# Implementowanie czystej architektury w **Pythonie**

Sebastian **Buczyński**

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek 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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktorzy prowadzący: Grzegorz Krzystek, Szymon Szwajger
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Helion S.A.
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/imczar*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-8686-0

Copyright © Helion S.A. 2022

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

# SPIS TREŚCI


1. **WPROWADZENIE** **11**

1.1. Era narzędzi 11
1.2. Dla kogo jest ta książka? 13
1.3. Co znajdziesz w książce? 13

2. **PODSTAWY CZYSTEJ ARCHITEKTURY** **15**

2.1. Po co to wszystko? 15
2.2. System płytki kontra system głęboki 16
2.2.1. CRUD, czyli system płytki 16
2.2.2. System głęboki 17
2.3. Założenia czystej architektury 20
2.3.1. Niezależność od frameworków 20
2.3.2. Wysoka testowalność 20
2.3.3. Niezależność od API i interfejsu użytkownika 20
2.3.4. Niezależność od bazy danych 21
2.3.5. Niezależność od firm trzecich 21
2.3.6. Elastyczność 21
2.3.7. Rozszerzalność 21
2.4. Warstwy, czyli horyzontalna organizacja kodu 22
2.4.1. Świat zewnętrzny 22
2.4.2. Infrastruktura 23
2.4.3. Aplikacja 23
2.4.4. Domena 24
2.4.5. Zasada zależności 26
2.4.6. Granice 26
2.5. Podsumowanie 29




Kup ksi k Pole ksi k

3. WZORCOWA IMPLEMENTACJA 30
3.1. Oznajmienie 30
3.2. Przepływ sterowania w czystej architekturze 30
3.3. Wymagania biznesowe 31
3.4. Implementacja 33
3.4.1. Diagram sekwencji 33
3.4.2. Granica wejściowa (input boundary) 33
3.4.3. Granica wyjściowa (output boundary) 34
3.4.4. Prezenter (presenter) 34
3.4.5. Model widoku (view model) 36
3.4.6. Przypadek użycia (use case) 36
3.4.7. Interfejs dostępu do danych (data access interface) 38
3.4.8. Dostęp do danych (data access) 38
3.4.9. Encja oferty (bid) 38
3.4.10. Encja aukcji (auction) 39
3.5. Podsumowanie 40

4. MODYFIKACJE CZYSTEJ ARCHITEKTURY 41
4.1. Dylemat prezentera 41
4.2. Pozbywamy się granicy wejściowej 44
4.3. Alternatywne podejścia do projektowania przypadków użycia 45
4.3.1. Fasada 45
4.3.2. Mediator pomiędzy wejściowym DTO a przypadkiem użycia 46
4.4. Użycie modeli bazodanowych jako encji 48
4.5. Podsumowanie 55

5. WSTRZYKIWANIE ZALEŻNOŚCI 57
5.1. Czym są zależności? 57
5.2. Wszędobylskie abstrakcje i klasy 58
5.3. Abstrakcje w czystej architekturze 61
5.4. Odwrócenie sterowania a zależności 64
5.5. Kontener IoC kontra service locator 66
5.6. Wstrzykiwanie zależności kontra konfiguracja 68
5.7. Podsumowanie 68

6. CQRS 70
6.1. Wstęp 70
6.2. Co to ma wspólnego z czystą architekturą? 72
6.3. Osobny stos odczytu — dlaczego? 73
6.4. Osobny stos odczytu — jak? 74
6.4.1. Zapytanie jako DTO 75
6.4.2. Zapytania jako osobne klasy 76
6.4.3. Fasada modelu do odczytu 77



Kup książkę Poleć książkę

6.5. CQRS kontra REST API 78
6.6. CQRS kontra GraphQL 79
6.7. Podsumowanie 81

**7. OSTRE GRANICE 82**
7.1. Słowo o złożoności 82
7.2. Dwa światy 84
7.3. Granica pomiędzy warstwą aplikacji a światem zewnętrznym 85
7.4. Pisanie wejściowego DTO 85
7.5. Value objects 86
7.6. Podsumowanie 89

**8. STUDIUM PRZYPADKU — PLATFORMA AUKCYJNA 90**
8.1. Jak pracujemy? 90
8.2. Jak zacząć, czyli chodzący szkielet 91
8.3. Nasz chodzący szkielet 93
8.4. Przypadek użycia dla składania oferty na aukcji 94
8.4.1. Nazewnictwo 94
8.4.2. Argumenty 94
8.4.3. Wyjście 95
8.4.4. Testy 96
8.5. Encje aukcji i oferty 97
8.5.1. Nazewnictwo 97
8.5.2. Value objects jako identyfikatory 98
8.5.3. Implementacja 98
8.5.4. Testy jednostkowe 99
8.5.5. Implementacja — ciąg dalszy 100
8.6. Abstrakcyjne repozytorium 101
8.6.1. Nazewnictwo 102
8.6.2. Implementacja 102
8.7. Repozytorium 102
8.7.1. Nazewnictwo 102
8.7.2. Implementacja działająca w pamięci 103
8.7.3. Rozwijanie implementacji pod osłoną TDD 103
8.8. Kończymy przypadek użycia — składanie oferty 105
8.8.1. Wstrzykiwanie zależności 105
8.8.2. Sprawiamy, że pierwszy sensowny test przechodzi 105
8.8.3. Refaktoryzacja 106
8.9. Organizacja kodu 107
8.9.1. Jak można ułożyć kod w Pythonie? 107
8.9.2. Organizujemy kod projektu 108
8.9.3. Organizujemy kod warstwy infrastruktury 110
8.9.4. Łączymy wszystko razem w komponencie main 112
8.9.5. Wystawiamy API 116



