
zadanie1

Wydanie 1.0.0

Dawid Szuber

13 lis 2025

Contents

1	rozdzial1	1
2	rozdzial2	3
3	rozdzial3	5
4	rozdzial4	7
5	Indices and tables	11

Czym jest Programowanie Obiektowe (OOP)?

Programowanie obiektowe (z ang. **Object-Oriented Programming**, w skrócie **OOP**) to paradygmat programowania, który zrewolucjonizował sposób tworzenia oprogramowania. Zamiast myśleć o programie jako o sekwencji instrukcji i funkcji operujących na danych (jak w programowaniu proceduralnym), OOP proponuje modelowanie rzeczywistości za pomocą **obiektów**.

Wyobraźmy sobie programowanie proceduralne jako listę zadań dla kucharza: „Weź mąkę”, „Dodaj wodę”, „Wymieszaj”, „Wstaw do pieca”. Jeśli coś pójdzie nie tak (np. zabraknie mąki), cały proces może się załamać, a zmiana jednego kroku może wymagać modyfikacji wielu innych.

Programowanie obiektowe jest jak zorganizowana kuchnia. Zamiast jednej listy zadań, mamy obiekty: „Kucharz”, „Piekarnik”, „Miska”. Każdy obiekt ma swoje własne dane (**atrybuty**, np. zawartość miski) oraz własne funkcje (**metody**, np. zdolność kucharza do mieszania lub zdolność piekarnika do pieczenia). Kucharz nie musi wiedzieć, *jak* działa piekarnik; musi tylko wiedzieć, jak go użyć (wywołać jego funkcję pieczenia).

Główne Założenia i Korzyści

Paradygmat obiektowy opiera się na idei łączenia danych oraz funkcji, które na nich operują, w spójne jednostki zwane obiektami. Prowadzi to do wielu korzyści, szczególnie w dużych i złożonych projektach:

- **Modułowość:** Każdy obiekt jest niezależną jednostką. Możesz rozwijać i testować klasę Samochod bez martwienia się o to, jak działa klasa Uzytkownik.
- **Wielokrotne użycie kodu (Reusability):** Raz zdefiniowana klasa (np. Przycisk) może być używana wielokrotnie w różnych częściach aplikacji. Co więcej, dzięki dziedziczeniu, można tworzyć nowe klasy na bazie istniejących.
- **Łatwiejsze utrzymanie i skalowalność:** Gdy chcesz coś zmienić (np. sposób obliczania podatku), modyfikujesz tylko jedną klasę (KalkulatorPodatku), a nie przeszukujesz całego kodu w poszukiwaniu odpowiednich funkcji.
- **Lepsze odwzorowanie rzeczywistości:** Struktura kodu oparta na obiektach (Klient, Produkt, Zamówienie) jest często bardziej intuicyjna i łatwiejsza do zrozumienia dla programistów, ponieważ przypomina obiekty ze świata rzeczywistego.
- **Elastyczność:** Dzięki polimorfizmowi (omówionemu później), systemy obiektowe mogą łatwo adaptować się do nowych typów danych bez konieczności fundamentalnych zmian w logice.

Klasa vs. Obiekt: Plan i Budynek

Dwa najbardziej fundamentalne pojęcia w OOP to **klasa** i **obiekt**.

1. **Klasa (Class):** To jest plan, szablon, przepis lub projekt. Klasa Samochod definiuje, że samochód *będzie miał* kolor, markę i prędkość maksymalną oraz że *będzie potrafił* wykonywać akcje takie jak „jedź” i „hamuj”. Sama klasa nie jest samochodem – to tylko opis.
2. **Obiekt (Object):** To jest konkretna, fizyczna **instancja** klasy. To prawdziwy byt stworzony na podstawie planu. Na podstawie klasy Samochod możemy stworzyć wiele obiektów:
 - mojeAuto (czerwone Ferrari, prędkość 300 km/h)
 - autoSasiada (niebieski Fiat, prędkość 180 km/h)

Oba obiekty mają te same atrybuty (kolor, marka) i metody (jedź, hamuj), ale wartości ich atrybutów są niezależne i przechowywane oddzielnie.

Kiedy tworzymy obiekt (proces zwany **instancjacją**), zazwyczaj używamy specjalnej metody zwanej **konstruktorem**. Konstruktor pozwala nam ustawić początkowe wartości atrybutów dla nowo tworzonego obiektu (np. podać markę i liczbę przerzutek podczas tworzenia nowego roweru).

