

Michał Sobczak

# JAKOŚĆ OPROGRAMOWANIA



PODRĘCZNIK DLA PROFESJONALISTÓW

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Jan Paluch

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Helion SA

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: [helion@helion.pl](mailto: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/jakosc>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:

<ftp://ftp.helion.pl/przyklady/jakosc.zip>

ISBN: 978-83-283-6102-7

Copyright © Helion 2020

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

<b>Wprowadzenie .....</b>	<b>13</b>
<b>Rozdział 1. Praca u podstaw .....</b>	<b>23</b>
Klasyfikacja usterek, błędów i awarii .....	23
Terminologia niezawodności .....	24
Błędy .....	27
Zapobieganie defektom .....	28
Standaryzacja i organizacja kodu .....	28
Standaryzacja .....	28
Organizacja .....	28
Jakość danych .....	29
Wzorce projektowe, dowodzenie poprawności .....	30
Kryteria wzorców projektowych .....	31
Jakość poszczególnych wzorców projektowych .....	31
Uzasadnienie dla wzorców projektowych .....	31
Korzyści ze stosowania wzorców projektowych .....	32
Trudności .....	33
Wzorce oprogramowania nie są... ..	33
Wzorce oprogramowania są... ..	33
Rodzaje wzorców projektowych .....	33
Konwencja ponad konfigurację .....	59
Niezawodność .....	59
Weryfikacja, walidacja i testowanie .....	59
Odporność na błędy .....	60
Przeglądy jakości kodu .....	62
Przegląd konwencyonalny .....	62
Przegląd jako proces .....	63
Inspekcja oprogramowania krytycznego .....	63

Poziomy pokrycia kodu testami .....	64
Właściwy wybór architektury (DDD) i projektowanie .....	64
Prostota i minimalizm .....	65
Złożoność .....	65
Architektura .....	66
Projektowanie .....	66
Programowanie .....	69
Język programowania .....	71
Maksymy programistyczne .....	71
Metodyki (TDD, BDD, TIP) .....	72
TDD .....	72
BDD .....	73
TIP .....	74
Aplikacje mikrousługowe a monolityczne .....	74
Aplikacje monolityczne .....	74
Aplikacje mikrousługowe .....	75
Zasady testowania .....	75
Pojęcia .....	75
Styl testowania .....	76
Cel .....	76
Testowanie specyfikacji .....	76
Retrospekcja .....	76
Sytuacje .....	76
Filozofie testowania .....	77
Dane testowe .....	77
Kierunki testowania .....	77
Minimum .....	77
Aksjomaty testowania .....	78
Aksjomaty programistyczne .....	78
Pragmatyczny programista .....	79
Entropia oprogramowania .....	79
DRY .....	79
Inne aspekty .....	79
Błędotwórstwo .....	79
Dowodzenie poprawności programów .....	79
Przewidywanie zmian .....	80
Dyspozycyjność systemu .....	80
Uszkodzenia i powrót ze stanu błędu .....	81

Metodologia programowania .....	82
Kolejność celów .....	83
Wykonalność .....	83
Starzenie .....	84
Defekty w grach .....	84
Room bounds, problemy systemu optymalizacji .....	84
Problemy z teksturami .....	87
Brakujące elementy otoczenia .....	89
Błędna lokalizacja elementów otoczenia .....	90
<b>Rozdział 2. CI/CD .....</b>	<b>93</b>
Składnia, kompilacja i budowanie artefaktów .....	93
Składnia .....	93
Kompilacja .....	94
Budowanie .....	94
Automatyczne testy programistyczne .....	95
minitest .....	95
busted .....	98
Pozostałe przykłady testów .....	100
Integracyjne .....	100
UI .....	103
Specyfikacja, konfiguracja, środowisko .....	104
CI/CD w praktyce .....	105
Git branching .....	106
.gitlab-ci.yml .....	106
<b>Rozdział 3. Produkcja .....</b>	<b>109</b>
Współpraca pierwszej i drugiej linii wsparcia .....	109
Monitoring .....	109
APM .....	110
Baza danych .....	114
Błędy .....	117
Logi .....	122
Obsługa awarii .....	124
Hotfixing .....	124
Monkey-patching .....	124

<b>Rozdział 4. Błędy i komunikaty .....</b>	<b>125</b>
Zestawienie błędów popełnianych przez programistów .....	125
Typowe błędy .....	125
Błędy początkujących .....	126
Kategoryzacja błędów .....	127
Kompozycja programu .....	127
Pętle .....	128
Dane .....	128
Zmienne .....	128
Tablice .....	128
Operacje arytmetyczne .....	128
Podprogramy .....	128
Inne .....	129
 <b>Rozdział 5. Przypadki .....</b>	 <b>131</b>
W zasięgu wzroku .....	131
Monitoring .....	131
Planowanie .....	131
Spotkanie .....	131
Potrzeba .....	132
Życie .....	132
Konsumpcja .....	132
Rozrywka .....	132
Finanse .....	132
Nauka .....	133
Siły na zamiary .....	133
Odbiorca .....	133
Zamawiający .....	133
Beneficjent .....	133
Wykonawca .....	134
Operator .....	134
Użytkownik końcowy .....	134
Niedoskonałość .....	134
Człowiek .....	134
Organizacja .....	135
Czas .....	135
Ważność .....	135
Pilność .....	135
Mikrozarządzanie .....	135

---

Zadania cykliczne .....	136
Zmiana częstotliwości .....	136
Racjonalizacja operacji .....	137
Uruchamianie w Dockerfile .....	138
Nieporządek w harmonogramie zadań .....	138
Przykład .....	138
Rozwiązania .....	139
Skutki i działania .....	140
Martwy kod .....	140
Współdzielenie kodu .....	141
Rozwiązanie .....	141
Rails .....	141
Java .....	143
Porządkowanie struktury aplikacji .....	143
Timeout .....	144
Przykład .....	144
Rozwiązanie .....	145
Racjonalizacja serwera bazy danych .....	145
Przykład .....	146
Replikacja czasu rzeczywistego .....	146
Aplikacje .....	146
Monitoring .....	147
Nieprawidłowe rekordy .....	148
Typy danych jako klucze .....	148
Nadmiarowe przypisania .....	149
Agregacja danych .....	149
Dzielenie zapytań SQL .....	150
Testy inwazyjne .....	151
Cargo cult .....	152
Prokrastynacja .....	153
Obustronna weryfikacja .....	154
Zewnętrzne dane .....	154
Klucze obce a projekt bazy .....	155
Korekty danych .....	156
Spójność środowisk .....	157
NULL .....	157
Zmęczenie .....	159
Brak testów .....	160
Nemawashi .....	160
„Geniusz zła” .....	161