Kup książkę Poleć książkę

8.10. Finalizujemy aukcję w kolejnym przypadku użycia 118
8.10.1. Zarys przypadku użycia i wejściowe DTO 119
8.10.2. Rozszerzamy encję, by spełnić nowe wymagania 120
8.10.3. Skoro encje nie powinny mieć żadnych zależności, to czy mogą pytać o czas? 122
8.10.4. Wprowadzamy port dla płatności 123
8.10.5. Implementujemy adapter 125
8.10.6. Obsługa błędów kontra zasada zależności 126
8.10.7. A co, gdybyśmy chcieli dodać zapamiętywanie karty płatniczej? 127
8.10.8. Jak żyć, gdy adapter rośnie? 129
8.10.9. Bramka płatności ma już SDK. Nie możemy go po prostu użyć? 131
8.11. Przypadek użycia — rozpoczynanie nowej aukcji 131
8.11.1. Skąd się biorą nowe aukcje? 131
8.11.2. Encja aukcji i jej opis w jednym obiekcie — za i przeciw 132
8.11.3. Wprowadzamy deskryptor 134
8.11.4. Repozytorium z interfejsem kolekcji 134
8.11.5. Które repozytorium wybrać? 137
8.12. Operacje odczytu danych 138
8.12.1. Podejście z przypadkami użycia 138
8.12.2. CQRS na ratunek 140
8.12.3. Zapytania jako klasa 140
8.12.4. Model do odczytu 142
8.12.5. Podsumowanie operacji odczytujących dane 144
8.13. Odwracamy kontrolę za pomocą zdarzeń 144
8.13.1. Przykład — wysyłka e-maili 144
8.13.2. Techniki odwracania kontroli 145
8.13.3. Implementacja zdarzeń 146
8.13.4. Skąd wziąć szynę zdarzeń? 147
8.13.5. Jak wydostać zdarzenia z encji? 148
8.13.6. Encja gromadzi zdarzenia, które potem publikuje repozytorium 148
8.13.7. Encja zwraca zdarzenia z metod, które zmieniają jej stan 149
8.13.8 Testowanie encji, które zwracają zdarzenia 151
8.13.9. Subskrybowanie się na zdarzenia 152
8.13.10. Zdarzenia kontra transakcje kontra efekty uboczne 153
8.13.11. Niezawodne publikowanie zdarzeń — outbox pattern 155
8.13.12. Wprowadzamy jednostkę pracy 156
8.13.13. Czas życia jednostki pracy 158
8.13.14. Relacja pomiędzy jednostką pracy a szyną zdarzeń 158



Kup ksi k Pole ksi k

8.14. Radzimy sobie z innymi przekrojowymi zagadnieniami 160
8.14.1. Konfiguracja 161
8.14.2. Walidacja 162
8.14.3. Synchronizacja 163
8.15. Podsumowanie 166

**9. MODULARNOŚĆ 167**

9.1. Ciężar sukcesu — rozrost i ciągłe zmiany 167
9.2. Komponenty i kohezja 167
9.3. Organizacja kodu według komponentu 168
9.4. Komponenty i swoboda architektoniczna 170
9.5. Komponenty kontra mikroserwisy 170
9.6. Komponenty a użytkownik 172
9.7. Komponenty a bounded context 173
9.8. Komponenty — implementacja 173
9.9. Zależności między komponentami 175
9.9.1. Oddzielne drogi 175
9.9.2. Bezpośrednia zależność — oba komponenty implementują czystą architekturę 176
9.9.3. Niebezpośrednia zależność — oba komponenty implementują czystą architekturę 177
9.9.4. Zależność, gdy jeden z komponentów nie implementuje czystej architektury 178
9.9.5. Odmiany integracji za pomocą zdarzeń 178
9.9.6. Zależności między komponentami — podsumowanie 179
9.10. Studium przypadku — platforma aukcyjna 179
9.10.1. Odkrywamy komponenty 179
9.10.2. Komponenty platformy aukcyjnej 180
9.10.3. Co komponent wystawia na zewnątrz? 181
9.10.4. Tam, gdzie wszystko składa się w całość — komponent main 185
9.10.5. Korzystamy z komponentu main do uruchomienia aplikacji 186
9.10.6. Jedna architektura dla wszystkich komponentów — czy to możliwe? 187
9.10.7. Zależności pomiędzy komponentami 189
9.10.8. Integrowanie komponentów za pomocą zdarzeń 191
9.10.9. Wewnętrzna obsługa zdarzeń w tym samym komponencie 197
9.10.10. Integracja różnych komponentów za pomocą zdarzeń — prosty przypadek 198
9.10.11. Integracja różnych komponentów za pomocą zdarzeń — złożony przypadek 201
9.10.12. Inne ciekawe zastosowania menadżera procesu 206
9.10.13. Menadżer procesu kontra wyścigi 207
9.11. Podsumowanie 210



Kup ksi k Pole ksi k

10. TESTOWANIE 212
10.1. Strategia testowania i odmiany funkcji 212
10.1.1. Piramida testów — mit czy jedyna słuszna droga? 213
10.1.2. Rodzaje testów 215
10.1.3. Jak przetestować przeglądarkę do bazy danych? 218
10.1.4. Jak przetestować proxy? 220
10.1.5. Jak przetestować system głęboki? 223
10.2. Odkrywamy testowanie jednostkowe na nowo 224
10.2.1. Ile musi wiedzieć test? 224
10.3. Testowanie stanu kontra testowanie interakcji 228
10.3.1. Rodzaje weryfikacji 228
10.3.2. Niebezpieczeństwa związane z inspekcją stanu 229
10.3.3. Niebezpieczeństwa związane ze sprawdzaniem interakcji 231
10.3.4. Stuby kontra mocki 232
10.3.5. Rodzaje obiektów dublerów 233
10.4. Testujemy cały komponent jednostkowo 234
10.4.1. Ustawiamy komponent w pożądanym stanie 235
10.4.2. Wywołujemy akcję na komponencie 237
10.4.3. Weryfikujemy rezultat akcji na poziomie komponentu 237
10.4.4. Radzimy sobie z zależnościami w postaci portów i repozytoriów 240
10.5. Podsumowanie 241

11. ZAKOŃCZENIE 243
11.1. Co dalej? 243

12. SUPLEMENT A: MIGRACJA Z PROJEKTU ODZIEDZICZONEGO 245
12.1. Czy powinno się to robić? 245
12.2. Jak to zrobić? 245
12.3. „Nie mogę przestać dostarczać nowych funkcji!” 247

13. SUPLEMENT B: WPROWADZENIE DO EVENT SOURCING 248
13.1. Co to jest event sourcing? 248
13.2. Agregat z event sourcing kontra agregat z domain-driven design 250
13.3. Prosty przykład agregatu 251
13.3.1. Zamówienie jako encja 251
13.3.2. Istotne zmiany zamówienia w formie zdarzeń 252
13.3.3. Uwaga na te zdarzenia! 253
13.3.4. Zamówienie jako agregat 254
13.3.5. Testowanie agregatów 257
13.4. Persystencja 257
13.4.1. Nowe zdarzenia są dołączane na koniec strumienia zdarzeń 257
13.4.2. Pobieranie strumienia zdarzeń 259



