podstawie danych tworzy treść wiadomości kierowanych do zespołu i do klienta. Jeśli chodzi o narzędzie do tworzenia oprogramowania, to firma posiada niezbędne licencje, w związku z czym ich koszt nie zalicza się do końcowej ceny za przygotowanie programu.
3. Wybór metodyki — zdecydowano się na Scrumban. Zrezygnowano z codziennych spotkań (ang. *daily*).

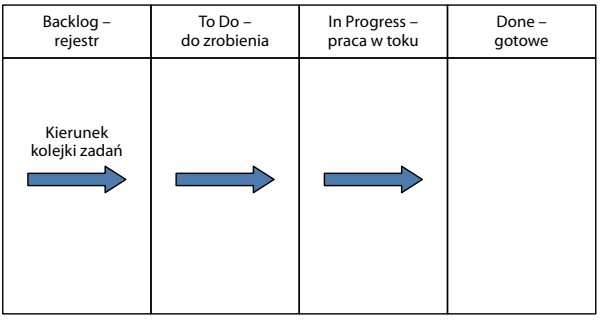
UWAGA

Liczba ról powołanych w projekcie jest zależna nie tylko od wybranej metodyki, ale też od dostępnych zasobów ludzkich, jak również od budżetu klienta (zamawiającego usługę).

7.2.1. Przygotowanie tablicy Scrumban

Prace przygotowawcze nie powinny sprawiać problemu nawet osobom zaczynającym przygodę z tą metodyką. Tworząc tablicę (rysunek 7.1), należy przestrzegać kilku zasad:

- Tabela musi zawierać podstawowe kolumny: *Backlog* — rejestr, *To Do* — do zrobienia, *In Progress* — praca w toku, *Done* — gotowe.
- Należy ustawić limit zadań; po umieszczeniu listy zadań w kolumnie *Backlog* ustawiamy limit postępu prac dla tej kolumny. Zespół przesuwa zadania z kolumny *Backlog* w prawą stronę tabeli, aż stanie się ona pusta, co jest dla zespołu informacją, że trzeba zaplanować kolejne zadania. Dobrą praktyką jest dbanie o to, by liczba zadań w kolumnie *Backlog* była dwa razy większa niż liczba programistów w zespole.
- Wyszukanie i zdefiniowanie tzw. wąskich gardeł. Warto podzielić kolumnę *In Progress* na mniejsze kolumny, w zależności od specyfiki projektu, aby wdrożyć oddzielne limity WIP. Takie podejście pozwala zidentyfikować wąskie gardła — punkty, które spowalniają przepływ procesu i mogą go nawet całkowicie wstrzymać.



Rysunek 7.1.
Przykładowy szablon tablicy Scrumban

- Pomiar i kontrola. Dwa najważniejsze wskaźniki to średni czas oczekiwania (czyli czas od momentu zgłoszenia wymagania przez klienta do momentu uzyskania w pełni działającej funkcjonalności w środowisku produkcyjnym) oraz średni czas wytworzenia (czyli czas, jaki zespół poświęcił na pracę nad danym wymaganiem).

7.3. Harmonogram prac

Prace zaplanowano w trzech sprintach, każdy z nich ma trwać dwa tygodnie. W zestawieniu (tabela 7.3) widnieje ponadto sprint 0. (jeden tydzień) — jest to czas na robocze spotkania, przygotowanie sprzętu, przekazanie wszelkich informacji o projekcie, zapoznanie się członków zespołu itp.

Tabela 7.3. Przykładowy podział sprintów

Sprint 0. (wprowadzenie do prac) **Termin: 4–8.05.2020**	**Sprint 1.** **Termin: 11–22.05.2020**	**Sprint 2.** **Termin: 25.05–5.06.2020**	**Sprint 3.** **Termin: 8–22.06.2020**
Cel sprintu: zapoznanie się ze stosowanymi rozwiązaniami i aplikacjami.	Cel sprintu: uzyskanie widoku z obiektem oraz sterowanie nim za pomocą wpisanego słowa.	Cel sprintu: dodanie ścian w labiryncie i obsługa ściany, początku i końca labiryntu.	Cel: oddanie całej aplikacji spełniającej wszystkie warunki określone w wymaganiach.
UX designer: realizacja makiet zgodnie z ustaleniami z klientem. Programista, tester, analityk: wdrożenie w projekt — konfiguracja środowiska. Architekt: decyduje, z jakich technologii będzie korzystał zespół, uwzględniając jego umiejętności. Projektuje cały system wysokopoziomowo.	Programista: realizacja możliwych kroków użytkownika. Architekt: rozwiązanie problemu przechodzenia między ścianami. Tester: rozpisanie przypadków testowych do realizowanej funkcji.	Programista: realizacja kroków w labiryncie. Tester: testy wybierania kroków. Klient: testowanie bieżącej wersji aplikacji. Drugi programista: przegląd (ang. *review*) kodu.	Programista: realizacja wymagań oraz dodanie listy komend widocznych dla użytkownika, refaktoryzacja kodu (czyli zwiększenie jego czytelności oraz optymalizacja) na podstawie uwag po przeglądzie kodu. Tester: testy nowych wymagań + testy regresyjne. Klient: realizacja testów po stronie klienta. Tester bezpieczeństwa: testy zabezpieczeń aplikacji. Tester wydajnościowy: testy wydajności aplikacji.

7.4. Realizacja prac projektowych

W niniejszym podrozdziale skupimy się na rolach osób zaangażowanych w projekt w poszczególnych fazach jego realizacji oraz na wykonywanych czynnościach, a także na rejestrowaniu zadań w narzędziach.