Siła programowania obiektowego opiera się na czterech fundamentalnych koncepcjach, często nazywanych filarami OOP. Zrozumienie ich jest kluczowe do efektywnego projektowania systemów obiektowych.

1. Hermetyzacja (Enkapsulacja)

Hermetyzacja (ang. *Encapsulation*) to idea łączenia danych (atrybutów) i metod, które na nich operują, w jedną całość (czyli klasę). Co ważniejsze, hermetyzacja oznacza **ukrywanie wewnętrznego stanu obiektu** przed światem zewnętrznym.

Inne obiekty nie powinny mieć bezpośredniego dostępu do atrybutów „właściciela”. Zamiast tego, dostęp do danych odbywa się za pośrednictwem publicznych metod (tzw. *getterów* i *setterów*).

- **Po co?** Wyobraźmy sobie klasę `KontoBankowe` z atrybutem `saldo`. Gdyby każdy mógł bezpośrednio zmienić saldo na ujemną wartość, doprowadziłoby to do chaosu.
- **Rozwiązanie:** Atrybut `saldo` jest *prywatny*. Oferujemy publiczną metodę „wypłać”, która przyjmuje żądaną kwotę. Ta metoda, zanim zmieni saldo, sprawdzi, czy kwota jest dodatnia i czy na koncie są wystarczające środki.

Hermetyzacja chroni obiekt przed nieprawidłowym użyciem i pozwala autorowi klasy zmienić jej wewnętrzną implementację (np. sposób przechowywania salda) bez psucia kodu, który z tej klasy korzysta.

2. Dziedziczenie

Dziedziczenie (ang. *Inheritance*) to mechanizm, który pozwala tworzyć nową klasę (nazywaną *klasą pochodną* lub *podklasą*) na podstawie istniejącej klasy (*klasy bazowej* lub *nadklasy*).

Klasa pochodna „dziedziczy” wszystkie atrybuty i metody klasy bazowej, może je rozszerzać o nowe lub modyfikować (nadpisywać) istniejące.

- **Przykład:** Mamy klasę bazową `Pojazd`, która ma atrybut `predkosc_maksymalna` i metodę „jedź”.
- Możemy stworzyć klasy pochodne: `Samochod` i `Motocykl`.
- Zarówno `Samochod`, jak i `Motocykl` automatycznie dziedziczą `predkosc_maksymalna` i metodę „jedź”. Nie musimy pisać tego kodu od nowa.
- `Samochod` może dodać nową metodę, np. „włącz wycieraczki”, której `Motocykl` nie będzie miał.

Dziedziczenie modeluje relację „jest” (ang. *is-a*). Samochod **jest** Pojazdem. Motocykl **jest** Pojazdem.

3. Polimorfizm

Polimorfizm (ang. *Polymorphism*) to greckie słowo oznaczające „wielopostaciowość”. W OOP oznacza to zdolność obiektów różnych klas do odpowiadania na to samo wywołanie metody (ten sam komunikat) w sposób specyficzny dla ich typu.

Najczęściej polimorfizm występuje w parze z dziedziczeniem.

- **Przykład:** Kontynuując przykład z Pojazdem. Załóżmy, że zarówno Samochod, jak i Motocykl inaczej implementują metodę „jedź” odziedziczoną z Pojazd:
 - Metoda „jedź” w Samochod wydrukuję: „Wrum, wrum, jadę czterema kołami.”
 - Metoda „jedź” w Motocykl wydrukuję: „Wrrrr, jadę na dwóch kołach!”
- Teraz, jeśli mamy listę różnych pojazdów (niektóre to samochody, inne to motocykle), możemy przejść przez tę listę i na każdym obiekcie wywołać *tę samą* metodę „jedź”.
- System automatycznie rozpozna, czy dany obiekt to Samochod, czy Motocykl i uruchomi właściwą (specyficzną dla niego) wersję tej metody.

4. Abstrakcja

Abstrakcja (ang. *Abstraction*) to proces ukrywania złożonych szczegółów implementacyjnych i pokazywania użytkownikowi tylko niezbędnych funkcji. Jest to „wyższy poziom” hermetyzacji.