Kup ksi k Pole ksi k

13.4.3. Dopisywanie nowych zdarzeń do strumienia 260
13.4.4. Wybór bazy danych — podsumowanie wymagań 261
13.4.5. Przykładowa implementacja z użyciem PostgreSQL 261
13.4.6. Użycie event store 266
13.4.7. Co robić, gdy wykryjemy wyścig? 266
13.4.8. Użycie repozytorium do ukrycia event store 268
13.4.9. Migawki stanu agregatu 268
13.5. Projekcje 272
13.6. Event sourcing w aplikacji składającej się z komponentów 276
13.6.1. Event sourcing to szczegół implementacyjny komponentu 276
13.6.2. Stosuj zdarzenia domenowe na potrzeby integracji 277
13.7. Podsumowanie 277

**BIBLIOGRAFIA 279**

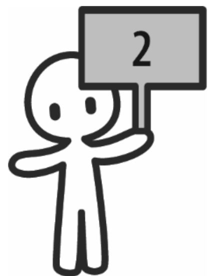


Kup ksi k

Pole ksi k

Kup ksi k

Pole ksi k




# PODSTAWY CZYSTEJ ARCHITEKTURY

## 2.1. Po co to wszystko?

Jedyną stałą rzeczą w życiu jest zmiana
— Heraklit

Zadaniem stawianym przed programistkami i programistami jest dostarczanie wartości poprzez pracę z kodem. Odbywa się ona poprzez pisanie kodu, rzadkie usuwanie i bardzo częste czytanie. Początki danego projektu zwykle są przyjemne i produktywne dzięki brakowi narzutu narosłego przez miesiące, a czasem lata — tak charakterystycznego w projektach odziedziczonych (ang. *legacy*).

Tymczasem to nie kto inny, jak właśnie osoby pracujące z kodem, napisał go tak, że ich następcy (a może nawet te same osoby?) mają problem, by połapać się w projekcie.

**UWAGA!**

Jeśli dodajemy kolejne funkcje do projektu i co gorsza, nie są one kompletnie niezależne od siebie, to całość zawsze robi się coraz bardziej skomplikowana.

Utrzymanie projektu w dobrym stanie przez długi czas i przewidywalne dostarczanie nowych funkcji wymaga wkładania świadomego wysiłku w postaci ciągłej refaktoryzacji. Inaczej będzie tylko gorzej. Chociaż sama refaktoryzacja jest zestawem prostych przekształceń, jak wydzielanie funkcji czy klas, to istotna jest postać kodu, do jakiej dążymy. Z tego punktu widzenia czysta



Kup ksi k Pole ksi k

architektura jest jak solidne rusztowanie, o które można oprzeć prostszy do zarządzania projekt.

Przede wszystkim jednak czysta architektura to inwestycja, która nie zawsze się opłaca w części lub całości.

## 2.2. System płytki kontra system głęboki

### 2.2.1. CRUD, czyli system płytki

Niektóre funkcje w aplikacjach są proste. Cała ich logika ogranicza się do pobrania danych prosto z bazy z małą lub żadną obróbką po drodze. Do tego będziemy mieć operacje dodawania nowych wpisów, modyfikacji istniejących lub usuwania. Prostym przykładem takiej funkcji jest zarządzanie swoim zestawem adresów na stronie sklepu internetowego. Adres możemy dodać, usunąć i czasem oznaczyć jako domyślny. Mamy do czynienia z kodem, który jest w zasadzie przeglądarką do bazy danych — operacje wstawiania (`INSERT`), usuwania (`DELETE`), aktualizacji (`UPDATE`) i wybierania (`SELECT`) są główną logiką.

Inną nazwą takich funkcji lub projektów jest CRUD — od słów: *create*, *read*, *update*, *delete*. Łatwo jest też ubrać te funkcje w RESTful API, bo każdą operację można jednoznacznie przyporządkować do metody HTTP. Do tego łatwo określić, na jakim zasobie dokonujemy tej operacji:

```
POST /addresses       # dodaj nowy adres
GET /addresses        # pobierz wszystkie adresy
GET /addresses/1      # pobierz adres o identyfikatorze 1
DELETE /addresses/1   # usuń adres o identyfikatorze 1
```

Podsumowując, funkcje typu CRUD będą proste w realizacji, a różnice między tym, co udostępnimy poprzez API na interfejsie użytkownika, a reprezentacją w bazie danych będą bardzo małe lub żadne. Jest to też przykład tak zwanego izomorficznego schematu danych, wspomnianego przy okazji omawiania wzorca *ActiveRecord* we wprowadzeniu.

W takiej sytuacji idealnie sprawdza się podejście z frameworka Django z rozszerzeniem Django REST framework. Najwięcej pracy mamy przy pisaniu modelu danych, a potem dużo mniejszym wysiłkiem udostępniamy operacje na nim poprzez REST API. Jest tak dlatego, że to ostatnie jest generowane właśnie z modelu:

```
class Address(models.Model):
    name = models.CharField(max_length=30)
    line_1 = models.CharField(max_length=100)
    line_2 = models.CharField(max_length=100)
```



Kup ksi k    Pole ksi k

```
    postal_code = models.CharField(max_length=24)
    city = models.CharField(max_length=30)
    country_code = models.CharField(max_length=2)
    state_or_province = models.CharField(max_length=30)
    is_default = models.BooleanField()
```

Następnie musimy zdefiniować, które pola będą widoczne na API i w jaki sposób wybierzemy, które adresy zwrócimy:

```
class AddressSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Address
        fields = "__all__"  # wystawmy wszystkie pola na API

class AddressesViewSet(viewsets.ModelViewSet):
    queryset = Address.objects.all()  # zwracajmy wszystkie adresy
    serializer_class = AddressSerializer
```

I to w zasadzie wszystko, by dostać gotowe do użycia API, które jest dokładnym odzwierciedleniem struktury danych w bazie.

Co ciekawe, z racji małej liczby przypadków testowych takie funkcje szybko i wygodnie można pokryć testami automatycznymi tylko na poziomie API czy UI. Implementacja tych testów też będzie prosta.

Dla czytelności przykładu pomijam aspekty takie jak wybieranie adresów tylko uwierzytelnionej osoby czy wymuszanie posiadania dokładnie jednego adresu domyślnego. Są one proste do zrealizowania i nie wnoszą nic do tego przykładu ani nie sprawiają trudności w implementacji przy użyciu Django REST framework.

W systemie płytkim czysta architektura będzie nietrafioną inwestycją i doprowadzi do nadmiernego skomplikowania (ang. *overengineering*).