Paradoks hazardzisty .....	162
Czytelność kodu .....	162
Rozmiar .....	162
Dobre praktyki .....	165
Standardy, czyli XYZ-way .....	166
Kosztowne błędy .....	166
NASA Mars Climate Orbiter .....	167
Ariane 5 Flight 501 .....	167
EDS Child Support System .....	167
Heathrow Terminal 5 Opening .....	168
The Mariner 1 Spacecraft .....	168
Patriot Missile Error .....	168
Pentium FDIV Bug .....	168
Ciekawe przypadki .....	169
Proxy .....	169
Planowanie .....	170
Macierz pokrycia zmianami .....	170
IDE a edytor tekstowy .....	171
Architektura systemowa .....	172
Systemy wsadowe o zdalnym dostępie .....	172
Systemy zbierania danych .....	173
System prosty (Simplex) .....	173
System prosty z wysuniętą transmisją .....	173
System prosty z oddzielnymi maszynami do obsługi transmisji i do zarządzania bazą danych .....	175
System nadrzędny – podległy .....	175
System o wspólnej pamięci zbiorów .....	176
Systemy zdwojone i dualne .....	177
Systemy wieloprocesorowe .....	177
Złożoność .....	178
Predyspozycje osobowe .....	179
Auto DB reconnect .....	180
ESB non-block .....	180
DB Deadlock .....	181
Liczba zgłoszeń .....	183
Konteneryzacja .....	183
Koncentracja i pośpiech .....	184
System transakcyjny a system wsadowy .....	184
Pair programming .....	184
Integracja .....	185



Estymaty .....	185
Kompatybilność API .....	185
Retro computing .....	187
Oprogramowanie samodostosowujące się .....	187
<b>Rozdział 6. Podsumowanie .....</b>	<b>189</b>
Niechronne .....	189
Organizacja .....	189
Post factum .....	189
Spokój .....	190
Co dalej? .....	190
Refleksja .....	190
Na koniec .....	191
<b>Dodatek A. Metodyka prewencji i szybkiego reagowania .....</b>	<b>193</b>
Prewencja .....	193
1. Zasada ograniczonego zaufania .....	193
2. Statyczne typowanie .....	193
3. Analiza składniowa .....	195
4. Dane w relacyjnej bazie .....	195
5. Aspekty projektowe .....	197
Reagowanie .....	197
1. Przechwytywanie wyjątków .....	197
2. Service Discovery .....	198
3. Używajmy własnych produktów .....	198
<b>Bibliografia .....</b>	<b>199</b>
<b>Skorowidz .....</b>	<b>203</b>



## Rozdział 2.

# CI/CD

## Składnia, kompilacja i budowanie artefaktów

### Składnia

Pracując z językami interpretowanymi, takimi jak Ruby czy Python, prędzej czy później dostrzeżemy potrzebę ściślejszego sprawdzania poprawności składni kodu, który napisaliśmy. W przypadku języków podlegających wprost pod procedury kompilacji do kodu wynikowego lub pośredniego, jak choćby Java czy C, problem ten oddalimy do momentu uruchomienia kodu i stosowania ewentualnych funkcji dynamizujących te języki semantycznie. W większości jednak przypadków proces kompilacji i łączenia kodu obejmie większość potencjalnych problemów występujących w źródłach i powiązanych bibliotekach. W językach wymienionych na początku sytuacja wygląda nieco inaczej. Kompilacja i zarazem weryfikacja kodu nie są od nas wymagane. Możemy je wykonać, przykładowo:

```
ruby -c app.rb
```

Jeśli wystąpią problemy natury składniowej, zostanie nam to oznajmione. Jeśli jednak będziemy stosować *eval* czy funkcje modularyzacyjne, to powinniśmy się skupić na pozostałych obszarach zapewnienia jakości lub przeprojektować kod tak, aby nie korzystał z tego rodzaju zabiegów. Tak czy inaczej, warto wprowadzić sprawdzanie składni nie tylko do codziennej praktyki lub stosować takie IDE, które tę weryfikację wykonuje za nas. Warto to samo sprawdzenie składni wykonywać jako pierwszy krok podczas naszego CI. Jeśli bowiem składnia jest nieprawidłowa, nie ma potrzeby uruchamiania testów lub — co gorsza — wdrażania takiego kodu do kolejnych środowisk. Przykładowo:

```
check_syntax:
  image: ruby:2.5.5
  stage: build
  script:
    - for i in `find | grep "\.rb$" `; do ruby -c $i; done
```

Warto oczywiście pokryć tym wszystkie obsługiwane przez nas aplikacje. Nie oczekujemy, że programiści rzeczywiście popełniają tego rodzaju błędy. Jeśli jednak w sytuacji stresowej ktoś by się to przydarzyło, będzie to pierwsza linia obrony systemu przed niewłaściwie skonstrowanym kodem.

## Kompilacja

W językach kompilowanych w momencie kompilacji mogą pojawiać się błędy. Błędy te muszą być sygnalizowane, gdy narusza się składnię [36]. Poza faktyczną obsługą błędów kompilator może dostarczać dodatkowych informacji w postaci ostrzeżeń jako wskazówek dotyczących potencjalnych błędów.

Błędy kompilacji mogą mieć charakter leksykalny, składniowy lub semantyczny. Błędy wykonania programu są całkiem trudne do wychwycenia w samym procesie składania programu, tj. analizy struktury, składni i generowania kodu wynikowego.

## Budowanie

Tak jak składnia i kompilacja to oczywiste elementy weryfikacji programu między innymi pod kątem jego podstawowych cech jakościowych, tak procesy budowania i łączenia elementów programu to znacznie bardziej rozległy temat. Jakością w tym obszarze może być przede wszystkim używanie właściwych, tj. aktualnych i stabilnych, bibliotek, które dołączane są do konstruowanego przez nas programu. Załączanie zbyt wielu bibliotek, w tym takich, które nie są finalnie wykorzystywane, to marnowanie przestrzeni na dysku i konieczność ładowania takiego kodu do pamięci operacyjnej — ma to zatem wpływ na wydajność i pojemność systemu.

Automatyzacja procesu budowania to rozwiązania takie jak `npm`, `bundler`, `maven`, `sbt`. W językach takich jak `C++` mamy do czynienia z łączeniem bibliotek w sposób dynamiczny lub statyczny, z generowaniem albo zmniejszaniem podatnej na zmiany środowiska wersji programu lub większej, ale gwarantującej używanie dokładnie tych wersji bibliotek, jakie chcemy, aby były używane.

## Scalanie kodu

W nawiązaniu do tematu ilości pracy w projekcie i organizacji pojęcie scalania kodu (ang. *merge request*) ma istotne znaczenie dla jakości oprogramowania. Jeśli w zespole są wskazane osoby odpowiedzialne za wykonywanie włączania gałęzi kodu do gałęzi wynikowych i wdrożeniowych, należy monitorować postępy takich prac. Ich wydajność i efekty mają bezpośredni wpływ na jakość. Jeżeli osoby łączące kod mają za dużo tych zadań przydzielonych, wówczas nie będą tego kodu czytały dokładnie, bo nie będzie na to czasu. Wskazuje to na niedostateczny rozmiar zespołu względem oczekiwań przed nim stawianych.

Od strony technicznej właściwe scalenie kodu, bez względu, z jakim językiem programowania i platformami mamy do czynienia, to *de facto* tworzenie nowego kodu. Podstawową zasadą jest, że złączone gałęzie tworzą nowy byt, który od nowa wymaga przetestowania. Samo przetestowanie elementów składowych jest niewystarczające. Należy ten proces rozpocząć zaraz po scaleniu kodu i wdrożeniu go do odpowiedniego środowiska.

Oczywistymi problemami jakościowymi związanymi ze scalaniem kodem są te sytuacje, w których jest to wykonane w sposób niezgodny z oczekiwaniami autorów zmian. Wybranie niewłaściwych wersji w przypadku konfliktu w znaczącej liczbie sytuacji powoduje powstanie kodu, który działa w sposób mało przewidywalny. Można wręcz przyjąć za zasadę, że w przypadku występowania konfliktów, nawet z pozoru mało ważnych i prostych do rozwiązania, można takie MR/PR odrzucać.