Celem abstrakcji jest uproszczenie interakcji z obiektem.

- **Przykład:** Kiedy prowadzisz samochód, używasz pedału gazu. Jest to **abstrakcja**. Naciskasz pedał, a samochód przyspiesza.
- Nie musisz wiedzieć (i nie chcesz wiedzieć) o tym, jak działa przepustnica, wtrysk paliwa, jaki jest skład mieszanki paliwowo-powietrznej i jak steruje tym komputer pokładowy. Te skomplikowane detale są **ukryte** (zaimplementowane wewnątrz).
- Twój interfejs jest prosty: pedał gazu.

W programowaniu abstrakcję często realizuje się za pomocą **klas abstrakcyjnych** lub **interfejsów**. Definiują one, *co* obiekt musi potrafić zrobić (np. „każdy pojazd musi umieć jechać i hamować”), ale nie mówią, *jak* ma to zrobić. Dopiero konkretne klasy (jak Samochod) implementują te metody.

Opanowanie czterech filarów to podstawa, ale efektywne programowanie obiektowe wymaga również znajomości wzorców i zasad projektowania, które pomagają tworzyć kod elastyczny, łatwy w utrzymaniu i odporny na błędy.

Konstruktory i Destruktory

Obiekty mają swój cykl życia.

- **Konstruktor:** To specjalna metoda, która jest automatycznie wywoływana w momencie tworzenia nowego obiektu (instancjacji). Jej głównym zadaniem jest inicjalizacja atrybutów obiektu, czyli nadanie im wartości początkowych (np. ustawienie salda na 0 dla nowego konta bankowego).
- **Destruktor:** To metoda (rzadziej używana w językach z automatycznym zarządzaniem pamięcią, jak Python czy Java) wywoływana tuż przed zniszczeniem obiektu. Może służyć do „posprzątania” po obiekcie, np. zamknięcia połączenia z bazą danych lub zwolnienia zasobów systemowych.

Dziedziczenie a Kompozycja

Dziedziczenie (relacja „jest”) to potężne narzędzie, ale bywa nadużywane. Często lepszym podejściem jest **kompozycja** (relacja „ma”).

- **Dziedziczenie (relacja „jest”):** Samochod **jest** Pojazdem.
- **Kompozycja (relacja „ma”):** Samochod **ma** Silnik.

Zamiast tworzyć klasę SamochodKtoryMaRadio dziedziczącą z Samochod, lepszym podejściem jest, aby klasa Samochod *zawierała w sobie* (jako atrybut) obiekt klasy Radio.

Dlaczego kompozycja jest często preferowana? Jest bardziej elastyczna. Możesz łatwo podmienić silnik w samochodzie (zmienić obiekt Silnik) w trakcie działania programu, czego nie da się zrobić z dziedziczeniem (samochód nie może przestać być pojazdem).

Porównanie: Dziedziczenie vs Kompozycja

Poniższa tabela podsumowuje kluczowe różnice między tymi dwoma podejściami do ponownego wykorzystania kodu.

Cecha	Dziedziczenie (Relacja „IS-A”)	Kompozycja (Relacja „HAS-A”)
D efinicja	Klasa pochodna dziedziczy po klasie bazowej.	Klasa „główna” zawiera instancję innej klasy jako atrybut.
** Przykład	Kwadrat dziedziczy po Prostokąt.	Samochod zawiera obiekt Silnik.
** Wiązani	Silne (ang. <i>tight coupling</i>). Zmiany w klasie bazowej mocno wpływają na pochodne.	Luźne (ang. <i>loose coupling</i>). Zmiany w klasie „zawartej” (Silnik) mają mniejszy wpływ na „zawierającą” (Samochód).
Elas tyczność	Mniejsza. Relacja jest ustalana na etapie kompilacji.	Większa. Obiekty „zawarte” można podmieniać w trakcie działania programu (np. wymiana silnika).
Kiedy używać	Gdy nowa klasa jest faktycznie <i>specjalnym typem</i> klasy bazowej i chcemy korzystać z polimorfizmu.	Gdy klasa <i>potrzebuje</i> funkcjonalności innej klasy, ale nie jest jej specjalnym typem. „Preferuj kompozycję nad dziedziczeniem”.