## 2.2.2. System głęboki

Funkcje w produkcyjnych projektach, szczególnie części, które zarabiają pieniądze, nie będą aż tak trywialne. W wypadku systemów głębokich analiza powinna się skupiać na poszukiwaniu tak zwanych niezmienników (ang. *invariants*), czyli pewnych warunków, które zawsze muszą być spełnione. Jeżeli brzmi to tajemniczo, przywołajmy przykład wymuszania posiadania dokładnie jednego adresu domyślnego. Ten niezmiennik łatwo jest zrealizować na poziomie modelu, a więc i bazy danych — wystarczy ustawić odpowiedni indeks. W systemie głębokim z kolei spotkamy się z niezmiennikami, których na poziomie bazy danych wymusić się nie da albo będzie to na tyle trudne, że stanie się nieopłacalne.



Kup ksi k    Pole ksi k

Jako przykład weźmy aukcję. Na poziomie modelu aukcja będzie miała cenę otwarcia, cenę bieżącą, datę zakończenia i skojarzoną z nią listę ofert.

```
class Auction(models.Model):
    starting_price = MoneyField(
        max_digits=19,
        decimal_places=4,
        default_currency="USD",
    )
    current_price = MoneyField(
        max_digits=19,
        decimal_places=4,
        default_currency="USD",
    )
    ends_at = models.DateTimeField()
```

Oferta z kolei będzie się składać z identyfikatora licytującego i ceny powiązanej z ofertą:

```
class Bid(models.Model):
    price = MoneyField(
        max_digits=19,
        decimal_places=4,
        default_currency="USD",
    )
    bidder = models.ForeignKey(
        get_user_model(), on_delete=models.PROTECT
    )
    auction = models.ForeignKey(
        Auction,
        related_name="bids",
        on_delete=models.CASCADE,
    )
```

Zanim jednak pomyślimy, by po prostu udostępnić tworzenie ofert na API, zastanówmy się, czy są jakieś warunki do spełnienia. Po pierwsze, wymagajmy, by nowa oferta była koniecznie wyższa niż obecna cena aukcji. Nie akceptujemy w ogóle ofert niższych lub równych obecnej cenie — nic nie zapisujemy. Kolejna rzecz to składanie ofert na zakończonej już aukcji — na to też nie można pozwolić. Inna sprawa, nad którą trzeba się zastanowić: co w sytuacji, gdy ktoś składa ponownie ofertę na tę samą aukcję? Jeżeli przebija innego licytującego, to sprawa wydaje się prosta. A gdy przebija sam siebie?

To tylko pierwsza część analizy wymagań. Co się dzieje, gdy oferta zostanie już złożona? Naturalnie podwyższamy obecną cenę aukcji. A co, jeśli któraś oferta z jakiegoś powodu zostaje wycofana? To zależy, czy dana oferta jest zwycięska, czy nie. W tej drugiej sytuacji możemy po prostu usunąć wiersz z bazy i zapominamy o sprawie. W pierwszej — musimy zmodyfikować aukcję, bo obecna cena nie będzie odzwierciedlać rzeczywistego stanu.



Kup ksi k    Pole ksi k

Gdy aukcja się zakończy, co dalej? Wtedy powstaje obowiązek zapłaty (dla tego, kto zwyciężył w aukcji), który także może się zmieniać w czasie i zależnie od akcji zwycięzcy, ale samej aukcji już nie ruszamy. I tak dalej, i tak dalej. Co najważniejszego wynika z tej długiej analizy? To, że niezmienników jest całkiem sporo i do tego obejmują więcej niż jeden model. Innymi słowy, w wielu scenariuszach nie możemy po prostu modyfikować czy usunąć jednego modelu bez wpływu na pozostałe. Inaczej aplikacja będzie działać w sposób niepożądany i pojawią się błędy.

Duża liczba niezmienników to już mocna podpowiedź, że możemy mieć do czynienia z systemem głębokim. Samo pełne pokrycie przypadków testami automatycznymi to duży wysiłek. Na tyle duży, że dobrze byłoby móc testować same niezmienniki bez konieczności używania bazy danych, bo to spowalnia testy. Najlepiej też byłoby testować przypadki brzegowe jak najbliżej miejsca ich implementacji, a nie przez API. W drugim wypadku potrzeba o wiele więcej wysiłku, by ustawić system w odpowiednim stanie, a więc testy będą trudniejsze do pisania i utrzymywania.

Skoro mowa o testach, to jedną z największych zmór jest testowanie kodu, który korzysta z zewnętrznych usług, komunikując się z nimi przez sieć. Może to być na przykład bramka płatności lub nawet mikroserwis innego zespołu. Trudność polega na tym, że nie kontrolujemy tych zewnętrznych części, a więc niemożliwe jest na przykład ustawienie ich do pożądanego stanu w testach. Do tego czasem usługi zewnętrzne są niedostępne, sieć zawodna lub wdraża się ich nowe wersje z błędami. Mówiąc prościej, integracja z nimi jest konieczna dla realizacji projektu, ale jednocześnie stanowi źródło problemów, przed którymi trzeba się jakoś zabezpieczyć.

Istnieje jeszcze kilka innych sygnałów świadczących o tym, że mamy do czynienia z systemem głębokim. Na dalszych kartach tej książki poświęcimy im więcej miejsca. Są to między innymi:

- duże różnice pomiędzy strukturą API a modelem danych w bazie;
- różne typy użytkowników, które mogą modyfikować te same zasoby, ale z innymi uprawnieniami i dostępnymi operacjami;
- realny problem równoczesnej modyfikacji tego samego zasobu przez różnych aktorów (osoby lub systemy) i potrzeba jego ochrony;
- funkcje, które są realizowane w formie procesu rozciągającego się na wiele kroków i akcji żywej osoby.

Na system głęboki odpowiednim remedium będzie czysta architektura i inne techniki, których użycie ona umożliwia.



Kup ksi k Pole ksi k

## 2.3. Założenia czystej architektury

Najprościej mówiąc, w idei czystej architektury kładzie się nacisk na logikę biznesową. Niedopuszczalne jest, by framework, baza danych lub usługa firmy trzeciej przeciekały i wpływały na reguły biznesowe. Właściwie zastosowana czysta architektura ma szereg cech, które pozwalają oddzielić logikę biznesową od mniej istotnych aspektów projektu. Robimy to dlatego, że owa logika biznesowa jest dostatecznie skomplikowana sama w sobie, a mieszanie jej z innymi aspektami tylko utrudnia poruszanie się w niej.