Fakt odrzucenia żądania scalenia powinien zdarzać się częściej, niż w praktyce możemy to spotkać. Lepiej ograniczyć ryzyko i oczekiwać uwzględnienia wszystkich uwag i niejasności, niż zakładać cokolwiek. Szczególnym przypadkiem będzie scalanie kodu wielu gałęzi, które powstawały na przestrzeni więcej niż kilku dni. Im więcej czasu mija, im częściej na kodzie wykonywany jest *rebase*, tym w praktyce większe prawdopodobieństwo, że gdzieś po drodze wykonano to niepoprawnie.

Klasyczną sytuacją zaburzenia historii (dla Gita) będzie właśnie wykonywanie *rebase*, *squash* na kodzie, który lokalnie jest nieaktualny. Jeżeli ktoś będzie w stanie publikować takie zmiany z opcją *force* na gałęziach wynikowych lub wdrożeniowych, wówczas prawie na pewno wystąpią problemy. Wynikają one wyłącznie z niewiedzy, a nie z problemów z systemem wersjonowania.

## Automatyczne testy programistyczne

U podstaw działania większości programów handlowych leży warstwa danych. Testujemy zatem kod i model danych, jednostkowo natomiast algorytmy funkcjonalnie pod kątem ich wydajności. Nie musimy się obawiać zbyt dużego uproszczenia takich testów, służą one właśnie zapewnieniu podstaw do działania reszty systemu i nie muszą weryfikować absolutnie wszystkiego.

Przykłady prezentowane poniżej pochodzą z języków Ruby, Lua oraz JavaScript.

### minitest

Na potrzeby prezentacji konkretnych przykładów automatycznych testów programistycznych weźmiemy na warsztat bibliotekę *minitest* autorstwa Ryana Davisa i Erica Hodela. Biblioteka ta jest przeznaczona do języka Ruby:

Składniki biblioteki *minitest* są następujące:

- *minitest/autorun*: uruchamianie zbiorcze testów,
- *minitest/test*: właściwy system testowy,
- *minitest/spec*: testy typu *Spec*,
- *minitest/benchmark*: weryfikacja czasowa testu/kodu,
- *minitest/mock*: mechanizmy do zaślepek (*mock/stub*).

Dalsze przykłady pochodzą z oficjalnej dokumentacji biblioteki, powinny być zatem reprezentatywne dla większości przypadków i zgodne z wizją użytkownika biblioteki przez jej autorów.

Za bazy kod podajemy następującą klasę (plik *02/base.rb*):

```
class Meme
  def i_can_has_cheezburger?
    "OHAI!"
  end
  def will_it_blend?
    "YES!"
  end
end
```

## Test jednostkowy

Plik *02/minitest/unit.rb*:

```
require "minitest/autorun"
load "base.rb"
class TestMeme < Minitest::Test
  def setup
    @meme = Meme.new
  end
  def test_that_kitty_can_eat
    assert_equal "OHAI!", @meme.i_can_has_cheezburger?
  end
  def test_that_it_will_not_blend
    refute_match /^no/i, @meme.will_it_blend?
  end
  def test_that_will_be_skipped
    skip "test this later"
  end
end
```

## Test funkcjonalny

Plik *02/minitest/spec.rb*:

```
require "minitest/autorun"
load "base.rb"
describe Meme do
  before do
    @meme = Meme.new
  end
  describe "when asked about cheeseburgers" do
    it "must respond positively" do
      _(@meme.i_can_has_cheezburger?).must_equal "OHAI!"
    end
  end
  describe "when asked about blending possibilities" do
    it "won't say no" do
      _(@meme.will_it_blend?).wont_match /^no/i
    end
  end
end
```

## Test wydajnościowy

Plik `02/minitest/benchmark1.rb`:

```
require "minitest/autorun"
require "minitest/benchmark"
def my_algorithm(n)
  puts n
  n
end
describe "Sample Benchmark" do
  bench_performance_constant "my_algorithm", 0.9999 do |n|
    5.times do
      my_algorithm(n)
    end
  end
end
end
```

Mamy do dyspozycji weryfikację wydajności pod kątem wydajności/złożoności algorytmu: stałą, wykładniczą, liniową, logarytmiczną czy potęgową [50]. Testy uruchamiamy za pomocą polecenia:

```
ruby -Ilib:test benchmark1.rb --verbose
```

Wynik działania pokazany jest na rysunku 2.1.

```
sobczam@sobczam-ThinkPad-T420:~/Documents/01_prywatne/q/src/q/02/minitest$ ruby -Ilib:test benchmark1.rb --verbose
Run options: --verbose --seed 12340

# Running:

bench_my_algorithm1
1
1
1
1
1
1
0.00017510
10
10
10
10
0.000101100
100
100
100
100
0.0000671000
1000
1000
1000
1000
0.00006710000
10000
10000
10000
10000
0.000094
Memo Benchmark#bench_my_algorithm = 0.03 s = .
Finished in 0.029833s, 33.5196 runs/s, 33.5196 assertions/s.
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

**RYSUNEK 2.1.** Uruchomienie testu wydajnościowego

## busted

Innym ciekawym przykładem, który warto przytoczyć, jest język Lua i biblioteka busted [51]. Język ten jest jednym z liderów skryptowych rozwiązań wbudowanych w różnego rodzaju urządzenia.

Interpreter i narzędzia instalujemy w następujący sposób:

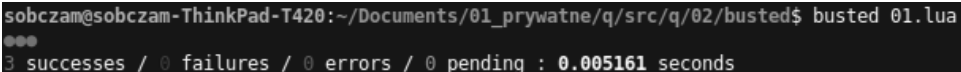
```
sudo apt-get install lua5.3
sudo apt-get install lua-busted
sudo ln -s /usr/bin/lua5.3 /usr/bin/lua
```

## Test funkcjonalny

Przykładowy test wygląda jak poniżej (plik *02/busted/01.lua*):

```
describe("Busted unit testing framework", function()
  describe("should be awesome", function()
    it("should be easy to use", function()
      assert.truthy("Yup.")
    end)
    it("should have lots of features", function()
      -- deep check comparisons!
      assert.are.same({ table = "great"}, { table = "great" })
      -- or check by reference!
      assert.are_not.equal({ table = "great"}, { table = "great"})
      assert.truthy("this is a string") -- truthy: not false or nil
      assert.True(1 == 1)
      assert.is_true(1 == 1)
      assert.falsy(nil)
      assert.has_error(function() error("Wat") end, "Wat")
    end)
    it("should provide some shortcuts to common functions", function()
      assert.are.unique({{ thing = 1 }, { thing = 2 }, { thing = 3 }})
    end)
  end)
end)
```

Wynik uruchomienia (*busted 01.lua*) przedstawia rysunek 2.2.