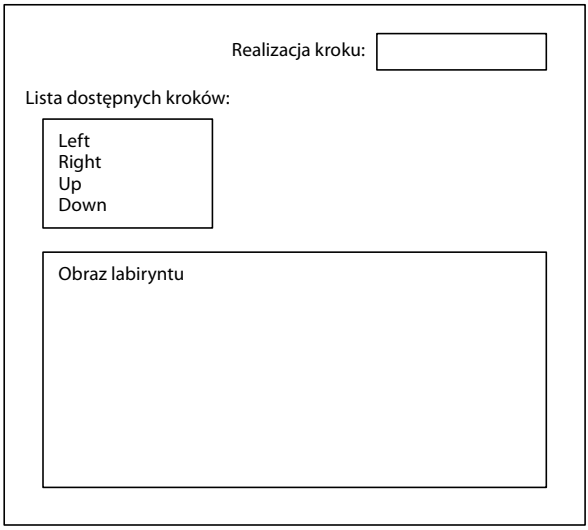
7.4.1. Sprint 0.

W ramach prac nad tzw. sprintem zerowym niezbędni będą specjaliści spoza stałego zespołu projektowego. Po zapoznaniu się z koncepcją rozwiązania UX designer przystępuje do realizacji makiet. Makiety mają za zadanie przedstawić w graficznej postaci proponowane rozwiązanie wraz z kompletną kolorystyką i nazewnictwem.

CIEKAWOSTKA

UX designer to osoba odpowiedzialna za zbadanie potrzeb potencjalnych użytkowników aplikacji, dowiedzenie się, czego właściwie oczekują, i doprowadzenie do dostarczenia produktu o obsłudze intuicyjnej, realizowanej naturalnie, wręcz bez chwili zastanowienia. Zajmuje się więc projektowaniem zorientowanym na człowieka i jego potrzeby.

UX designer zdecydował, że użytkownik będzie wpisywał kroki w polu tekstowym (*textfield*). Poniżej będzie widział listę dostępnych kroków (*textarea*). Następnie zostanie użyty komponent grafiki (*image*). Powyższe ustalenia zostały zaprezentowane na szablonie próbnym makiety (rysunek 7.2).



Rysunek 7.2. Szablon próbny makiety

Analityk po dokonaniu analizy wymagań i skonsultowaniu się z architektem IT tworzy wytyczne, na podstawie których konfigurowane jest środowisko testowe.

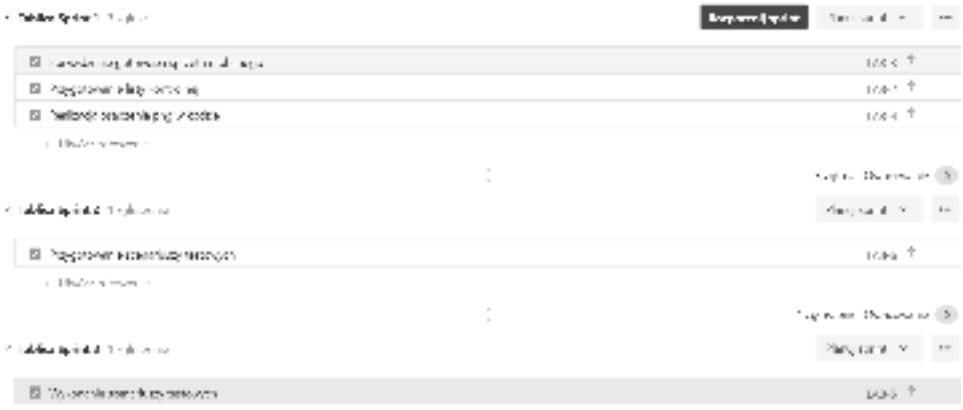
Następnym krokiem jest wspomniana już wyżej (w tabeli 7.3) konfiguracja środowiska, w której biorą udział programista i tester. Tester sprawdza, czy stworzone środowisko testowe jest zgodne z wymaganiami, np. czy na urządzeniach mobilnych jest właściwa wersja systemu Android, czy zainstalowane przeglądarki są w wymaganej wersji.

Kontrola wykonanych zadań to rola Project Managera, który korzystając z narzędzia Jira, wykonuje następujące czynności:

1. Wprowadza wszystkie zadania, które mają być wykonane w ramach realizacji projektu (rysunek 7.3).

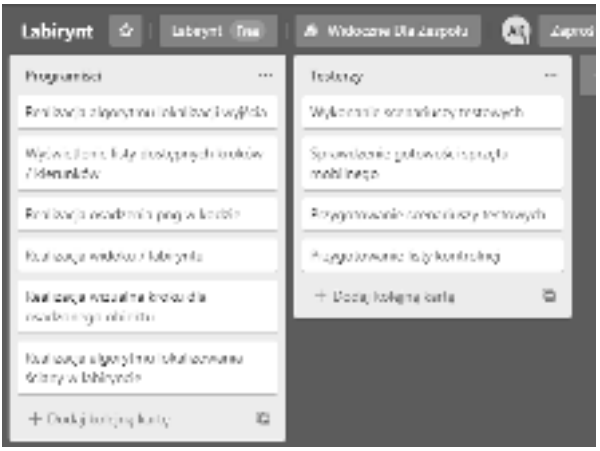
Rysunek 7.3. Przykładowy widok na backlog projektu

2. Następnie dzieli zadania na etapy, w których mają być one realizowane (rysunek 7.4).



Rysunek 7.4. Przykładowy widok backlogu z przydzieleniem zadań do sprintów

3. Po podziale zadań na etapy przystępuje do przydzielania prac konkretnym grupom lub osobom. Ze względu na to, że budżet jest ograniczony, warto sięgać po narzędzia bezpłatne, takie, które nie zaburzą cyklu życia projektu oraz nie utrudnią komunikacji i przejrzystości. Poniżej widać proponowany podział zadań w narzędziu Trello (rysunek 7.5).



Rysunek 7.5. Przykładowy widok na zadania w Trello

7.4.2. Sprint 1.

Kolejnym etapem jest sprint pierwszy. Programiści przystępują do tworzenia oprogramowania, a testerzy przygotowują scenariusze testowe — na podstawie informacji uzyskanych od analityka i Product Ownera.