### 2.3.1. Niezależność od frameworków

Jedną z pierwszych rzeczy, które nie powinny „śmiecić" w logice biznesowej, jest kod specyficzny dla frameworka. Umożliwienie łatwej zmiany frameworka nie jest celem, chociaż staje ona się dużo prostszym zadaniem, gdy jego kod nie wycieka do miejsc implementacji logiki biznesowej. W praktyce jest to uzasadnione tylko w rzadkich wypadkach. Celem jest swobodne rozwijanie trudnej i cennej części projektu bez dodatkowego narzutu.

### 2.3.2. Wysoka testowalność

Testowalnością nazywamy łatwość, z jaką możemy napisać i utrzymywać testy. Odpowiednio rozłożone granice w czystej architekturze oddzielają logikę od trudnych w testowaniu części systemu. Jako rezultat łatwo jest uzyskać wzorcową piramidę testów, w której szybkie, proste i lekkie w utrzymaniu testy jednostkowe stanowią zdecydowaną większość. Mniej liczne testy integracyjne i *end-to-end* dopełniają siatkę bezpieczeństwa, jaką stanowi zestaw testów. Więcej na temat dobierania strategii testowania w poświęconym jej rozdziale.

### 2.3.3. Niezależność od API i interfejsu użytkownika

W systemie płytkim struktura danych na API i w bazie danych niewiele się od siebie różni — często jest taka sama. W systemie głębokim różnica staje się wyraźna.

Pamiętajmy, że API jest swojego rodzaju interfejsem użytkownika i powstaje dla określonego klienta. Z tego powodu możemy dostosowywać API do wygody i potrzeb klientów, jeżeli jest to uzasadnione. Niemniej API musi



Kup książkę Poleć książkę

być podporządkowane logice aplikacji, która może być dowolnie skomplikowana. Nie w drugą stronę — API nie będzie dyktować sposobu działania logiki, chociaż może na przykład wpływać na dostępne modele danych tylko do odczytu. Więcej na ten temat w rozdziale o CQRS.

### 2.3.4. Niezależność od bazy danych

Chociaż do wyboru są różnorodne typy baz danych — relacyjne, dokumentowe, klucz – wartość i parę innych — jest to nadal pewien szczegół implementacyjny. Jedna z technik płynnie łączących się z czystą architekturą, to jest taktyczne *domain-driven design*, zakłada budowanie obiektów biznesowych przy skupieniu się wyłącznie na niezmiennikach.

Najważniejsze do zapamiętania jest to, że nie powinniśmy być ograniczeni w modelowaniu logiki biznesowej (szczególnie wymuszaniu niezmienników) poprzez sposób, w jaki dane są zapisywane w bazie danych.

### 2.3.5. Niezależność od firm trzecich

Bardzo możliwe, że aby rozwijać projekt, będziemy korzystać z usług innych firm. Typowe przykłady to bramka płatności czy system do marketingu. Każda kolejna integracja to nowe źródło ryzyka i niestabilności.

W idei czystej architektury postuluje się, aby oddzielić się od zewnętrznych dostawców i nie pozwolić, by informacje specyficzne dla nich przeciekały nam przez cały projekt. Do tego można zauważyć tendencję, by takie integracje wymieniać na inne, gdy projekt jest rozwijany przez kilka lat.

### 2.3.6. Elastyczność

Dzięki wprowadzaniu warstw abstrakcji możemy pewne decyzje odroczyć do momentu, aż będziemy mieć więcej informacji. Wtedy będzie można podjąć trafniejszą decyzję.

Moja osobista definicja architektury mówi, że jest to sztuka unikania decyzji, które potem przeszkadzają nam w rozwijaniu projektu.

### 2.3.7. Rozszerzalność

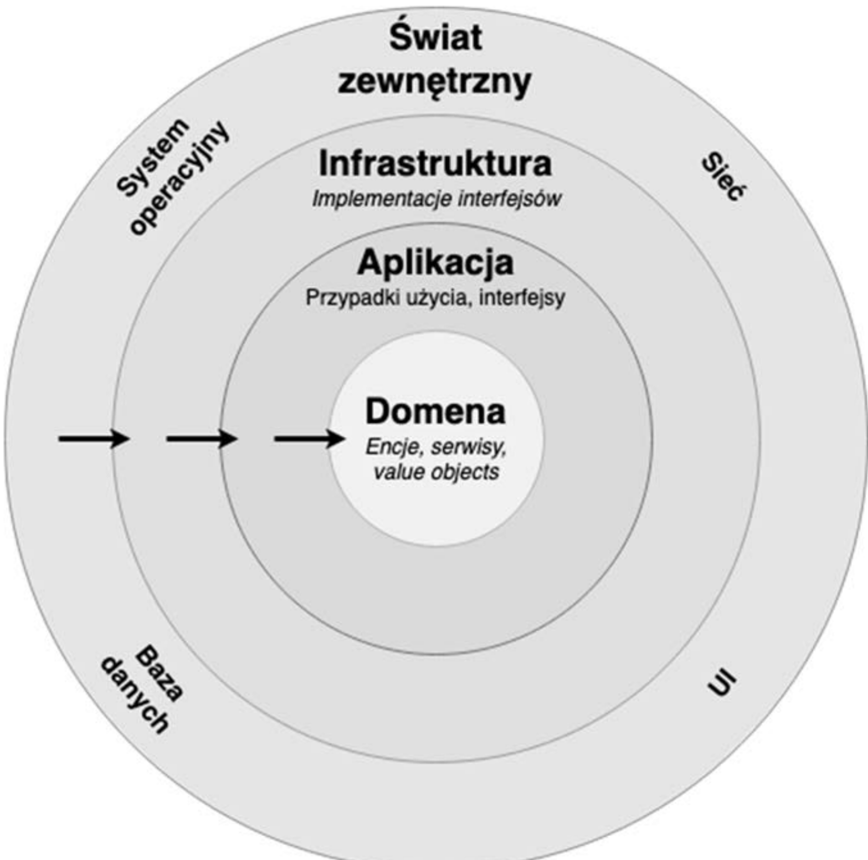
Łatwo jest zastosować bardziej wyrafinowane techniki z czystą architekturą, gdy jest taka potrzeba. Do przykładów zaliczają się CQRS, *event sourcing* czy *domain-driven design*.