```
sobczam@sobczam-ThinkPad-T420:~/Documents/01_prywatne/q/src/q/02/busted$ busted 01.lua
...
3 successes / 0 failures / 0 errors / 0 pending : 0.005161 seconds
```

RYSUNEK 2.2. Wynik uruchomienia testu Lua/Busted

## Minimalna postać testu

W mojej opinii jedną z ważniejszych kwestii związanych z wdrażaniem programistycznych testów automatycznych jest prostota ich budowy i praktycznego zastosowania. Im bardziej narzędzia są skomplikowane, tym większa jest bariera wejścia i tym bardziej ludzie są wobec tego sceptyczni. W przypadku biblioteki busted mamy możliwość osadzenia całego testu i wywołania go w jednym pliku źródłowym (plik *02/busted/02.lua*):



```
require 'busted.runner'()
describe("a test", function()
  it("should be easy to use", function()
    assert.truthy("Yup.")
  end)
end)
```

Uruchamiamy wtedy taki test za pomocą polecenia `lua 02.lua`.

## Zaśleпки

### Spy

Biblioteka `busted` oferuje bardzo ciekawy mechanizm o nazwie *Spy*, który „podpina” się pod wywołania faktycznych funkcji, śledząc ich wykonania. Umożliwia to różnego rodzaju weryfikacje działania programu i testu (plik `02/busted/spy.lua`):

```
require 'busted.runner'()
describe("spies", function()
  it("registers a new spy as a callback", function()
    print "\n-> new spy"
    local s = spy.new(function(a,b,c)
      print(a,b,c)
    end)
    s(1, 2, 3)
    s(4, 5, 6)
    assert.spy(s).was.called()
    assert.spy(s).was.called(2) -- twice!
    assert.spy(s).was.called_with(1, 2, 3) -- checks the history
  end)
  it("replaces an original function", function()
    print "\n-> replace"
    local t = {
      greet = function(msg) print(msg) end
    }
    local s = spy.on(t, "greet")
    t.greet("Hey!") -- prints 'Hey!'
    assert.spy(t.greet).was_called_with("Hey!")
    t.greet:clear() -- clears the call history
    assert.spy(s).was_not_called_with("Hey!")
    t.greet:revert() -- reverts the stub
    t.greet("Hello!") -- prints 'Hello!', will not pass through the spy
    assert.spy(s).was_not_called_with("Hello!")
  end)
end)
```

### Stub

*Stub* działa podobnie jak *Spy* z tą różnicą, że nie wywołuje on zamienianej oryginalnej funkcji. Mamy po prostu możliwość „przesłonięcia” pewnych funkcji, co jest przydatne podczas testowania warstwy danych (plik `02/busted/stub.lua`):

```
require 'busted.runner'()
describe("stubs", function()
  it("replaces an original function", function()
    local t = {
```

```

    greet = function(msg) print(msg) end
  }
  stub(t, "greet")
  t.greet("Hey!") -- DOES NOT print 'Hey!'
  assert.stub(t.greet).was.called_with("Hey!")
  t.greet:revert() -- reverts the stub
  t.greet("Hey!") -- DOES print 'Hey!'
end)
end)

```

## Pozostałe przykłady testów

### Integracyjne

Testy tego rodzaju weryfikują kilka warstw abstrakcji, np. łączą model danych i kontrolery w konwencji MVC. Może to być jednak integracja nie na poziomie technologicznym, ale społecznym, jeśli integrujemy warstwę biznesu z warstwą produkcji oprogramowania. Właściwym wtedy określeniem dla tego rodzaju testów są testy behawioralne, weryfikujące zachowanie aplikacji i jej interakcję z elementami bezpośredniego otoczenia.

Przykładem tutaj wybranym jest biblioteka Cucumber. Jest ona dostępna w wielu językach, m.in. Java, Ruby czy JavaScript. Ten ostatni właśnie z racji rosnącej popularności nawet w zakresie programowania serwerowego zasługuje na więcej uwagi [52].

Plik *02/cuk/cucumber.js*:

```

module.exports = {
  default: '--format-options '{"snippetInterface": "synchronous"}'
}

```

Plik *02/cuk/features/step\_definitions/stepdefs.js*:

```

const assert = require('assert');
const { Given, When, Then } = require('cucumber');

```

Plik *02/cuk/package.js*:

```

{
  "name": "cuk",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "cucumber-js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "cucumber": "^6.0.3"
  }
}

```

Wynik uruchomienia takiego pustego szablonu (`npm test`) pokazany jest na rysunku 2.3.

```
sobczam@sobczam-ThinkPad-T420:~/Documents/01_prywatne/q/src/q/02/cuk$ npm test
> cuk@1.0.0 test /home/sobczam/Documents/01_prywatne/q/src/q/02/cuk
> cucumber-js

0 scenarios
0 steps
0m00.000s
```

**RYСУNEK 2.3.** Uruchomienie pustego szablonu Cucumber w JavaScriptcie

Taka forma testów po czasie może być znacząca jako rzeczywista, zgodna z kodem, dokumentacja techniczna programu:

*When we do Behaviour-Driven Development with Cucumber we use concrete examples to specify what we want the software to do. Scenarios are written before production code. They start their life as an executable specification. As the production code emerges, scenarios take on a role as living documentation and automated tests [52].*

Poza podejściem BDD i implementacją testów integracyjnych mamy możliwość (i konieczność zarazem) zastosowania podejścia TDD, czyli zaczynać od testów, a kończyć na kodzie. Ta kolejność ugruntuje nas w przekonaniu, że na pewno wiemy, co chcemy zrobić, i że zostało to odpowiednio przemyślane i zaplanowane.

Plik `02/cuk/features/isitfriday_yet.feature`:

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario: Sunday isn't Friday
    Given today is Sunday
      When I ask whether it's Friday yet
        Then I should be told "Nope"
```

Nazwa naszego *Feature* powinna odzwierciedlać nazwę pliku, będzie to dobrą konwencją. Druga linijka nie jest uruchamiana, jest to opis naszego zestawu testowego. Jako że praktykujemy tutaj podejście TDD, wynikiem uruchomienia takiego zestawu testowego będzie błąd widoczny na rysunku 2.4.

Kolejny kod dodany do pliku `02/cuk/features/step_definitions/stepdefs.js`:

```
Given('today is Sunday', function () {
  // Write code here that turns the phrase above into concrete actions
  return 'pending';
});
When('I ask whether it\'s Friday yet', function () {
  // Write code here that turns the phrase above into concrete actions
  return 'pending';
});
Then('I should be told {string}', function (string) {
  // Write code here that turns the phrase above into concrete actions
  return 'pending';
});
```

```

sobczam@sobczam-ThinkPad-T420:~/Documents/01_prywatne/q/src/q/02/cuk$ npm test
> cuk@1.0.0 test /home/sobczam/Documents/01_prywatne/q/src/q/02/cuk
> cucumber-js

UUU

Warnings:

1) Scenario: Sunday isn't Friday # features/is_it_friday_yet.feature:4
  ? Given today is Sunday
    Undefined. Implement with the following snippet:

    Given('today is Sunday', function () {
      // Write code here that turns the phrase above into concrete actions
      return 'pending';
    });

  ? When I ask whether it's Friday yet
    Undefined. Implement with the following snippet:

    When('I ask whether it\'s Friday yet', function () {
      // Write code here that turns the phrase above into concrete actions
      return 'pending';
    });

  ? Then I should be told "Nope"
    Undefined. Implement with the following snippet:

    Then('I should be told {string}', function (string) {
      // Write code here that turns the phrase above into concrete actions
      return 'pending';
    });

1 scenario (1 undefined)
3 steps (3 undefined)
0m00.000s
npm Test failed. See above for more details.