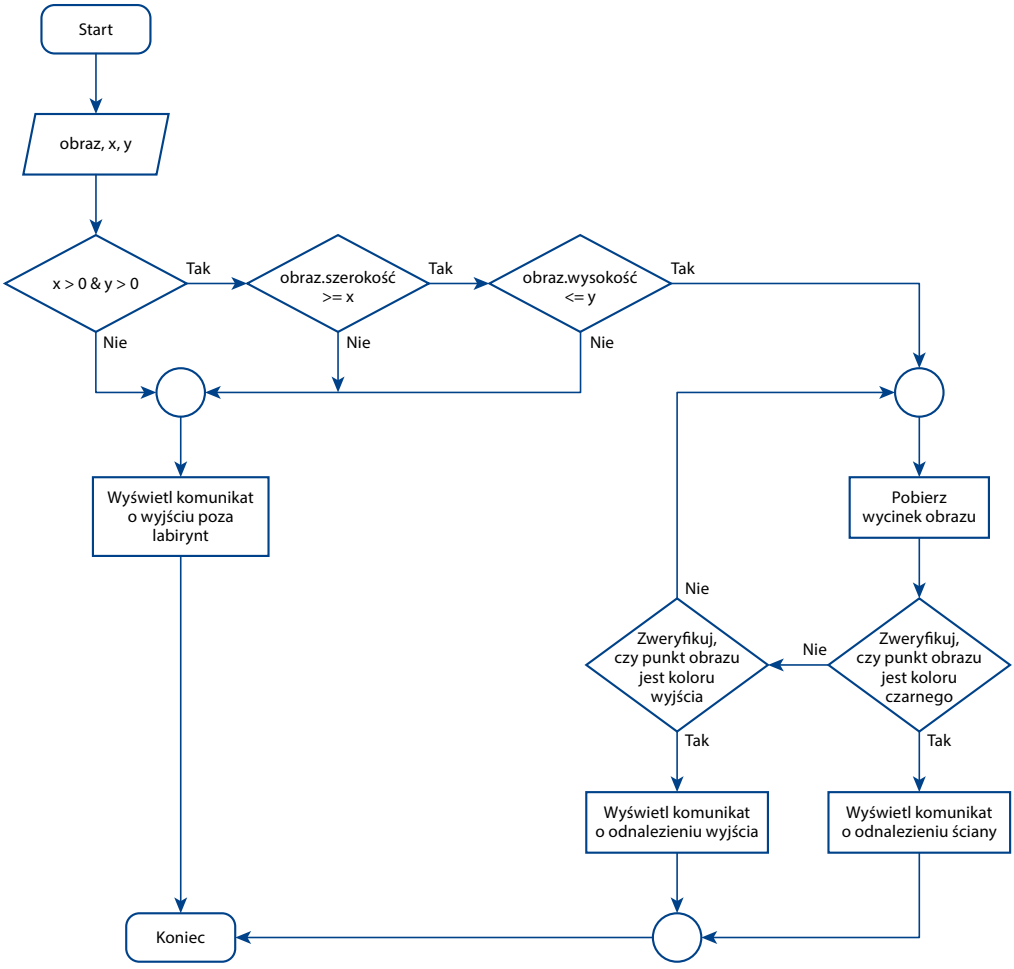
Efektem zakończenia sprintu ma być uzyskanie widoku obiektu, który przemieszcza się w dwu kierunkach: w lewo albo w prawo po wpisaniu — odpowiednio — słowa *left* lub *right*.

Podczas prac programistycznych pojawił się problem — mianowicie obiekt „przechodził" przez ściany labiryntu. Programiści musieli opracować algorytm, który realizował to zadanie.

Na tym etapie stwierdzono, że należy zastosować schemat blokowy do opisu problemu. Lider techniczny projektu zaproponował, by wykorzystać właściwości obrazu (rysunek 7.6).

UWAGA

Kolory w obrazach w postaci cyfrowej reprezentowane są przez liczby określające stopień nasycenia trzech (lub czterech w schemacie CMYK) kolorów składowych: czerwonego (*red*), zielonego (*green*) i niebieskiego (*blue*) — stąd nazwa RGB. Kolor czarny reprezentowany jest w tym schemacie przez wartości — zapisywane w nawiasach, po kolejnych przecinkach — (0, 0, 0), a biały to (255, 255, 255).



Rysunek 7.6. Schemat blokowy odnalezienia wyjścia oraz ściany w labiryncie

Programiści wraz z liderem technicznym znaleźli proste i szybkie rozwiązanie problemu.

Testerzy przygotowali zestawy testów. Nazwa każdego z nich odpowiada za konkretną funkcję (rysunek 7.7).

- *Left* oznacza kierowanie obiektu w lewo.
- *Right* oznacza kierowanie obiektu w prawo.
- *Up* oznacza kierowanie obiektu do góry.
- *Down* oznacza kierowanie obiektu do dołu.
- *Wprowadź kierunek* odpowiada za możliwość nadania obiektowi dowolnego kierunku, wybranego z listy.

Rysunek 7.7. Widok na zestaw testów w narzędziu TestLink

7.4.3. Sprint 2.

Podczas sprintu drugiego prace programistyczne są już mocno zaawansowane. Efekty tego sprintu to:

- dodanie ścian i obsługa ściany w labiryncie,
- implementacja początku i końca labiryntu.

Okazało się, że wytworzony w tym sprincie przez młodszych programistów kod (dostępny jako gałęzie — *branch* — w systemie kontroli wersji) wymagał zoptymalizowania, dlatego wezwano bardziej doświadczonego programistę, który mu się przyjrzał i zaproponował wprowadzenie poprawek w algorytmach. Programista ten, przeglądając kod, zwrócił także uwagę na kilka innych miejsc, które wymagały poprawienia. Po wprowadzeniu wszystkich zgłoszonych przez niego poprawek aplikacja zaczęła działać znacznie szybciej.

Przykład 7.2

Oto przykładowy przegląd kodu (w nomenklaturze programistycznej Code Review, w skrócie CR). Programista napisał funkcję przedstawioną na listingu 7.1.