Kup ksi k Pole ksi k

## 2.4. Warstwy, czyli horyzontalna organizacja kodu

Projekt napisany według zasad czystej architektury jest zorganizowany w warstwy. W podstawowej i często wystarczającej wersji wyróżniamy cztery warstwy — pokazano je na rysunku 2.1.




Rysunek 2.1. Warstwy w czystej architekturze

### 2.4.1. Świat zewnętrzny

Skrajna warstwa, *świat zewnętrzny*, zawiera wszystkie usługi używane przez projekt, ale nie należące do tej samej bazy kodu. Najprościej mówiąc, ta warstwa obejmuje wszystko, co zostało zaimplementowane poza projektem.



Kup ksi k Pole ksi k

### 2.4.2. Infrastruktura

Druga warstwa, *infrastruktura*, zawiera cały kod potrzebny w projekcie do korzystania z dobrodziejstw znajdujących się w warstwie świata zewnętrznego. Na przykład, jeżeli używamy MariaDB jako głównej bazy danych, klasy i funkcje odpowiedzialne za komunikację z MariaDB będą się znajdować właśnie w warstwie infrastruktury. Podobnie będzie wyglądała sytuacja wszystkich innych usług, z którymi będziemy się integrować w projekcie.

Wszystko zależy od projektu — przykładowo w projekcie z branży e-commerce będziemy potrzebować bramki płatności, a projekt z branży turystycznej pewnie będzie się integrować z systemami rezerwacji lotów, samochodów na wynajem czy hoteli.

### 2.4.3. Aplikacja

Trzecia warstwa, *aplikacja*, będzie domem dla aplikacyjnych reguł biznesowych. Tutaj znajduje się kod określający, co projekt właściwie robi. Są to przede wszystkim **przypadki użycia** (ang. *use cases*), alternatywnie nazywane **interaktorami** (ang. *interactors*). Przypadek użycia reprezentuje pojedynczą operację, która prowadzi do zmiany stanu systemu, pod warunkiem że wszystko pójdzie dobrze.

Poruszamy się w przykładzie systemu aukcyjnego, zatem będziemy potrzebować osobnych przypadków użycia dla składania nowej oferty i wycofywania oferty. Z kolei w sklepie internetowym mielibyśmy przypadki użycia związane z dodawaniem przedmiotu do koszyka, usuwaniem go lub potwierdzaniem zamówienia.

Przypadek użycia reprezentuje pojedynczą akcję czy też polecenie aktora (osoby lub innego systemu). Ta akcja musi być istotna z biznesowego punktu widzenia i prowadzić do zmian w systemie.

Oprócz przypadków użycia w tej warstwie znajdują się **interfejsy** (ang. *interfaces*), znane też jako **porty** (ang. *ports*). Stanowią one warstwę abstrakcji nad wszystkim, co znajduje się w warstwie wyżej, czyli infrastrukturze, a co jest potrzebne w przypadkach użycia. W Pythonie interfejsy można zaimplementować między innymi przy użyciu abstrakcyjnych klas bazowych z modułu `abc`.

```
# application/ports/payments.py
import abc

class Payments(abc.ABC):
    @abc.abstractmethod
    def zero_auth(self, card_token: str) -> None:
        pass
# infrastructure/adapters/xyz_payments.py
```



Kup książkę Poleć książkę

```
from application.ports.payments import Payments

class XyzPayments(Payments):
    """
    Xyz to konkretny dostawca usług płatności
    """
    def zero_auth(self, card_token: str) -> None:
        # korzystamy tu z biblioteki dostarczonej przez Xyz
        # lub bezpośrednio komunikujemy się z API XyZ
        …
```

Powyższe fragmenty kodu pokazują relację pomiędzy interfejsami (portami) z warstwy aplikacji i ich adapterami z warstwy infrastruktury. Te ostatnie dziedziczą po pierwszych. Najważniejsze, by na poziomie przypadku użycia nie było wiadomo, czy właściwie używamy `XyzPayments`. Widzimy tylko abstrakcję `Payments`.

Podsumowując, warstwa *aplikacja* zawiera kod opisujący wszystkie akcje w projekcie i interfejsy dla świata zewnętrznego, wymagane do przeprowadzenia tych akcji. Na przykład w projekcie z branży rozrywkowej jednym z przypadków użycia będzie rezerwacja biletów do muzeum, a interfejsem potrzebnym do realizacji tej logiki — klasa abstrakcyjna do kupowania biletów, stanowiąca abstrakcję nad jakimś konkretnym systemem dostarczonym przez firmę trzecią.

## 2.4.4. Domena

W warstwie *domena* jest miejsce na cały kod wymuszający reguły biznesowe, które muszą być prawdziwe niezależnie od kontekstu użycia (czyli opisane już wcześniej niezmienniki). Podstawowym wzorcem używanym w tej warstwie jest encja.

Jeszcze raz użyjmy przykładu systemu aukcyjnego — tu encją mogłaby być aukcja z metodami pozwalającymi na składanie i wycofywanie ofert. Chronionym niezmiennikiem będzie bieżąca cena aukcji, która musi wynikać z najwyższej oferty.

```
# domain/entities/auction.py
class Auction:
    def place_bid(self, user_id: int, amount: Decimal) -> None:
        …
    def withdraw_a_bid(self, bid_id: int) -> None:
        …
```

Właściwie po co nam encje i implementacja logiki jako ich metoda? Przecież można to samo zrobić w przypadkach użycia:

```
class PlacingBidUseCase:
    def execute(self, user_id: int, amount: Decimal) -> None:
        if auction.current_price < amount:
            auction.winners = [new_bid.bidder_id]
```



Kup ksi k Pole ksi k

```
            auction.current_price = new_bid.amount
        …
```

Takie podejście doprowadziłoby do tak zwanych anemicznych encji. Inna nazwa tych „stworzeń" to *data classes*[1] (nie mylić z modułem `dataclasses` z biblioteki standardowej!) lub *plain old Python objects* (POPO). Anemiczne encje są po prostu prymitywnymi workami na dane bez zdefiniowanych metod (czyli zachowania). Cała logika, która operuje na tych danych, jest wtedy implementowana na zewnątrz. Swoją drogą to podejście też jest pewnym wzorcem — jego właściwa nazwa to *skrypt transakcji* (ang. *transaction script*)[2].