```

#### **RYSunek 2.4.** Pierwszy Feature, jeszcze bez implementacji właściwych funkcji

I kolejno:

```

function isItFriday(today) {
  // We'll leave the implementation blank for now
}

```

Następnie:

```

function isItFriday(today) {
  // We'll leave the implementation blank for now
}
Given('today is Sunday', function () {
  this.today = 'Sunday';
});
When('I ask whether it\'s Friday yet', function () {
  this.actualAnswer = isItFriday(this.today);
});
Then('I should be told {string}', function (expectedAnswer) {
  assert.equal(this.actualAnswer, expectedAnswer);
});

```

oraz:

```
function isItFriday(today) {
  return 'Nope';
}
```

Taka seria kolejnych elementów dodawanych zgodnie z postępowaniem prac, czyli określenie specyfikacji testu, przygotowanie podstawowych elementów mapujących, dodanie brakujących funkcji, uzupełnienie funkcji o zwracanie właściwych, oczekiwanych wartości, jest to właśnie BDD i TDD jednocześnie w praktyce.

## UI

Biblioteka Cucumber opisana na przykładzie języka JavaScript oraz metodyk TDD i BDD ma możliwość również wykonywania testów interfejsu użytkownika. Są to już końcowe testy, najbliższe rzeczywistego użytkownika. Wszelkie rozbieżności, które narosły od momentu przekazania wymagań, poprzez ich tłumaczenie, zapis oraz wykonanie, zostaną tutaj spotęgowane. Jeśli jednak zapewnimy testy tego rodzaju, będą one również elementem dokumentacji systemu, a zatem będzie możliwość weryfikacji założeń w postaci testów jednostkowych, funkcjonalnych zapisanych w tej samej postaci, z testami końcowymi, weryfikującymi sposób użytkownika systemu.

Akurat w przypadku użytkownika interfejsem jest przeglądarka internetowa. Mamy do dyspozycji szereg sterowników odpowiadających za współpracę z przeglądarkami. Jednymi z nich są *selenium-webdriver* oraz *chromedriver*. Należy ich obecność zapewnić poleceniem `npm`.

Przykładowy scenariusz (plik `02/cuk/features/findingsomecheese.feature`):

```
Feature: Finding Some Cheese
  Scenario: I want some cheese
    Given I am on the Wikipedia cheese article
    Then the page title should start with "Cheese"
```

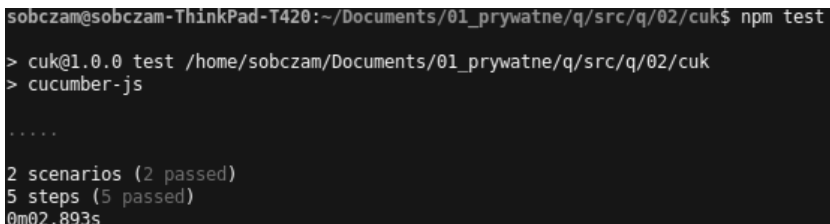
oraz definicja poszczególnych kroków (plik `02/cuk/features/step_definitions/stedefs.js`):

```
// UI
// driver setup
const capabilities = Capabilities.chrome();
capabilities.set('chromeOptions', { "w3c": false });
const driver = new Builder().withCapabilities(capabilities).build();
Given('I am on the Wikipedia cheese article', async function () {
  await driver.get('https://en.wikipedia.org/wiki/Cheese');
});
Then('the page title should start with {string}', {timeout: 60 * 1000}, async function
↳(searchTerm) {
  const title = await driver.getTitle();
  const isTitleStartsWithCheese = title.lastIndexOf(`${searchTerm}`, 0) === 0;
  expect(isTitleStartsWithCheese).to.equal(true);
});
AfterAll('end', async function(){
  await driver.quit();
});
```

i konfiguracja:

```
const assert = require('assert');
const { Given, When, Then, AfterAll } = require('cucumber');
const { Builder, By, Capabilities, Key } = require('selenium-webdriver');
const { expect } = require('chai');
require("chromedriver");
```

Wynik uruchomienia to otwarte okno przeglądarki, nawigacja do strony Wikipedii oraz weryfikacja treści strony. Podsumowanie takiego testu przedstawia rysunek 2.5.



```
sobczam@sobczam-ThinkPad-T420:~/Documents/01_prywatne/q/src/q/02/cuk$ npm test
> cuk@1.0.0 test /home/sobczam/Documents/01_prywatne/q/src/q/02/cuk
> cucumber-js
.....
2 scenarios (2 passed)
5 steps (5 passed)
0m02.893s
```

**RYСУNEK 2.5.** Wynik działania testu UI

Poza testami automatycznymi tego rodzaju jak powyżej mamy możliwość użycia narzędzi automatyzujących pracę poprzez wprowadzanie danych do formularzy, nawigowanie po kolejnych stronach — wszystko nagrywane z poziomu sesji użytkownika celem późniejszego odtwarzania tego w sposób zautomatyzowany.

## Specyfikacja, konfiguracja, środowisko

Testowanie kodu jest relatywnie prostą czynnością, łatwo bez wątpienia określić, jakie są jej ramy. W jaki sposób jednak możemy weryfikować poprawność na podstawie tego, jak kod powstaje, a później w jakim otoczeniu funkcjonuje? Sądzę, że aspekt techniczny, tj. konfiguracja i środowisko, może być walidowany za pomocą spójnego procesu wdrożeniowego, np. opartego na powtarzalności — Docker będzie tego dobrym przykładem. Oczywiście jesteśmy ograniczeni obecnością na zdalnych serwerach określonych pakietów, jeśli chcemy cokolwiek w naszej konfiguracji zmienić, acz jeśli takowych zmian nie wykonujemy, możemy nawet uzyskać powtarzalność takich wdrożeń na rok, dwa itd. Wszystko to dzięki zapamiętywaniu poszczególnych warstw, jakimi są kolejne grupy poleceń, podczas budowania obrazu na podstawie pliku *Dockerfile*.

Jeżeli procedura wdrożenia aplikacji obejmuje zawsze naniesienie odpowiednich ustawień konfiguracyjnych do kodu (parametryzacja), zainstalowanie pakietów, aktualizację systemu itd., wówczas w przypadku wprowadzenia niekompatybilnych zmian do kodu aplikacji momentalnie się o tym przekonamy podczas nieudanego procesu budowania wersji do wydania i wdrożenia. Jeśli konfiguracja bazuje na składniach Ruby, JSON, YAML, XML, możemy użyć odpowiednich narzędzi weryfikujących składnię takiej konfiguracji, co jest pierwszym, obowiązkowym krokiem podczas procedury wdrożeniowej.