Listing 7.1

Kod funkcji zwracającej numer planety w Układzie Słonecznym

```
function draw(numb) {
    if(numb == 'Merkury') {
return 1;
    }else if(numb == 'Wenus') {
        return 2;
    }else if(numb == 'Ziemia') {
        return 3;
    }else if(numb == 'Mars'){
        return 4;
    }else if(numb == 'Jowisz'){
return 5;
    }
else if(numb == 'Saturn'){return 6;
    }else if(numb == 'Uran'){
        return 7;
    }
else if(numb == 'Neptun'){
        return 8;
    }
}
```

Podczas przeglądu tego fragmentu kodu drugi programista zgłosił następujące uwagi (to zawsze są uwagi, a nie nakazy; można się do nich odnieść poprzez dyskusję, nie ma obowiązku natychmiastowego poprawiania kodu):

1. Ogólny opis: trudno powiedzieć, co metoda `draw` ma robić; z kontekstu mogę wnioskować, że zwraca numer kolejnych planet, licząc od Słońca.
2. Brak weryfikacji poprawności wprowadzonych danych (co zrobi program, jeśli wprowadzi się liczbę albo ciąg znaków inny niż w kolejnych instrukcjach warunkowych?).
3. Dlaczego nie użyto tablicy? Byłoby prościej i szybciej.
4. Brak formatowania (wystarczy domyślne, za pomocą skrótu klawiszowego).

UWAGA

Zastrzeżenie wyrażone w punkcie drugim najprawdopodobniej zostanie sformułowane w języku technicznym: „Brak walidacji wprowadzanych danych”.

Jak widać, uwagi są oczywiste. Warto zadbać o to, by kod zawsze był weryfikowany przez drugą osobę, bo pozwoli to znaleźć rzeczy, które nie zostały zauważone przy pierwszej implementacji.

Uwagi zgłaszane w procesie Code Review nie mogą być nacechowane negatywnie, powinny być formułowane spokojnie i bez emocji. Ich celem nie jest wyśmianie czy pognębienie mniej doświadczonego kolegi, ale doprowadzenie do udoskonalenia jego umiejętności technicznych — a często także umiejętności przeprowadzającego przegląd.

UWAGA

Regularne przeglądy kodu znacznie zwiększają efektywność pracy programistów. Dzięki wymienianym uwagom dostrzegają oni więcej niedociągnięć, tworzą lepszy i czytelniejszy kod, w naturalny sposób dzielą się wiedzą i umiejętnościami, co ma ogromne znaczenie także przy kolejnych projektach. W niektórych zespołach niesprawdzony kod nie może wejść nawet do wersji testowych aplikacji.

W tym sprincie, na specjalnym, poświęconym temu spotkaniu, po raz pierwszy zaprezentowano klientowi aplikację. Klient uznał, że:

- Aplikacja spełnia założone wymagania.
- Wybranie formuły labiryntu jest trafne, bo dzieci lubią szukać wyjścia.

Zgłosił jednak także pewne zastrzeżenia i propozycje:

- Poprosił, by dzieciom na wysokości ich wzroku wyświetlała się lista dostępnych kroków.
- Uznał, że dostępnych jest za mało kierunków.
- Zasugerował, że dobrze byłoby ładować kilka labiryntów, w zależności od wieku przedszkolaka.

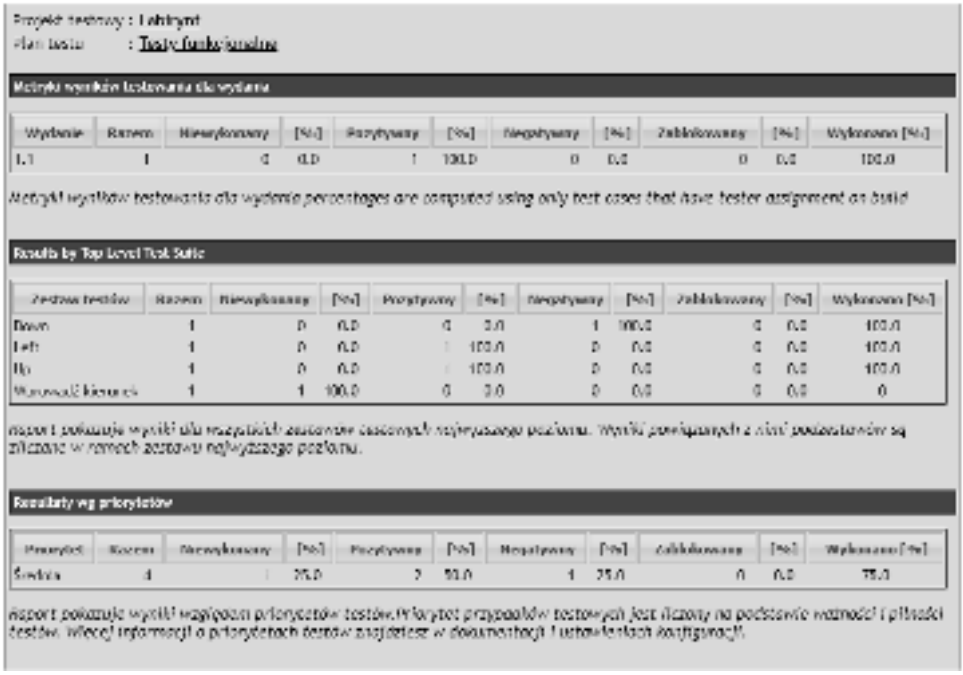
Na takich spotkaniach demonstracyjnych jest obecny analityk, który ustalał założenia z klientem. Po dyskusji pomiędzy analitykiem a klientem uzgodniono, że ostatnie dwa punkty nie mieszczą się w pierwotnych założeniach, a zatem mogą być realizowane jako poprawka dodana już po wdrożeniu aplikacji (tzw. *fix*).

W kolejnym sprincie testerzy przystępują do wykonania testów gotowych już funkcji. Wyniki testów zaznaczane są w narzędziu TestLink, a raport z ich wykonania udostępniany jest całemu zespołowi. Przykładowe zestawienie wykonanych testów przedstawia rysunek 7.8.

UWAGA

W przypadku dużych projektów, z budżetem pozwalającym na wykonanie automatyzacji testów, warto mieć na uwadze narzędzia, w których można taką automatyzację przeprowadzić. Niezbędne kompetencje do wykonania testów automatycznych ma tester automatyzujący lub programista.

