Programowanie Obiektowe: Teoria i Praktyka

Dawid Szuber

03.12.2025

Streszczenie

Niniejsza praca stanowi kompleksowe wprowadzenie do paradygmatu programowania obiektowego (OOP). Opracowanie prowadzi czytelnika od podstawowych definicji klasy i obiektu, poprzez omówienie czterech fundamentalnych filarów OOP (hermetyzacja, dziedziczenie, polimorfizm, abstrakcja), aż po zaawansowane dobre praktyki, takie jak zasady SOLID oraz różnice między dziedziczeniem a kompozycją. Całość zwieńczona jest przeglądem literatury branżowej oraz propozycjami projektów praktycznych.

Spis treści

1 Wprowadzenie do Programowania Obiektowego **2**
1.1 Czym jest Programowanie Obiektowe (OOP)? 2
1.2 Główne Założenia i Korzyści 2
1.3 Klasa vs. Obiekt: Plan i Budynek 2
1.3.1 Klasa (Class) 3
1.3.2 Obiekt (Object) 3

2 Cztery Filary Programowania Obiektowego **4**
2.1 Hermetyzacja (Enkapsulacja) 4
2.2 Dziedziczenie 4
2.3 Polimorfizm 4
2.4 Abstrakcja 4

3 Zaawansowane Koncepcje i Dobre Praktyki **6**
3.1 Konstruktory i Destruktory 6
3.2 Dziedziczenie a Kompozycja 6
3.2.1 Porównanie: Dziedziczenie vs Kompozycja 6
3.3 Zasady SOLID 6

4 Co Dalej? Nauka w Praktyce, Projekty i Zasoby **8**
4.1 Przykładowe Projekty do Nauki OOPs 8
4.2 Klasyczne Książki o OOPs 8
4.3 Inne Zasoby 9

5 wnioski **10**

Rozdział 1

Wprowadzenie do Programowania Obiektowego

1.1 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, OOP proponuje modelowanie rzeczywistości za pomocą **obiektów**.

Wyobraźmy sobie programowanie proceduralne jako listę zadań dla kucharza: „Weź mąkę", „Dodaj wodę", „Wymieszaj". Jeśli coś pójdzie nie tak, cały proces może się załamać. Programowanie obiektowe jest jak zorganizowana kuchnia. Mamy obiekty: „Kucharz", „Piekarnik", „Miska". Każdy obiekt ma swoje własne dane (**atrybuty**) oraz własne funkcje (**metody**).

1.2 Główne Założenia i Korzyści

Paradygmat obiektowy opiera się na idei łączenia danych oraz funkcji w spójne jednostki. Prowadzi to do wielu korzyści:

- **Modułowość:** Każdy obiekt jest niezależną jednostką.
- **Wielokrotne użycie kodu:** Raz zdefiniowana klasa może być używana wielokrotnie.
- **Łatwiejsze utrzymanie:** Modyfikujesz tylko jedną klasę zamiast przeszukiwać cały kod.
- **Lepsze odwzorowanie rzeczywistości:** Struktura kodu jest bardziej intuicyjna.
- **Elastyczność:** Systemy mogą łatwo adaptować się do nowych typów danych dzięki polimorfizmowi.

1.3 Klasa vs. Obiekt: Plan i Budynek

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

1.3.1 Klasa (Class)

To jest plan, szablon lub projekt. Klasa `Samochod` definiuje, że samochód *będzie miał* kolor i markę oraz że *będzie potrafił* jechać. Sama klasa nie jest samochodem – to tylko opis.

1.3.2 Obiekt (Object)

To jest konkretna, fizyczna **instancja** klasy. Na podstawie klasy `Samochod` możemy stworzyć wiele obiektów (np. czerwone Ferrari, niebieski Fiat). Oba obiekty mają te same atrybuty i metody, ale wartości ich atrybutów są niezależne.

Kiedy tworzymy obiekt (proces zwany **instancjacją**), używamy specjalnej metody zwanej **konstruktorem**, która pozwala ustawić początkowe wartości atrybutów.