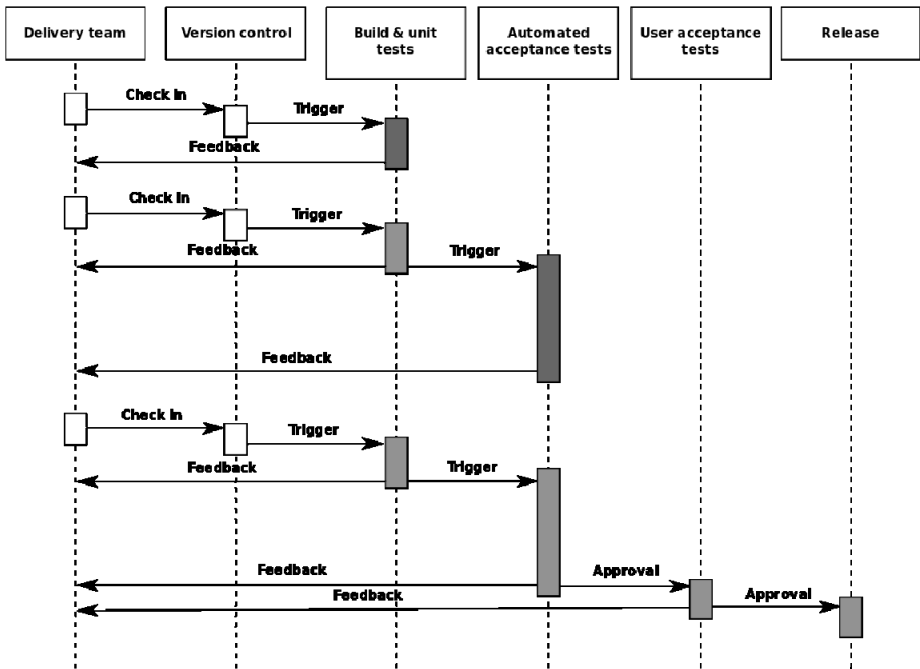
Jak widać, szanse na skuteczną weryfikację konfiguracji i środowiska uruchomieniowego są całkiem spore. Mamy przynajmniej ku temu narzędzia. Z weryfikacją specyfikacji sprawa

wygląda nieco gorzej. Przystudiowałeś kilka specjalistycznych pozycji dotyczących tego zagadnienia, muszę przyznać, że żadne z narzędzi mnie wystarczająco nie przekonało, że ma szansę być stosowane w praktyce. Mógłbym sam zaproponować, że stosując podejście BDD, czyli przygotowując scenariusze funkcjonalne przy współudziale właścicieli poszczególnych produktów w projekcie, oraz łącząc to z podejściem DDD, czyli zaangażowaniem ekspertów w danej domenie, będziemy w stanie dostarczyć właściwe procedury (wspólne projektowanie) i narzędzia (scenariusze funkcjonalne) dające zręby procesu weryfikacji specyfikacji w projekcie.

## CI/CD w praktyce

Jakość to między innymi porządek. Narzędziem wspierającym zapewnienie takiego porządku jest GitLab. Jest to system, który integruje repozytorium kodu, zgłoszenia, wiki oraz CI/CD. Bardzo ważne jest, aby w pracy projektowej podążać za określonym schematem, uzyskuje się wtedy bowiem standaryzację. Wszelkie działania i wiedza mogą być rejestrowane właśnie w GitLab. Czym różni się od innych rozwiązań? W podstawowej wersji zdecydowanie dorównuje takim narzędziom jak Jira (i całości tego ekosystemu) oraz nieco prostszym jak Mantis czy Redmine.

Dzięki stosowaniu GitLab jest możliwa łatwa realizacja pełnego procesu wytwarzania oprogramowania, począwszy od kodu, poprzez testy różnego rodzaju, kończąc na wydaniach (rysunek 2.6).

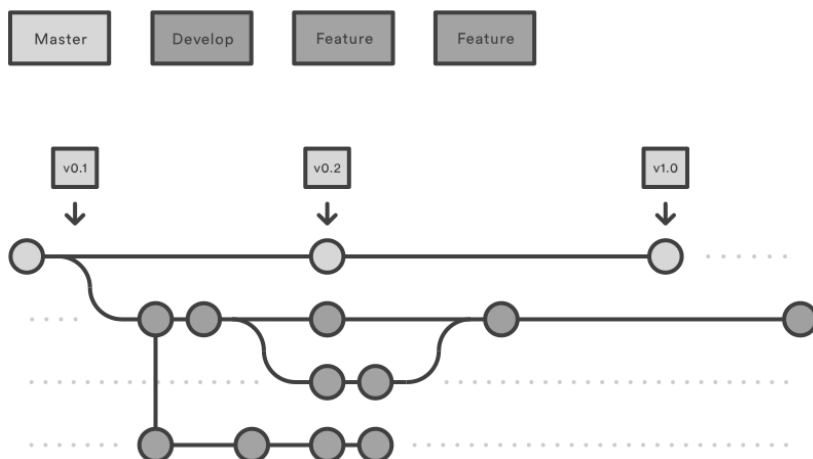


RYСУNEK 2.6. Od kodu do wydania kolejnej wersji [53]

**Dalsza lektura:** Poka Yoke.

## Git branching

Aby rozpocząć pracę z narzędziami CI/CD, należy wpierv opanować wersjonowanie kodu. Zapewni nam to porządek w ścieżce rozwoju naszych aplikacji oraz wprowadzi zasadę zarządzalności tym kodem. Mimo że brzmi to jak coś oczywistego, to praktyka pokazuje, że bardzo wiele zespołów nie korzysta lub korzysta w niewłaściwy sposób z dobrodziejstw takich narzędzi jak Git (lub innych systemów wersjonowania). Możemy zaproponować następujący schemat pracy z kodem (rysunek 2.7):



RYSUNEK 2.7. Wersjonowanie kodu [54]

Jest to jeden z możliwych przykładów. Sprowadza się do rozdzielenia kodu głównego (stabilnego) w gałęzi (ang. *branch*) master od gałęzi rozwojowych w develop i feature. Punkty w czasie wskazujące na kolejne usystematyzowane wydania aplikacji to tagi. Kod pomiędzy gałęziami powinien być scalany zgodnie z zasadą *code review* i szeregowany do dalszych scalań i wydań w czasie.

## .gitlab-ci.yml

Poniżej znajduje się przykładowy plik `.gitlab-ci.yml` konfiguracji wdrożeniowej aplikacji do zastosowania w ramach GitLab do wdrożenia w klastrze Nomad. Analogiczny plik można przygotować, aby wdrożyć aplikację w klastrze OKD:

```
image: docker:latest

services:
  - docker:dind

before_script:
  - docker info

build-master:
  stage: build
  script:
    - docker login -u admin -p password https://registry.home.lab:5000
    - docker build --pull -t "registry.home.lab:5000/lb:latest" .
    - docker push registry.home.lab:5000/lb:latest
  only:
    - master
```

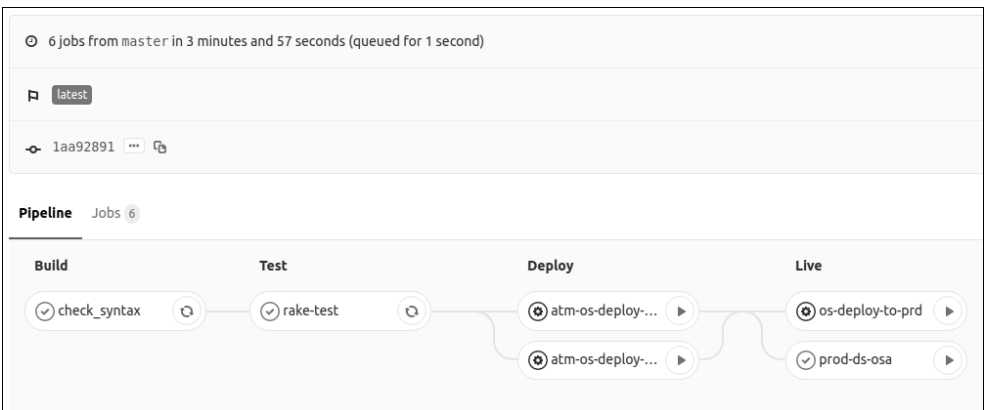


```

development:
  stage: deploy
  script:
    - export NOMAD_ADDR=http://172.28.128.11:4646
    - nomad job run .nomad
  only:
    - master

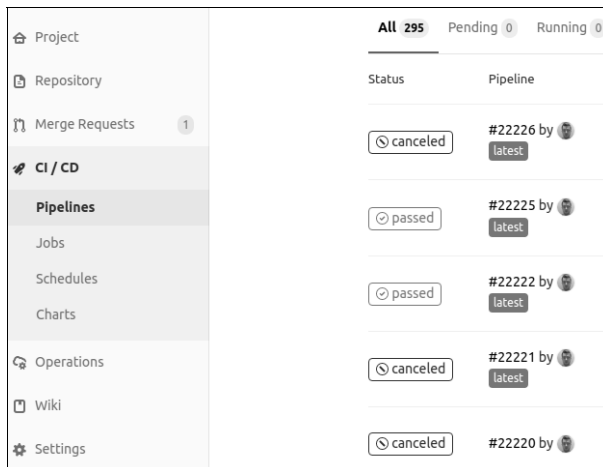
```