Obecnie najpopularniejszym narzędziem do automatyzacji jest Selenium.



Rysunek 7.8. Przykładowe zestawienie wykonanych testów w narzędziu TestLink

7.4.4. Sprint 3.

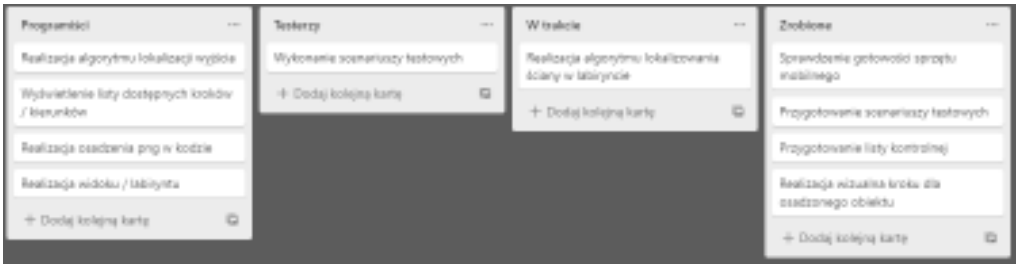
W ramach prac na tym etapie odbywają się pełne testy, zarówno funkcjonalne, jak i niefunkcjonalne.

UWAGA

Część testów niefunkcjonalnych została zaprezentowana w rozdziale 4. Pozostałe zaliczające się do tej grupy to:

- Testy użyteczności — testy, podczas których sprawdza się łatwość korzystania z oprogramowania bądź to, jak szybko użytkownicy uczą się z niego korzystać.
- Testy pielęgnowalności — następuje w nich sprawdzenie, czy oprogramowanie łatwo się modyfikuje lub czy łatwo dostosować je do nowych wymagań.
- Testy niezawodności — w ich trakcie określamy niezawodność naszego oprogramowania. Sprawdzenie oprogramowania odbywa się poprzez wykonanie wymaganych funkcji w określonych warunkach.
- Testy przenośności — proces testowy, dzięki któremu określamy, jak łatwo oprogramowanie może być przeniesione z jednego środowiska na drugie.

Wyniki testów funkcjonalnych są rejestrowane w narzędziu TestLink. Każde wykonane zadanie jest przenoszone na kolejną docelową tablicę w narzędziu Trello (rysunek 7.9).



Rysunek 7.9. Przykładowy widok zadań w Trello

Prace programistyczne zostały zakończone sukcesem. Błędy krytyczne nie zostały znalezione.

Po zakończeniu prac programistycznych oraz wykonaniu testów przyszedł moment prezentacji ostatecznej wersji aplikacji klientowi. Najczęściej w takich spotkaniach uczestniczą jedynie klient i dwie osoby reprezentujące firmę realizującą projekt.

Na spotkaniu podsumowującym wszystkie sprinty uzgodniono, że udało się zmieścić w budżecie, co było kluczowe dla klienta, a aplikacja dla dziecka w wieku 3–4 lat pozwoli mu nie tylko poznać podstawy języka angielskiego, ale także zaznajomić się z kierunkami, co ma ogromne znaczenie na dalszych etapach edukacji.

Zgodnie ustalono, że wygląd aplikacji (*design*) należy dopracować, trzeba też wprowadzić kolejne kierunki (lewy skos, prawy skos).

UWAGA

Najważniejsza nauka, jaka płynie z realizacji tego projektu, jest następująca: projekt rzadko bywa zrealizowany w stopniu w pełni satysfakcjonującym wszystkie strony. Najczęściej przyjęte rozwiązanie jest formą kompromisu. Rzadko kiedy udaje nam się zrealizować początkowe założenia w stu procentach i czasem trzeba pójść na ustępstwa, mając na uwadze, że to zadowolenie klienta jest ostatecznie najważniejsze.

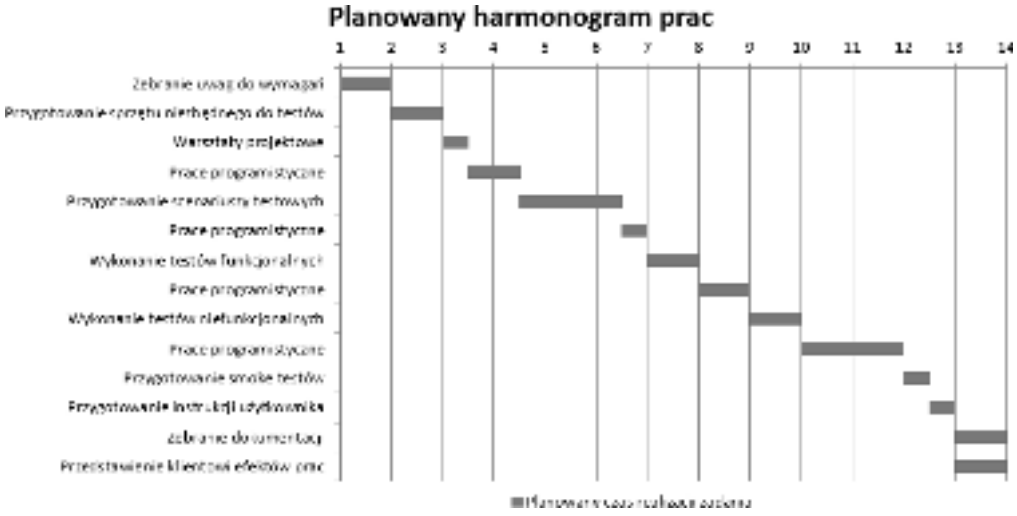
Wiele projektów można wykorzystywać także w późniejszym czasie. Na przykład grę Labirynt można później sprzedać przychodni jako rozwiązanie dla dzieci oczekujących w kolejce do lekarza.

7.5. Diagram Gantta — charakterystyka, przykłady

Skuteczna realizacja projektów czy też złożonych zadań wymaga jasnej komunikacji i przejrzystego podziału prac. Jednym ze sposobów na stworzenie precyzyjnego harmonogramu pracy jest tzw. diagram Gantta, znany już od końca XIX wieku.