Rozdział 2

Cztery Filary Programowania Obiektowego

Siła programowania obiektowego opiera się na czterech fundamentalnych koncepcjach.

2.1 Hermetyzacja (Enkapsulacja)

Hermetyzacja to idea łączenia danych i metod w jedną całość oraz **ukrywanie wewnętrznego stanu obiektu** przed światem zewnętrznym. Inne obiekty nie powinny mieć bezpośredniego dostępu do atrybutów. Dostęp odbywa się przez publiczne metody (gettery i settery).

Przykład: W klasie `KontoBankowe` atrybut `saldo` jest prywatny. Metoda „wypłać" sprawdza poprawność operacji przed zmianą stanu, co chroni obiekt przed nieprawidłowym użyciem.

2.2 Dziedziczenie

Dziedziczenie pozwala tworzyć nową klasę (pochodną) na podstawie istniejącej klasy (bazowej). Klasa pochodna dziedziczy atrybuty i metody, mogąc je rozszerzać lub nadpisywać.

Przykład: Klasy `Samochod` i `Motocykl` dziedziczą po klasie `Pojazd`. Modeluje to relację „jest" (ang. *is-a*).

2.3 Polimorfizm

Polimorfizm to zdolność obiektów różnych klas do odpowiadania na to samo wywołanie metody w sposób specyficzny dla ich typu.

Przykład: Metoda „jedź" działa inaczej dla klasy `Samochod` i inaczej dla klasy `Motocykl`, ale wywołujemy ją tak samo.

2.4 Abstrakcja

Abstrakcja to proces ukrywania złożonych szczegółów implementacyjnych i pokazywania użytkownikowi tylko niezbędnych funkcji.

Przykład: Używasz pedału gazu (interfejs), nie martwiąc się o wtrysk paliwa (implementacja). W kodzie realizuje się to przez klasy abstrakcyjne lub interfejsy.

Rozdział 3

Zaawansowane Koncepcje i Dobre Praktyki

Efektywne programowanie wymaga znajomości wzorców i zasad, które pomagają tworzyć kod elastyczny.

3.1 Konstruktory i Destruktory

- **Konstruktor:** Metoda wywoływana automatycznie przy tworzeniu obiektu (inicjalizacja).
- **Destruktor:** Metoda wywoływana przed zniszczeniem obiektu (zwalnianie zasobów).

3.2 Dziedziczenie a Kompozycja

Często lepszym podejściem od dziedziczenia (relacja „jest") jest **kompozycja** (relacja „ma").

- Dziedziczenie: `Samochod` **jest** `Pojazdem`.
- Kompozycja: `Samochod` **ma** `Silnik`.

Kompozycja jest bardziej elastyczna i pozwala na luźniejsze powiązania między klasami.

3.2.1 Porównanie: Dziedziczenie vs Kompozycja

Poniższa tabela podsumowuje kluczowe różnice:

3.3 Zasady SOLID

Zbiór pięciu zasad SOLID pomaga tworzyć zrozumiałe oprogramowanie:

1. **S (Single Responsibility):** Jedna klasa, jedna odpowiedzialność.
2. **O (Open/Closed):** Otwarte na rozszerzenia, zamknięte na modyfikacje.

Cecha	**Dziedziczenie („IS-A”)**	**Kompozycja („HAS-A”)**
Definicja	Klasa pochodna dziedziczy po bazowej.	Klasa zawiera instancję innej klasy.
Wiązanie	Silne (*tight coupling*).	Luźne (*loose coupling*).
Elastyczność	Mniejsza (ustalana przy kompilacji).	Większa (możliwa wymiana w trakcie działania).

Tabela 3.1: Porównanie dziedziczenia i kompozycji.

3. **L (Liskov Substitution):** Podklasy muszą móc zastąpić klasy bazowe.
4. **I (Interface Segregation):** Wiele małych interfejsów zamiast jednego dużego.
5. **D (Dependency Inversion):** Zależność od abstrakcji, nie od konkretów.