Forma ustrukturyzowana i systematyczna jest bardzo istotna do zachowania kontroli w projekcie, w którym występuje duża liczba aplikacji. Bez tej formy wdrożenia odbywałyby się chaotycznie i bez kontroli. Połączenie tego rodzaju konfiguracji wdrożeniowej i odpowiedni *Git branching* to podstawowe kroki, aby podnieść jakość procesu jako całości. Można te kroki wdrożeniowe rozszerzać o kroki związane z wykonywaniem testów (rysunek 2.8) czy sprawdzaniem poprawności składni aplikacji, np. analizą statyczną czy analizą pokrycia kodu testami.



**RYСУNEK 2.8.** Widok na pojedynczy proces wdrożeniowy

Mamy możliwość śledzenia bieżących wdrożeń (rysunek 2.9), jak również ustalania harmonogramów uruchamianych na podstawie określonych reguł.



**RYСУNEK 2.9.** Widok na listę uruchomień procesów wdrożeniowych



# Skorowidz

## A

agregacja danych, 149  
aksjomaty  
  programistyczne, 78  
  testowania, 78  
analiza składniowa, 195  
API, 185  
aplikacje  
  mikrouslugowe, 75  
  monolityczne, 74  
  porządkowanie struktury, 143  
APM, Application Performance Monitoring, 110  
architektura, 66  
  systemowa, 172  
artefakty, 93  
aspekty projektowe, 197  
auto DB reconnect, 180  
automatyczne testy, 95  
awarie, 15, 26, 124

## B

bazy danych, 114  
  aplikacje, 146  
  auto DB reconnect, 180  
  DB Deadlock, 181  
  monitoring, 147  
  projektowanie, 155  
  racjonalizacja serwera, 145  
  relacyjne, 195  
  replikacja, 146  
  zarządzanie, 175  
BDD, Behavior Driven Development, 73  
beneficjent, 133  
biblioteka

  busted, 98  
  Cucumber, 103  
  minitest, 95  
blokady transakcji, 181  
bloki wznawiania, 62  
błędotwórstwo, 79  
błędy, 15, 18, 25, 66, 117, 125  
  dane, 128  
  kategoryzacja, 127  
  kompozycja programu, 127  
  koszty, 166  
  nieuchronność, 189  
  operacje arytmetyczne, 128  
  pętle, 128  
  początkujących, 126  
  podprogramy, 128  
  powtarzalne, 27  
  tablice, 128  
  typowe, 125  
  ulotne, 27  
  z przemęczenia, 159  
  zapobieganie, 70  
  zestawienie, 125  
  zmiennie, 128  
budowanie  
  artefaktów, 93  
  elementów programu, 94  
busted, 98

## C

cargo cult, 152  
CI/CD, 93, 105  
Cucumber, 103  
czas odpowiedzi aplikacji, 113  
czytelność kodu, 19, 162

**D**

dane  
  w relacyjnej bazie, 195  
  zewnętrzne, 154

DB Deadlock, 181

debuging, 70

defekty, 24  
  w grach, 84  
  zapobieganie, 28

dobre praktyki, 18

Dockerfile, 138

dowodzenie poprawności programów, 79

DRY, 79

dublowanie procesów, 61

dyspozycyjność systemu, 80

działania doraźne, 189

**E**

edytor tekstowy, 171

Elasticsearch, 113

entropia oprogramowania, 79

ESB non-block, 180

estymaty, 185

**G**

Git branching, 106

GitLab, 105, 121

gromadzenie informacji, 197

gry  
  elementy otoczenia, 89, 90  
  mesh objects, 91  
  problemy z teksturami, 87  
  room bounds, 84  
  system optymalizacji, 84  
  wyświetlanie tekstur, 85

**H**

harmonogram zadań, 138

hotfixing, 124

**I**

IDE, 171

inspekcja oprogramowania krytycznego, 63

integracja, 185

**J**

jakość, 14, 18  
  danych, 29  
  wzorców projektowych, 31

język programowania, 71  
  jednolitość, 71  
  prostota, 71

**K**

klucze, 148  
  obce, 155

kolejkowanie  
  AMQP, 181  
  RabbitMQ, 180

kompatybilność API, 185

kompilacja, 94

kompromisy projektowe, 19, 69

komunikacja z użytkownikiem, 68

konfiguracja, 59

konteneryzacja, 183

konwencja, 20, 59

korekty danych, 156

**L**

liczba  
  wierszy programu, 65  
  zgłoszeń, 183

logi, 122

**M**

macierz pokrycia zmianami, 170

maksymy programistyczne, 71

martwy kod, 140

mechanizm  
  Spy, 99  
  Stub, 99

mesh objects, 91  
 metodologia programowania, 82  
 metodyka prewencji, 193  
 metodyki, 16  
 mikrorestarty, 62  
 mikrozarządzanie, 135  
 minitest, 95  
 model procesu tłumaczenia, 67  
 monitoring, 109, 131  
 monkey-patching, 124

## N

nadmiarowe przypisania, 149  
 narzędzie
 

- Elasticsearch, 113
- Git, 106
- GitLab, 105
- NewRelic, 110
- Sentry, 117

 nemawashi, 160  
 NewRelic, 110  
 niedoskonałość, 134  
 nieprawidłowe rekordy, 148  
 niezawodność, 27, 59  
 NOT NULL, 195  
 NULL, 157, 195

## O

obsługa
 

- awarii, 124
- błędów, 69

 ochrona bazy danych, 81  
 odbiorca, 133  
 odmładzanie, 62  
 odporność na błędy, 60  
 operator, 134  
 opis programu, 67  
 oprogramowanie
 

- kolejność celów, 83
- krytyczne, 63
- poprawność programów, 82
- samodostosowujące się, 187
- starzenie, 84
- wykonalność, 83

 optymalizacje, 16, 84  
 organizacja kodu, 21, 28

## P

pair programming, 184  
 paradoks hazardzisty, 162  
 pilność, 135  
 planowanie, 131, 170  
 plik .gitlab-ci.yml, 106  
 pliki .nif, 91  
 zdarzenia, 117  
 podsumowanie pracy aplikacji, 112  
 poprawianie stanu, 62  
 poziomy jakości, 16  
 predyspozycje osobowe, 179  
 prewencja, 193  
 problemy
 