Diagram Gantta to nic innego jak graficzne zobrazowanie harmonogramu prac w projekcie czy zadaniu. Konstrukcja diagramu Gantta to kaskadowe zestawienie zadań, operacji, czynności lub procesów z czasem ich trwania. Na osi X znajduje się zazwyczaj oś czasu. Jednostki czasu, jakimi się posługujemy, zależą od tego, co obrazujemy — mogą to być minuty, godziny, ale też dni, tygodnie, miesiące czy lata.

Na rysunku 7.10 widoczny jest przykładowy diagram Gantta wykonany w Excelu. Oś pozioma to właśnie oś czasu wyrażona w dniach (zadania są realizowane od 1. dnia miesiąca i trwają do 14. dnia miesiąca). Oś pionowa to kolejne zadania.



Rysunek 7.10. Przykładowy diagram Gantta

Konstrukcja wykresu Gantta nie jest niczym skomplikowanym. Informacje i dane potrzebne do jego sporządzenia to:

- nazwa czynności/zadania/działania/operacji;
- czas rozpoczęcia czynności/zadania/działania/operacji;
- czas trwania czynności/zadania/działania/operacji.

A oto główne korzyści wynikające ze stosowania wykresu:

- Wizualizacja pomaga ułożyć sekwencję czynności do wykonania dla danej operacji i dostarcza informacji o tym, ile czasu powinna trwać.

- Widok harmonogramu pozwala szybko namierzyć wąskie gardła w procesie (jeżeli oczywiście wykres dotyczy czasów trwania procesu).
- Zyskuje się bardzo przejrzysty sposób przekazania planu prac. Jego odczytanie i zrozumienie nikomu nie powinno przysporzyć trudności.

7.6. Proponowane rozszerzenia aplikacji

Wróćmy do naszej gry Labirynt. Jej najprostszą wersję (można ją pobrać z repozytorium pod adresem: *https://github.com/weronikakortas/labyrinth*) można śmiało rozbudować, chociażby dodając funkcję zliczania i zapisywania wyników.

Czas, jaki upłynął od chwili, gdy został pierwszy raz uruchomiony obiekt przesuwany po labiryncie, aż do momentu wyjścia, można policzyć, wykorzystując obiekt `Date` w JS. Jest w nim m.in. funkcja `now()`, która wyświetla bieżący czas. Ustawmy więc czas początkowy na czas pobrany przy użyciu tej właśnie funkcji. Uzyskamy to za pomocą polecenia:

```
const startTime = Date.now();
```

W celu zmierzenia czasu wykonania zadania obliczamy różnicę pomiędzy czasem początkowym a tym, który zwróci funkcja `now()` w chwili wyjścia z labiryntu:

```
const elapsedTime = startTime - Date.now();
```

UWAGA

Czas mierzony za pomocą funkcji `now()` zwracany jest w milisekundach. Aby zaprezentować go użytkownikowi, musielibyśmy przekonwertować go na jakąś formę bardziej czytelną dla człowieka.

UWAGA

W C++ możemy zaimplementować rozwiązanie, wykorzystując bibliotekę `chrono`:

```
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

int main()
{
```

→

UWAGA cd.

```
        auto start = high_resolution_clock::now();
        // kod z implementacją od wejścia do labiryntu do wyjścia z niego

        auto stop = high_resolution_clock::now();

        auto duration = duration_cast<microseconds>(stop - start);

        cout << "Czas wyjścia z labiryntu to "
             << duration.count() << " microseconds" << endl;

        return 0;
    }
```

Skoro policzyliśmy czas przejścia labiryntu, to teraz dobrze byłoby go zapisać.

Sposobów na zrobienie tego jest sporo. Opiszemy tutaj tylko jeden, podstawowy.

Możemy użyć tablicy wielowymiarowej, w której w pierwszej komórce przechowywane będzie imię gracza, a w kolejnej znajdzie się najlepszy uzyskany wynik (listing 7.2):

Listing 7.2

Zapisanie wyniku w tablicy wielowymiarowej

```
let scores = [
    ['Jan', 1.2],
    // ...pozostali gracze
    ['Michal', 3.4]
]
```

UWAGA

W C++ możemy wykorzystać zapisywanie danych do pliku w formacie CSV, czyli takiego, w którym dane są rozdzielone wskazanym przez nas wyróżnikiem.

```
#include <fstream>
#include <string>
Using namespace std;
int main(){
    string name = "Jan";
    int score = 1.2;
    fstream plik("plik.txt", ios::out);
```

UWAGA cd.

```
        if(plik.good()) {
            plik << name << "; " << score;
            plik.flush();
            plik.close();
        }
        return(0);
    }
```

W tym przypadku zdecydowaliśmy, że wyróżnikiem będzie średnik.

Równie ciekawym rozwiązaniem jest wykorzystanie struktury w C++. Struktura to zestaw właściwości różnego typu.

```
#include <iostream>
#include <string>
using namespace std;
struct Player  // deklaracja struktury
{
     // opis struktury
    string name;
    double score;
};
int main()
{
     // tworzenie obiektu struktury i wypełnianie
    Player jan =
    {
        "Jan",
        1.2
    };

    Player michal =
    {
"Michał",
        3.4
    };
   // wyświetlenie obiektów
```

UWAGA cd.

```
        cout << "Oto lista graczy wraz z ich wynikami"
        << endl << "imię " << jan.name
        << endl << "wynik " << jan.score
        << endl
        << endl
        << endl << "imię " << michal.name
        << endl << "wynik " << michal.score;
        cin.get();
        return(0);
    }
```

Struktura to w pewnym sensie mały obiekt. Dlaczego mały? Bo ma o wiele mniej możliwości i funkcji niż tradycyjny obiekt. Jednakże ma pewną istotną zaletę — zużywa mniej pamięci w komputerze.

UWAGA

Możliwości rozbudowania naszej aplikacji jest bardzo dużo. Z czasem zapewne zauważysz (i będziesz w stanie samodzielnie opracować) funkcje, o które warto ją wzbogacić.