Rozdział 4

Co Dalej? Nauka w Praktyce, Projekty i Zasoby

Zrozumienie teorii to pierwszy krok. Drugim jest praktyka.

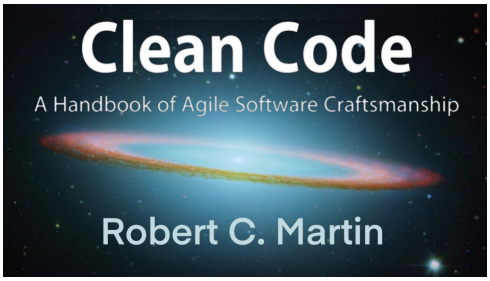
4.1 Przykładowe Projekty do Nauki OOPs

- **Projekt 1: Prosty System Biblioteczny.** Ćwiczy podstawy klas i hermetyzację.
- **Projekt 2: Gra Tekstowa (RPG).** Zastosowanie dziedziczenia i polimorfizmu.
- **Projekt 3: Symulator Bankomatu.** Ćwiczy kompozycję i obsługę wyjątków.

4.2 Klasyczne Książki o OOPs

Warto sięgnąć po literaturę uznawaną za kanon:

- **„Wzorce Projektowe”** (Banda Czterech) [1].
- **„Czysty kod”** (Robert C. Martin) [2].



Rysunek 4.1: Okładka książki „Czysty kod” Roberta C. Martina.

- **„Rusz głową! Wzorce projektowe”** [3].
- **„Refaktoryzacja”** (Martin Fowler) [4].

4.3 Inne Zasoby

Warto również korzystać z platform e-learningowych, oficjalnej dokumentacji oraz analizować kod open source na GitHubie.

Rozdział 5

wnioski

W tym projekcie, podczas doboru klasy do tekstu w latexie, podjęłem decyzję o wyborze klasy dokumentu `report`, rezygnując z klasy `book`.

Główne powody, dla których klasa `report` jest lepszym rozwiązaniem w tym przypadku:

1. **Struktura dokumentu:** Klasa `book` jest przeznaczona do składu pełnowymiarowych książek. Dzieli ona treść na części (*parts*), rozdziały (*chapters*) i sekcje, a także wprowadza podział na *frontmatter* (wstęp), *mainmatter* (treść główna) i *backmatter* (dodatki). Dla pracy o objętości kilkunastu stron taka struktura jest nadmiarowa. Klasa `report` idealnie sprawdza się w przypadku prac dyplomowych i raportów technicznych, które składają się z kilku rozdziałów, ale nie wymagają skomplikowanej struktury książkowej.

2. **Formatowanie stron (Jednostronne vs Dwustronne):** Domyślnie klasa `book` zakłada druk dwustronny. Oznacza to, że marginesy na stronach parzystych i nieparzystych są różne (szerszy margines wewnętrzny na oprawę), a każdy nowy rozdział musi zaczynać się na stronie nieparzystej (prawej). Powoduje to powstawanie wielu pustych stron w krótkich dokumentach. Klasa `report` jest bardziej elastyczna i czytelna przy przeglądaniu dokumentu na ekranie komputera, nie generując zbędnych pustych stron.

3. **Nagłówki i stopki:** W klasie `book` domyślne nagłówki są przygotowane pod druk dwustronny (tytuł rozdziału na lewej stronie, tytuł sekcji na prawej). W krótszej pracy prostszy układ nagłówków oferowany przez klasę `report` zapewnia większą przejrzystość i jest łatwiejszy w konfiguracji.

Dostęp do kodu źródłowego

Kod źródłowy projektu jest dostępny w repozytorium GitHub pod adresem:

`https://github.com/Dejvid1/latexNaOcene.git`

Bibliografia

[1] Gamma E., Helm R., Johnson R., Vlissides J., *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku.* Helion.

[2] Martin R. C., *Czysty kod. Podręcznik dobrego programisty.* Helion.

[3] Freeman E., Robson E., *Rusz głową! Wzorce projektowe.* Helion.

[4] Fowler M., *Refaktoryzacja. Ulepszanie struktury istniejącego kodu.* Helion.