Pozbawienie encji metod może działać w pewnych projektach, ale zdecydowanie nie zadziała, gdy modelujemy problem, w którym występują niezmienniki do ochrony. Dla przykładu każda zmiana zwycięskiej oferty wpływa na bieżącą cenę aukcji. Już teraz znamy co najmniej dwie sytuacje, w których to się zdarza:

1. Ktoś oferuje więcej niż obecny wygrywający uczestnik aukcji.
2. Wycofujemy ofertę, która jednocześnie jest zwycięska.

Z pewnością będziemy mieć osobne przypadki użycia do obsługi tych dwóch akcji, a więc naiwne podejście z encją pozbawioną metod oznaczałoby powielenie logiki przeliczania bieżącej ceny aukcji.

Pośród wielu heurystyk wywodzących się z programowania obiektowego jedna szczególnie przestrzega przed tworzeniem anemicznych encji. Mowa o ***TellDontAsk***[3]. Jej główne założenie brzmi: „Powiedz obiektowi, co ma zrobić. Nie wypytuj go o szczegóły i nie decyduj z zewnątrz o tym, co ma zostać zmienione w jego atrybutach. Powierz pracę obiektowi — ma wszystko, co potrzebne do jej wykonania bez zdradzania struktury swoich danych".

```
class PlacingBidUseCase:
    def execute(self, user_id: int, amount: Decimal) -> None:
        # zamiast dłubać we wnętrznościach obiektu…
        if auction.current_price < amount:
            auction.winners = [new_bid.bidder_id]
            auction.current_price = new_bid.amount

        # …pozwól mu wykonać pracę!
        auction.place_bid(…)
```

---

[1] M. Fowler, *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*, wyd. 2, tłum. A. Watrak, J. Walkowska, Helion, Gliwice 2019 [p. rozdz. 3., *Brzydkie zapaszki w kodzie*].

[2] Tenże, *Architektura systemów…*, dz. cyt. [p. rozdz. 9., *Wzorce logiki dziedziny*].

[3] Tenże, *TellDontAsk*, *https://martinfowler.com/bliki/TellDontAsk.html* [dostęp: 7 stycznia 2022].



Kup książkę                                                    Poleć książkę

## 2.4.5. Zasada zależności

Pogrupowanie klas i funkcji w warstwy jeszcze nie wystarczy do otrzymania przejrzystej i łatwej w utrzymaniu bazy kodu. W trakcie wykonywania kodu, który implementuje czystą architekturę, kontrola musi przecież przekroczyć i tak co najmniej kilka (jeśli nie wszystkie) warstw, by cokolwiek zrobić.

Tyle samo uwagi, ile budowaniu warstw, musimy poświęcić ich interakcji między sobą. Inaczej co właściwie stoi na przeszkodzie, by w warstwie domeny zaimportować kod z frameworka lub jakiejś konkretnej bazy danych? Na straży porządku w czystej architekturze stoi zasada zależności. Mówi ona, że dana warstwa może używać rzeczy znajdujących się w warstwie poniżej i tylko od nich zależeć. Odwrotnie mówiąc, niższa warstwa nie powinna wiedzieć o istnieniu warstwy od niej wyższej.

Na przykład nie wolno używać klas, funkcji lub modułów z warstwy infrastruktury, jeżeli jesteśmy w warstwie domeny. Nie chodzi tylko o jawne importowanie kodu i jego używanie, ale też o przyjmowanie nieznanych w danej warstwie obiektów jako na przykład argumentów funkcji. Zasada zależności została przedstawiona na diagramie 2.1 w postaci strzałek. Ich kierunek pokazuje możliwe zależności:

- infrastruktura używa aplikacji;
- aplikacja używa domeny;
- infrastruktura może używać domeny;

ale

- domena nie może używać aplikacji;
- domena nie może używać infrastruktury;
- aplikacja nie może używać infrastruktury.

W Pythonie nie da się wymusić tych reguł bez użycia zewnętrznych narzędzi. W samym języku nie ma natywnych metod pozwalających wymuszać przyjęte zasady organizowania kodu w warstwy. Nieco lepiej sytuacja wygląda w Javie lub C#, ale nawet tam potrzebne są zewnętrzne biblioteki. Niemniej w Pythonie istnieje co najmniej kilka sposobów na implementację warstw. Zostaną one omówione w dalszej części książki.

## 2.4.6. Granice

Ostatnim, ale nie mniej ważnym składnikiem czystej architektury są granice między warstwami. Określają one sposób, w jaki komunikujemy się z daną warstwą. Warstwa grupuje kod. Ten składa się z klas i funkcji. Większość



Kup książkę Poleć książkę

z nich nie jest przeznaczona do użytku na zewnątrz warstwy — realizują bowiem wewnętrzne mechanizmy, ale nie służą do ich wyciągania i używania osobno. W pewnym sensie są więc *prywatne* dla danej warstwy — stanowią tak zwany szczegół implementacyjny. Oznacza to również, że nikogo spoza tej warstwy nie powinno interesować istnienie danej klasy czy funkcji.

Z racji istnienia w ramach warstwy dwóch rodzajów funkcji/klas — do użytku na zewnątrz i tylko do użytku wewnętrznego — powinniśmy zbudować dla warstwy API. Będzie ono stanowić wskazówkę i pomoc dla każdego, kto chce wykorzystać usługi zapewniane przez daną warstwę.

Z punktu widzenia kodu granica warstwy to po prostu zbiór wybranych klas i ich metod. Rzadziej możemy używać samych funkcji. W oryginalnej koncepcji czystej architektury poleca się też zbudować warstwę abstrakcji w postaci abstrakcyjnych klas bazowych lub `Protocol` z wbudowanego modułu `typing`. Jest to możliwe, ale dalej omówimy, czy i dlaczego można z tego zrezygnować.

Jeśli zdecydujemy się na budowę API danej warstwy, musimy się zastanowić nad argumentami. Jak mówi zasada zależności, warstwa niższa nie może importować, a nawet przyjmować jako argumentów obiektów z klasy wyższej. Na przykład kod z warstwy aplikacji nie może operować na modelach z biblioteki ORM, bo to by oznaczało, że do aplikacji wycieka wiedza o używanej bazie danych. W językach dynamicznie typowanych lub bez wykorzystania adnotacji typów (dostępnych w Pythonie od wersji 3.5) łatwo można to przeoczyć. Na szczęście samo zaimportowanie czegoś z wyższej warstwy, by tylko dodać adnotację typu, zwykle skutkuje przeczuciem, że coś może być nie tak.

**UWAGA!**

By nie mieć z tym problemu, wystarczy przyjąć, że argumenty wejściowe są częścią granicy warstwy, a więc muszą być przez nią definiowane.

W ten sposób nigdy nie przyjmiemy do środka niczego, co nie wchodzi w skład warstwy.