Zasady SOLID

Pisanie dobrego kodu obiektowego to sztuka. Zbiór pięciu fundamentalnych zasad projektowania, znanych pod akronimem **SOLID**, pomaga tworzyć oprogramowanie, które jest zrozumiałe, elastyczne i łatwe w utrzymaniu.

1. S (Single Responsibility Principle): Zasada jednej odpowiedzialności.

- *Mówi, że:* Każda klasa powinna mieć tylko jeden powód do zmiany (powinna być odpowiedzialna tylko za jeden aspekt funkcjonalności).
- *Przykład:* Klasa odpowiedzialna za logikę biznesową użytkownika nie powinna jednocześnie zajmować się zapisywaniem go do bazy danych. Zapis do bazy powinien być w osobnej klasie.

2. O (Open/Closed Principle): Zasada otwarte/zamknięte.

- *Mówi, że:* Klasy powinny być otwarte na rozszerzenia, ale zamknięte na modyfikacje.
- *Przykład:* Zamiast modyfikować klasę kalkulatora wysyłki za każdym razem, gdy dodajemy nową firmę kurierską, klasa ta powinna korzystać ze wspólnego interfejsu, a nowe firmy (DPD, InPost) powinny implementować ten interfejs.

3. L (Liskov Substitution Principle): Zasada podstawienia Liskov.

- *Mówi, że:* Obiekty klasy pochodnej muszą być w stanie zastąpić obiekty klasy bazowej bez powodowania błędów i zmiany oczekiwanego zachowania programu.
- *Przykład:* Jeśli funkcja działa poprawnie z obiektem klasy Ptaszek, musi działać tak samo poprawnie, gdy prześlemy jej obiekt Kaczka (która dziedziczy po Ptaszek).

4. I (Interface Segregation Principle): Zasada segregacji interfejsów.

- *Mówi, że:* Lepiej jest mieć wiele małych, specyficznych interfejsów niż jeden duży, ogólny. Klienci (klasy) nie powinni być zmuszani do implementowania metod, których nie używają.
- *Przykład:* Zamiast jednego interfejsu „Pracownik” z metodami „pracuj” i „zarządzaj zespołem”, lepiej stworzyć dwa osobne interfejsy dla pracownika i dla menedżera.

5. D (Dependency Inversion Principle): Zasada odwrócenia zależności.

- *Mówi, że:* Moduły wysokiego poziomu (logika biznesowa) nie powinny zależeć od modułów niskiego poziomu (np. obsługa bazy danych). Oba powinny zależeć od abstrakcji (interfejsów).
- *Przykład:* Klasa generująca raporty nie powinna być na sztywno powiązana z bazą danych MySQL. Powinna polegać na abstrakcyjnym interfejsie „BazaDanych”, co pozwoli w przyszłości łatwo podmienić MySQL na inną bazę.

Zrozumienie teorii programowania obiektowego to pierwszy krok. Drugim, znacznie ważniejszym, jest przełożenie tej teorii na praktykę i utrwalenie jej poprzez budowanie konkretnych aplikacji. Wiedza, która nie jest aktywnie wykorzystywana, szybko zanika. Poniżej znajduje się lista zasobów – klasycznych książek, materiałów dodatkowych oraz przykładowych projektów – które pomogą Ci nie tylko pogłębić wiedzę, ale przede wszystkim stać się lepszym programistą obiektowym.

Przykładowe Projekty do Nauki OOP

Najlepszym sposobem na naukę jest samodzielne zmaganie się z problemami projektowymi. Oto kilka pomysłów na projekty o rosnącym stopniu skomplikowania, które idealnie nadają się do przećwiczenia zasad OOP:

1. Projekt: Prosty System Biblioteczny (konsolowy)

- **Cel:** Przećwiczenie podstaw: tworzenia klas, atrybutów, metod i relacji między obiektami.
- **Kluczowe obiekty (Klasy):**
 - Książka: Powinna mieć atrybuty takie jak tytuł, autor, ISBN.
 - Czytelnik: Powinien mieć atrybuty imię, nazwisko, id_czytelnika oraz listę aktualnie wypożyczonych książek.
 - Biblioteka: Główna klasa zarządzająca. Powinna zawierać listy wszystkich książek i wszystkich czytelników. Metody tej klasy to np. dodajKsiazke, dodajCzytelnika, wypożyczKsiazke (która sprawdza dostępność i dodaje książkę do listy czytelnika) oraz zwrocKsiazke.
- **Co ćwiczysz? Hermetyzację** (np. Biblioteka zarządza stanem książek, a nie sam czytelnik) oraz proste relacje (Biblioteka „ma” Książki i Czytelników).