7.7. Prawa autorskie

Przygotowując się do pracy programisty, warto poznać swoje prawa w zakresie pracy wytwórczej. Jest taki rodzaj prawa, które specjalizuje się w zabezpieczaniu wytworów naszej pracy — to prawo autorskie. Podstawa prawna o prawie autorskim i prawach pokrewnych to Ustawa z dnia 4 lutego 1994 r., opublikowana w Dzienniku Ustaw nr 90, w pozycji 631.

DEFINICJA

Prawa autorskie to zespół norm i aktów prawnych mających na celu zabezpieczenie wytworów ludzkiej działalności przed nielegalnym rozpowszechnianiem, kopiowaniem lub czerpaniem z nich korzyści majątkowych.

Prawa autorskie dzielą się na:

- Autorskie prawa osobiste — prawa do powiązania nazwiska autora z jego dziełem. Prawa te nigdy nie wygasają i są z natury rzeczy niezbywalne. Nie można się ich zrzec ani przenieść na inną osobę.

- Autorskie prawa majątkowe — prawa zabezpieczające interesy twórców utworów albo wydawców, którzy nabyli od autora prawa majątkowe do rozporządzania tymi utworami. Prawa te wygasają po 70 latach od śmierci autora lub po 70 latach od rozpowszechnienia utworu, gdy autor nie jest znany.
- Prawa pokrewne — prawa do artystycznych wykonań, fonogramów i wideogramów, nadań programów, pierwszych wydań oraz wydań naukowych i krytycznych.

UWAGA

W jakim celu wprowadzono prawa autorskie?

Prawa autorskie wprowadzono po to, aby chronić interesy twórców i wydawców.

Co nie podlega prawu autorskiemu?

Ochronie nie podlegają proste informacje prasowe lub informacje pochodzące od instytucji:

- informacje o wydatkach,
- prognozy pogody,
- kursy walut,
- programy radiowe i telewizyjne (godzinowe spisy programów zamieszczane w gazetach),
- repertuary kin i teatrów,
- ustawy i ich projekty, a także dokumenty urzędowe i materiały, znaki i symbole oraz opublikowane opisy patentowe lub ochronne.

Kiedy mówimy o naruszeniu praw autorskich?

Mówimy o tym np. w przypadku likwidacji podpisów zdjęć, udostępnionych np. w ramach darmowych kolekcji na stronie internetowej, oraz tzw. cyfrowych „znaków wodnych", np. z logotypem autora. Nawet jeśli wymienione materiały są udostępnione darmowo, nie zwalnia nas to od respektowania osobistych praw autorskich twórcy.

Tak samo jest w przypadku skryptów w programowaniu. Można z nich korzystać w swoich programach, o ile autor wyraził na to zgodę, jednak likwidacja komentarzy z jego nazwiskiem lub adresem e-mailowym jest naruszeniem praw autorskich.

Naruszeniem praw autorskich jest też samowolne wprowadzenie utworu muzycznego do sieci w postaci pliku MP3 bez zgody osoby uprawnionej. Aby legalnie rozpowszechniać utwory w internecie, należy zawsze uzyskać zgodę autora, producenta lub innej osoby uprawnionej do utworu.

Jakie szkody może wyrządzić naruszenie praw autorskich? Przede wszystkim są to:

- utrata przez twórców zysków z tytułu rozpowszechniania utworów;
- straty firm zajmujących się dystrybucją i promocją utworów;
- straty państwa związane z nieodprowadzonymi podatkami.

7.7.1. Licencja

Licencja to nic innego jak umowa między producentem (autorem lub wydawcą) a użytkownikiem dotycząca zasad użytkowania produktu. Wyroby objęte licencją mają instrukcję instalacji oraz dają prawo do bezpłatnych porad i serwisu, łącznie z aktualizacjami. Legalnie zakupiony program ma numer licencyjny (przekazywany w formie drukowanej lub elektronicznej) służący do identyfikacji. Przed rozpoczęciem korzystania z oprogramowania może być wymagana rejestracja lub aktywacja produktu.

W tabeli 7.4 zestawiono typy licencji wraz z ich charakterystykami.

Tabela 7.4. Zestawienie typów licencji

Rodzaj licencji	Charakterystyka
Freeware	Programy, które można wykorzystywać, a także rozpowszechniać za darmo. Nie wolno ich jednak sprzedawać ani dokonywać w nich zmian, np. poprzez umieszczanie wewnątrz nich własnej reklamy — mogą być rozpowszechniane wyłącznie w niezmienionej formie.
Public domain	Licencja czyniąca z oprogramowania własność ogółu. W jej myśl autor (lub autorzy) oprogramowania zrzeka się praw do upowszechniania oprogramowania na rzecz ogółu użytkowników. Dozwolona jest dalsza dystrybucja oprogramowania bez zgody autora.
Shareware	Oprogramowanie udostępniane bezpłatnie do testów. Każdy potencjalny nabywca, przed podjęciem decyzji o zakupie, może gruntownie sprawdzić w działaniu zazwyczaj w pełni funkcjonalną wersję.
GNU GPL (ang. *General Public License*)	Zasady licencyjne określone przez konsorcjum *Free Software Foundation*. Jeśli ktoś wprowadza do obiegu oprogramowanie zawierające jakąkolwiek część podlegającą licencji GPL, to musi udostępnić wraz z każdą dystrybucją binarną jej postać źródłową.
Licencja grupowa	Licencja zezwalająca na użytkowanie oprogramowania w sieci lub w zestawie komputerów, np. w szkole lub w pracowni, określająca maksymalną liczbę stanowisk, na których wolno zainstalować objęte nią oprogramowanie.
Licencja jednostanowiskowa	Licencja uprawniająca użytkownika do zainstalowania nabytego oprogramowania tylko w jednym komputerze, obejmująca zakaz udostępniania takiego oprogramowania w sieci oraz na innych własnych komputerach.
Licencja na obszar	Umowa między producentem oprogramowania a nabywcą uprawniająca tego drugiego do sporządzenia określonej liczby kopii zakupionego oprogramowania na swój własny użytek. Takie rozwiązanie jest często stosowane przez firmy korzystające z sieci lokalnych LAN.