W prawdziwym, produkcyjnym projekcie poszczególne metody API warstwy będą wymagać kilku różnych argumentów. Jak wiemy, im mniej argumentów, tym lepiej. Trikiem, za pomocą którego można sobie z tym poradzić, jest użycie wzorca pod nazwą **obiekt transferu danych** (ang. *data*



Kup ksi k

Pole ksi k

*transfer object*, *DTO*), czyli niemutowalnego obiektu — struktury danych. Do implementacji można zastosować `dataclasses` z biblioteki standardowej:

```
@dataclass(frozen=True)
class EmailDto:
    src: EmailAddress
    reply_to: EmailAddress
    contents: str
```

Najważniejszą granicą jest ta otaczająca warstwę aplikacji. Jak się już zapewne domyślasz, jest ona zbudowana z przypadków użycia. By uniknąć silnego powiązania (ang. *coupling*) pomiędzy klientami aplikacji a samą warstwą, można wprowadzić dodatkową warstwę abstrakcji nad przypadkiem użycia, nazywaną **granicą wejściową** (ang. *input boundary*). Do wywołania przedstawianej w ten sposób akcji potrzebujemy jeszcze odpowiedniego DTO. Analogicznie warstwa aplikacji przekaże wynik operacji w innym DTO. Ta trójka, granica wejściowa, wejściowe DTO i wyjściowe DTO, razem tworzy solidną jak skała granicę, za którą można ukryć wszystkie szczegóły implementacyjne warstwy aplikacji.

Czasem w źródłach można znaleźć inne nazwy, **żądanie** (ang. *request*) i **odpowiedź** (ang. *response*), stosowane zamiennie z wejściowym i wyjściowym obiektem transferu danych. Aby uniknąć zamieszania w związku z takim samym nazewnictwem w protokole HTTP, w całej książce będę używać terminów: wejściowy obiekt transferu danych i wyjściowy obiekt transferu danych. Krócej — wejściowe DTO i wyjściowe DTO.

Instancjom klas typu DTO obowiązkowo zapewniamy niemutowalność (`frozen=True`). Nie ma żadnego powodu, dla którego moglibyśmy chcieć je modyfikować po utworzeniu. Są jak wiadomości — jedna na wejściu, druga na wyjściu.

```
@dataclass(frozen=True)
class PlacingBidInputDto:
    bidder_id: int
    auction_id: int
    amount: Decimal

@dataclass(frozen=True)
class PlacingBidOutputDto:
    is_winning: bool
    current_price: Decimal

class PlacingBidInputBoundary:
    @abc.abstractmethod
    def execute(self, request: PlacingBidInputDto) -> None:
        …
```



Kup książkę    Poleć książkę

## 2.5. Podsumowanie

Jeżeli tylko projekt jest czymś więcej niż pudrowaną przeglądarką do bazy danych, to będziemy w nim mieć mnóstwo reguł biznesowych do wymuszenia i niezmienników do chronienia. Czysta architektura traktuje je jako obywateli pierwszej kategorii, kładąc na nie szczególny nacisk. Oznacza to, że zamiast pozwalać na ukrywanie odrobiny logiki tu i tam w gąszczu kodu modeli bazowanych lub widokach frameworka, grupujemy ją osobno w postaci klas i funkcji w warstwach domeny i aplikacji.

Tak oddzielona logika biznesowa jest łatwa w testowaniu, ponieważ nie jest świadoma istnienia świata zewnętrznego, na przykład baz danych czy usług zewnętrznych. Kod odpowiedzialny za komunikację ze światem zewnętrznym znajduje się w warstwie infrastruktury. Może ona używać aplikacji, ale w drugą stronę jest to niedozwolone. Reguła, która określa dozwolone zależności między warstwami, jest nazywana zasadą zależności:

świat zewnętrzny -> infrastruktura -> aplikacja -> domena

Oczywiście podczas wykonywania kodu przypadku użycia często musimy korzystać z bazy danych lub zewnętrznego API. W takiej sytuacji czysta architektura rekomenduje napisanie w warstwie aplikacji zestawu portów (zwanych także interfejsami), które będą stanowić miejsca wpinania różnych rozszerzeń. Właściwe implementacje, tak zwane adaptery, będą się znajdowały w warstwie infrastruktury.

Aby utrzymać porządek, pomiędzy warstwami stawiamy wyraźne granice. Poszczególne warstwy udostępniają swoje funkcje poprzez interfejsy, które w argumentach metod przyjmują obiekty transferu danych (DTO). Granice warstw, szczególnie aplikacji, pozwalają ukryć przed światem szczegóły implementacyjne. Z zewnątrz warstwa jest widoczna jako zestaw interfejsów i ich argumentów.



Kup ksi k Pole ksi k

# 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

# Implementowanie czystej architektury w **Pythonie**

## Zaawansowane programowanie zaczyna się tam, gdzie kończy się framework

Praca programisty wydaje się dziś znacznie prostsza niż kilkanaście lat temu. Wydaje się taka, ponieważ ma on dostęp do niezliczonych bibliotek przypisanych do języków programowania. Gdy pojawia się problem, sięga do biblioteki — i po sprawie. Problem rozwiązany, zgadza się? I tak, i nie. Owszem, w przypadku prostych projektów wystarczy opanowanie bazowych zasad programowania, podstawowa znajomość danego języka i wiedza na temat tego, co zawiera konkretna biblioteka. Tak jednak działa to jedynie przy nieskomplikowanych aplikacjach. Bez wątpienia dziś łatwiej zacząć programować i szybciej można uzyskać mierzalne efekty, ale…

…prawdziwe programowanie zaczyna się poziom wyżej. Na etapie większych projektów. Bo duże systemy, niezależnie od języka, w jakim zostały napisane, zawsze są trudne — zarówno w rozwijaniu, jak i w utrzymaniu. Książka, którą trzymasz w ręku, została napisana ze świadomością tej programistycznej prawdy. Adresowana do średnio zaawansowanych programistów zajmujących się rozwojem aplikacji internetowych, stanowi kompletny przewodnik po implementacji czystej architektury. Znajdziesz tu także opisy wielu technik, które pomogą Ci zapanować nad projektami rozwijanymi od dłuższego czasu, takich jak strategia testowania czy modularyzacja. Dzięki ich opanowaniu będzie Ci o wiele łatwiej dbać o poprawność funkcjonowania systemów, nad którymi sprawujesz programistyczną pieczę.

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl



ISBN 978-83-283-8686-0
9 788328 386860