2. Projekt: Gra Tekstowa (Text-based RPG)

- **Cel:** Zastosowanie dziedziczenia i polimorfizmu.
- **Kluczowe obiekty (Klasy):**
 - Postac (Klasa bazowa/abstrakcyjna): Powinna definiować wspólne cechy, jak punktyZycia, silaAtaku oraz metody atakuj i otrzymajObrazenia.

- Bohater (dziedziczy po Postac): Dodaje specyficzne cechy, jak ekwipunek (który sam może być listą obiektów klasy Przedmiot) oraz metodę `uzyjPrzedmiotu`.
- Przeciwnik (dziedziczy po Postac): Może mieć dodatkowy atrybut `typ` (np. Goblin, Smok).
- Bron (dziedziczy po Przedmiot): Może nadpisywać metodę `uzyj` z klasy Przedmiot, aby zwiększać siłę ataku bohatera.
- Mikstura (dziedziczy po Przedmiot): Nadpisuje metodę `uzyj`, aby leczyć bohatera.
- **Co ćwiczysz? Dziedziczenie** (Bohater „jest” Postacią, Smok „jest” Postacią) oraz **Polimorfizm** (wywołanie metody `atakuj` na obiekcie Bohater i na obiekcie Smok działa inaczej; wywołanie `uzyj` na Mikstura i na Bron daje zupełnie różne efekty).

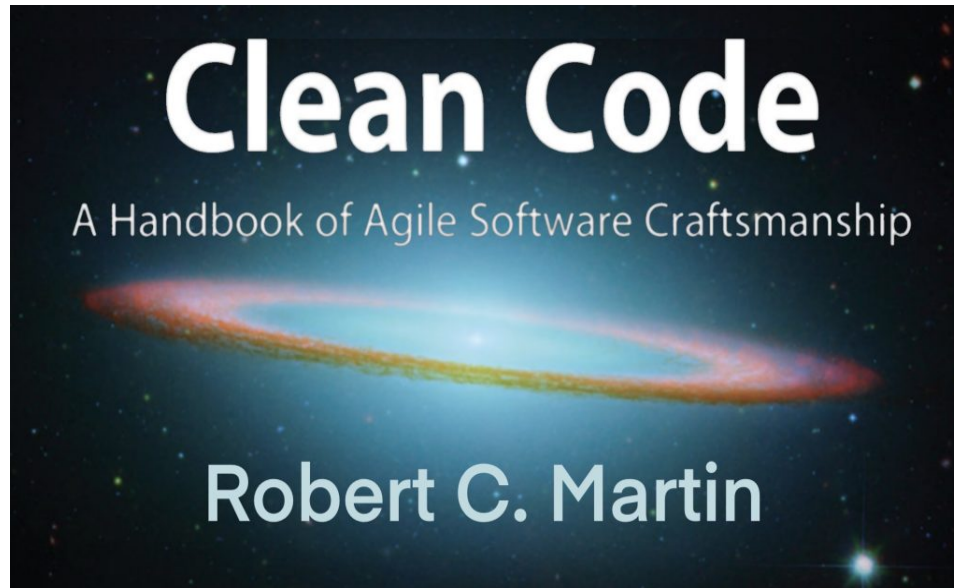
3. Projekt: Symulator Bankomatu

- **Cel:** Zrozumienie hermetyzacji, kompozycji i obsługi wyjątków w kontekście OOP.
- **Kluczowe obiekty (Klasy):**
 - KontoBankowe: Przechowuje saldo. Powinno mieć *prywatny* atrybut `saldo` i publiczne metody `wplac`, `wypłac` oraz `sprawdzSaldo`. Metoda `wypłac` musi zawierać logikę sprawdzającą, czy na koncie są wystarczające środki.
 - Uzytkownik: Posiada imię, nazwisko oraz (poprzez **kompozycję**) obiekt `KontoBankowe`.
 - Bank: Zarządza listą użytkowników i ich kont.
 - Bankomat: Główna klasa aplikacji, która dostarcza interfejsu (np. konsolowego). Używa obiektu `Bank` do weryfikacji użytkownika (np. po numerze PIN) i wykonywania operacji na jego koncie.
- **Co ćwiczysz? Hermetyzację** (nikt poza `KontoBankowe` nie może bezpośrednio zmienić `saldo`), **Kompozycję** (`Uzytkownik` „ma” `KontoBankowe`) oraz zasady SOLID (np. zasada jednej odpowiedzialności – Bankomat odpowiada za interfejs, a `KontoBankowe` za logikę `saldo`).