- minimalizowanie, 189
- systemu optymalizacji, 84
- z teksturami, 87

 procedury CI/CD, 14  
 produkcja, 109  
 programowanie, 69
 

- czytelność kodu, 162
- defensywne, 83
- dobrze praktyki, 165
- metodologia, 82
- N-wersji, 61
- w parach, 184

 projektowanie, 16, 64, 66  
 prokrastynacja, 153  
 prostota i minimalizm, 65  
 proxy, 169  
 przechwytywanie wyjątków, 117, 197  
 przegląd
 

- jako proces, 63
- konwencjonalny, 62

 przepustowość aplikacji, 114  
 przestoje, 81  
 przewidywanie zmian, 80  
 punkty
 

- funkcyjne, 65
- kontrolne, 62
- obiektywne, 65

## R

racjonalizacja operacji, 137  
 redundancja
 

- dynamiczna, 60
- statyczna, 60

replikacja czasu rzeczywistego, 146  
 retro computing, 187  
 room bounds, 84  
 rozwiązywanie problemów, 68

## S

scalanie kodu, 94  
 Sentry, 117
 

- integracja z GitLab, 121
- przechwytywanie wyjątków, 117

 Service Discovery, 198  
 serwer bazy danych, 145  
 Simplex, 173  
 skutki, 20  
 specyfikacja zewnętrzna, 67  
 spójność, 20
 

- środowisk, 157

 Spy, 99  
 SQL
 

- dzielenie zapytań, 150

 standardy, 166  
 standaryzacja kodu, 28  
 statyczne typowanie, 193  
 struktura aplikacji, 143  
 Stub, 99  
 systemy
 

- dualne, 177
- gromadzenia informacji, 197
- nadrzędne, 175
- o wspólnej pamięci zbiorów, 176
- proste, 173, 175
- proste z wysuniętą transmisją, 173
- wieloprocessorowe, 177
- transakcyjne, 184
- wsadowe, 172, 184
- zbierania danych, 173
- zdwojone, 177

## T

TDD, Test Driven Development, 72  
 tekstury, 85, 87, 88  
 test
 

- akceptacyjny, 75
- automatyczny, 95
- funkcjonalny, 96, 98
- integracyjny, 75, 100

inwazyjny, 151  
 jednostkowy, 96  
 wydajnościowy, 97  
 testowanie, 15, 60
 

- aksjomaty, 78
- BDD, 73
- cel, 76
- dane testowe, 77
- filozofie, 77
- funkcji zewnętrznych, 75
- kierunki, 77
- modułu, 75
- specyfikacji, 76
- styl, 76
- systemu, 75
- TDD, 72
- TIP, 74
- UI, 104
- zasady, 75

 timeout, 144  
 TIP, Test In Production, 74  
 topologia baz danych, 116  
 typy danych, 195
 

- daty, 196
- indeksy, 197
- klucze, 148, 197
- liczby całkowite, 195
- precyzja, 196
- wartości ujemne, 195
- znakowe, 196

## U

UI, 103  
 upraszczanie systemów, 190  
 uruchamianie zadania, 138  
 uszkodzenia, 81
 

- główne przyczyny, 81

 użytkownik końcowy, 134

## W

walidacja, 60  
 watchdog, 62  
 ważność, 135  
 wersjonowanie kodu, 106  
 weryfikacja, 59
 

- obustronna, 154

współdzielenie kodu, 141  
wybór architektury, DDD, 64  
wyjątki  
  przechwytywanie, 117, 197  
wykonawca, 134  
wymagania użytkownika, 67  
wzorce projektowe, 30  
  fabryki, 34  
  jakość, 31  
  konstruktora, 35  
  korzyści, 32  
  kreatywne, 33  
  kryteria, 31  
  prototypowe, 36  
  rodzaje, 33  
  strukturalne, 39  
  trudności, 33  
  uzasadnienie, 31  
wzorzec projektowy  
  Adapter, 39  
  Dekorator, 42  
  Fasada, 45  
  Gość, 58  
  Iterator, 49  
  Kompozyt, 41  
  Łańcuch, 46  
  Mediator, 50  
  Memento, 51  
  Mostek, 40  
  Obserwator, 53  
  Polecenia, 48  
  Proxy, 46  
  Pyłek, 43  
  Singleton, 37  
  Stan, 54  
  Strategia, 55  
  Szablon, 56

**Z**

zadania cykliczne, 136  
założenia programu, 19, 67  
zamawiający, 133  
zapobieganie  
  błędom, 70  
  prze stojom, 81  
zapytania SQL, 150  
zarządzanie bazą danych, 175  
zasada  
  DRY, 79  
  ograniczonego zaufania, 193  
  testowania, 75  
zaśleпки, 99  
zdarzenia  
  historia, 118  
  lista, 118  
  podsumowanie, 119  
zespół, 20  
złożoność, 65, 178  
  cykliczna, 65  
zmiana częstotliwości, 136  
zwinność, 190





# PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA  
**Helion**

## Wszystko, co musisz wiedzieć o jakości oprogramowania, w jednym miejscu!

- \_Poznaj rodzaje błędów
- \_Zapewnij wysoką jakość kodu
- \_Dowiedz się, jak należy testować
- \_Twórz niezawodne oprogramowanie

Czym jest jakość oprogramowania? Czy to wyłącznie miara technicznej doskonałości kodu? A może jakość to coś znacznie więcej? Czy niezawodne działanie i brak błędów świadczą o wysokiej jakości programu? Co charakteryzuje kod, który można uznać za dobry? Jak mierzy się jakość w projektach informatycznych? Co robić, aby zapewnić wysoką jakość tworzonego oprogramowania? Jakie techniki i rozwiązania mogą w tym pomóc?

Odpowiedzi na te i wiele innych pytań znajdziesz w książce *Jakość oprogramowania. Podręcznik dla profesjonalistów*. To pierwsze tego rodzaju wydawnictwo na polskim rynku szybko wprowadzi Cię w zyskującą coraz większe znaczenie tematykę jakości oprogramowania komputerowego. Dzięki lekturze dowiesz się nie tylko, jak uzyskać i utrzymać odpowiednią jakość w projekcie informatycznym oraz jakie narzędzia są w tym pomocne, lecz również co robić, aby ograniczyć szkody spowodowane przez ewentualne błędy.

- \_Zapewnianie jakości kodu źródłowego
- \_Sposoby testowania programów
- \_Architektura, wzorce projektowe, metodyki
- \_Ciągłe dostarczanie i ciągła integracja
- \_Monitoring produkcji i obsługa awarii
- \_Najczęściej popełniane błędy i typowe przypadki

**Lektura obowiązkowa dla każdego, kto jest zaangażowany w proces wytwarzania oprogramowania!**

	<i>Sprawdź nasze szkolenia!</i>	<b>KOD KORZYŚCI</b> <i>Sięgnij po więcej!</i>	
 <b>helion.pl</b>	 <b>SZKOLENIA</b> <b>AKADEMIA IT &amp; BUSINESS</b>	ISBN 978-83-283-6102-7	
 <b>HELION SA</b> ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	<b>HELIONSZKOLENIA.PL</b>	 9 788328 361027	
<b>INFORMATYKA W NAJLEPSZYM WYDANIU</b>			Cena: 49,00 zł