7.7.2. Własność intelektualna

Omówiliśmy już rodzaje praw autorskich, teraz warto wyjaśnić, czym są elementy własności intelektualnej. Otóż są to prawa dotyczące *dóbr niematerialnych*. Termin ten stosowany jest w różnych działach prawa regulujących zasady korzystania z tzw. własności intelektualnej.

Podstawowym celem, ku któremu dąży większość aktów prawnych dotyczących własności intelektualnej (z wyjątkiem znaków towarowych), jest „promowanie postępu". Według tej filozofii dzięki wymianie ograniczonych oraz wyłącznych praw do przedstawiania koncepcji i utworów zarówno społeczeństwo, jak i właściciel patentu lub praw autorskich czerpią zyski, co jednocześnie stanowi motywację dla wynalazców i twórców.

Zakres ochrony poszczególnych praw własności intelektualnej w Polsce jest różny w zależności od charakteru tych praw.

- Dobra objęte prawem autorskim są chronione od momentu ich powstania bez konieczności ich rejestracji. Na podstawie umów międzynarodowych ochrona ta obowiązuje w prawie wszystkich krajach świata.
- W przypadku większości przedmiotów własności przemysłowej (rodzaj praw wyłącznych wynikających z narodowego, międzynarodowego lub regionalnego ustawodawstwa; prawa te należy rozumieć nie tylko w kontekście przemysłu, ale także handlu i rolnictwa lub regionalnego ustawodawstwa) dla uzyskania pełnej ochrony prawnej wymagane jest zgłoszenie do Urzędu Patentowego RP oraz wydanie przez ten organ decyzji w sprawie udzielenia patentu, praw ochronnych lub praw z rejestracji. Zakres tej ochrony jest ograniczony do terytorium RP, a ewentualne jej rozszerzenie wymaga zgłoszeń w urzędach patentowych krajów, w których prawa mają być objęte taką ochroną.

7.8. Zakończenie

Ten rozdział wieńczy początek Twojej przygody z tworzeniem oprogramowania. Znasz już podstawy programowania, wiesz, jak testować aplikację. Znasz też zasady organizacji pracy nad projektem. Na pewno pozwoli Ci to lepiej zrozumieć pracę w branży IT i — być może — w przyszłości sprawnie się w nią wdrożyć.

Mamy nadzieję, że udało nam się zaszczepić w Tobie ciekawość dotyczącą technologii, podejścia projektowego czy też doboru narzędzi wspierających proces.

I pamiętaj — ani analityk, ani tester, ani programista nie powinni stać w miejscu! Są to zawody, które wymagają nieustannego poszerzania posiadanej wiedzy i zdobywania nowej.

7.9. Zadania

Zadanie 7.1

Wypisz kolejne uwagi do kodu dotyczącego planet.

Zadanie 7.2

Wypisz cel sprintu 4. oraz zadania dla poszczególnych ról.

Zadanie 7.3

Dobierz drugą osobę do zespołu i podzielcie się rolami: jedno z Was niech będzie klientem, drugie przedstawicielem firmy wykonawczej. Na podstawie charakterystyki zamówienia złożonego przez klienta przedstawiciel firmy powinien opracować listę zadań dla poszczególnych ról w projekcie. Przyjmijmy, że klient nalega, by realizacja projektu trwała krócej, niż wynika to z harmonogramu sporządzonego przez firmę wykonawczą.

Zadanie 7.4

Na podstawie informacji z rozdziału utwórz aplikację labiryntu.

Bibliografia

Certyfikowany tester. Sylabus poziomu podstawowego ISTQB®. Wersja 2018 V 3.1. Prawa autorskie wersji polskiej zastrzeżone dla © Stowarzyszenie Jakości Systemów Informatycznych (SJSI).

Certyfikowany Tester. Sylabus rozszerzenia poziomu podstawowego. Tester zwinny. Wersja 2014.

Chrapko M., *Scrum. O zwinnym zarządzaniu projektami*, Helion, Gliwice 2012.

Gamma E., Helm R., Johnson R., Vlissides J., *Wzorce projektowe. Elementy programowania obiektowego wielokrotnego użytku*, Helion, Gliwice 2017.

Kaczor K., *Scrum i nie tylko. Teoria i praktyka w metodach Agile*, Wydawnictwo Naukowe PWN, Warszawa 2014.

Pytel K., Osetek S., *Systemy operacyjne i sieci komputerowe. Część 1*, WSiP, Warszawa 2010.

Pytel K., Osetek S., *Systemy operacyjne i sieci komputerowe. Część 2*, WSiP, Warszawa 2010.

Słownik terminów testowych ISTQB®. Wersja 3.3.1. Prawa autorskie wersji polskiej zastrzeżone dla © Stowarzyszenie Jakości Systemów Informatycznych (SJSI).

Tomasiewicz J., *Zaprzyjaźnij się z algorytmami*, Wydawnictwo Naukowe PWN, Warszawa 2016.

Akty prawne

Ustawa z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych, Dz. U. 1994 Nr 24 poz. 83.

Źródła internetowe

https://agilemanifesto.org/iso/pl/principles.html

https://devdocs.io/javascript/

https://hygger.io/blog/how-users-see-kanban/

https://leanactionplan.pl/wykres-gantta/

https://miroslawzelent.pl/informatyka/licencje-programow-prawa-autorskie/

https://pl.wikipedia.org/

https://wearecreativelabs.com/waterfall-vs-agile-roznice-pomiedzy-tradycyjnym-a-zwinnym-podejsciem-do-zarzadzania-projektem/

https://www.grupa-tense.pl/blog/ux-to-nie-ui-roznice-miedzy-user-interface-a-user-experience/

https://www.nexio.pl/blog/scrumban/

https://www.w3schools.com/

Skorowidz

J

K

L

M

N

O

P

R

S

Ś

T

U

W

Y

Z