Klasyczne Książki o OOP i Projektowaniu

Gdy już zaczniesz pisać własne projekty, szybko natkniesz się na powtarzalne problemy. W tym momencie warto sięgnąć po literaturę, która opisuje sprawdzone rozwiązania. Istnieje kilka książek, które są uznawane za kanon w świecie inżynierii oprogramowania.

- **„Wzorce Projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku”** (autorzy: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - tzw. „Banda Czterech” lub „GoF”).
 - To absolutna biblia wzorców projektowych. Opisuje 23 fundamentalne wzorce, które rozwiązują typowe problemy w projektowaniu oprogramowania (np. wzorzec Fabryka, Obserwator, Dekorator). Jest to lektura gęsta i akademicka, ale niezwykle wartościowa dla zaawansowanych programistów.
- **„Czysty kod. Podręcznik dobrego programisty”** (autor: Robert C. Martin - „Wujek Bob”).
 - Choć nie jest to książka *tylko* o OOP, to zasady w niej zawarte (jak zasada SOLID, czytelne nazewnictwo, dobre funkcje) są fundamentem dojrzałego programowania obiektowego. Uczy, jak pisać kod, który jest czytelny dla innych ludzi i łatwy w utrzymaniu. Jest to pozycja obowiązkowa.



- **„Rusz głową! Wzorce projektowe”** (autorzy: Eric Freeman, Elisabeth Robson).
 - Jeśli książka „Bandy Czterech” wydaje się zbyt sucha i odstrasząca, ta pozycja jest idealnym startem. Tłumaczy te same wzorce projektowe w znacznie bardziej przystępny, wizualny i humorystyczny sposób, używając wielu analogii i przykładów z życia.
- **„Refaktoryzacja. Ulepszanie struktury istniejącego kodu”** (autor: Martin Fowler).
 - OOP nie polega tylko na pisaniu kodu od zera, ale także na ulepszaniu istniejącego. Często dołączamy do projektu, gdzie kod jest źle zorganizowany. Ta książka to katalog „ruchów refaktoryzacyjnych” – małych, bezpiecznych kroków, które pozwalają przekształcić zły kod w dobry, często wykorzystując do tego zasady OOP i wzorce projektowe.

Inne Zasoby i Dalsza Ścieżka

Poza książkami i własnymi projektami, istnieje wiele innych miejsc, z których można czerpać wiedzę i inspirację:

- **Platformy e-learningowe:** Strony takie jak Udemy, Coursera, Pluralsight czy polskie (np. Strefa Kursów) oferują dziesiątki kursów wideo na temat OOP, wzorców projektowych i zasad SOLID w kontekście konkretnych języków (Java, C#, Python, JavaScript). Często prowadzą one krok po kroku przez budowę projektu podobnego do tych wymienionych powyżej.
- **Oficjalna Dokumentacja:** Jeśli uczysz się OOP w konkretnym języku (np. Pythonie), czytanie oficjalnej dokumentacji na temat klas, dziedziczenia i wbudowanych modułów jest nieocenione. To podstawowe źródło prawdy.
- **Społeczności i Blogi:** Strony takie jak Stack Overflow (do rozwiązywania konkretnych problemów) oraz blogi prowadzone przez doświadczonych programistów (np. blog Martina Fowlera lub polskie blogi branżowe) to kopalnia wiedzy o rzeczywistych zastosowaniach i problemach związanych z OOP.
- **Repozytoria Open Source (GitHub):** Analizowanie kodu dobrze napisanych projektów open source to jedna z najlepszych metod nauki. Możesz zobaczyć, jak doświadczeni programiści stosują zasady OOP, wzorce projektowe i zasady SOLID w praktyce do rozwiązywania realnych, skomplikowanych problemów. To jak czytanie książek napisanych przez mistrzów, ale w formie kodu.

CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`