Wprowadzenie

Projektując, implementując lub rozszerzając pewną funkcjonalność w aplikacji, musimy zawsze zastanowić się, w jaki sposób ustrukturyzować nasz kod. Zamiast wymyślać nowe rozwiązania, warto rozważyć czy ktoś już nie rozwiązał podobnego problemu, jaki my aktualnie rozważamy.

Wzorce projektowe są zestawem gotowych szkieletów rozwiązań, które powinniśmy wykorzystywać w naszych aplikacjach. Sekcja ta oparta jest o wzorce opisane w książce "Wzorce Projektowe: Elementy oprogramowania obiektowego wielokrotnego użytku". Praca ta została napisana przez czterech autorów: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - potocznie mówi się o nich Gang of Four. Pomimo, że od jej wydania minęło ponad 25 lat, opisane problemy i ich rozwiązania wciąż pozostają aktualne.

Wszystkie wzorce projektowe zostały podzielone na trzy główne grupy:

- creational (wzorce konstrukcyjne), które opisują, w jaki sposób tworzyć obiekty
- structural (wzorce strukturalne), które wskazują jak wiązać ze sobą obiekty
- behavioral (wzorce behawioralne), które opisują sposób, w jaki powiązane obiekty się ze sobą komunikują i jakie jest ich zachowanie względem innych.

Wzorce projektowe można też podzielić ze względu na zakres stosowania na:

- Klasowe -> opisują związki między klasami i ich podklasami, są ustanowione poprzez dziedziczenie, są statyczne
- Obiektowe -> opisują związki między obiektami, są statyczne - czasu wykonania

Pomimo że nazwy wzorców mają swoje polskie odpowiedniki, w tej sekcji opierać będziemy się na ich anglojęzycznych nazwach.

Nie można jednak wybrać wzorca i po prostu skopiować go do programu, jak bibliotekę czy funkcję zewnętrznego dostawcy. Wzorzec nie jest konkretnym fragmentem kodu, ale ogólną koncepcją pozwalającą rozwiązać dany problem. Postępując według wzorca możesz zaimplementować rozwiązanie które będzie pasować do realiów twojego programu.

Wzorce często myli się z algorytmami, ponieważ obie koncepcje opisują typowe rozwiązanie jakiegoś znanego problemu. Algorytm jednak zawsze definiuje wyraźny zestaw czynności które prowadzą do celu, zaś wzorzec to wysokopoziomowy opis rozwiązania. Kod powstały na podstawie jednego wzorca może wyglądać zupełnie inaczej w różnych programach.

Algorytm jest jak przepis kulinarny: oba mają wyraźnie określone etapy które trzeba wykonać w określonej kolejności by osiągnąć cel. Wzorzec bardziej przypomina strategię: znany jest wynik i założenia, ale dokładna kolejność implementacji należy do ciebie.

Dobre praktyki

Zamin zaczniemy omawiać sobie konkretne wzorce, warto omówi sobie jeszcze jeden bardzo ważny temat w ontekście programowania - dobre praktyki. Jednną z najważniejszych, a właściwie najważniejszą jest SOLID.

Oczywiście! SOLID to akronim, który opisuje pięć podstawowych zasad projektowania obiektowego. Te zasady pomagają w tworzeniu oprogramowania, które jest łatwiejsze do zrozumienia, modyfikacji i rozbudowy. Oto one:

1. S - Single Responsibility Principle (Zasada Jednej Odpowiedzialności). Każda klasa powinna mieć tylko jedną odpowiedzialność, czyli jeden powód do zmiany. Dzięki temu, jeśli zachodzi potrzeba modyfikacji, wpływa to tylko na jedną klasę, a nie na wiele innych powiązanych klas.
2. O - Open/Closed Principle (Zasada Otwarte/Zamknięte). Oprogramowanie powinno być otwarte na rozszerzenia, ale zamknięte na modyfikacje. Oznacza to, że powinniśmy być w stanie dodawać nowe funkcje bez konieczności modyfikowania istniejącego kodu.
3. L - Liskov Substitution Principle (Zasada Podstawienia Liskov). Obiekty klasy bazowej powinny być w stanie być zastąpione przez obiekty klasy pochodnej bez wpływania na poprawność programu. Innymi słowy, jeśli mamy klasę pochodną, która dziedziczy po klasie bazowej, to powinniśmy być w stanie użyć obiektu klasy pochodnej wszędzie tam, gdzie używaliśmy obiektu klasy bazowej, bez zakłócania działania programu.
4. I - Interface Segregation Principle (Zasada Segregacji Interfejsu). Lepiej jest mieć wiele specyficznych interfejsów niż jeden uniwersalny. Oznacza to, że nie powinniśmy zmuszać klasy do implementowania interfejsów, których nie używa. Klasy powinny implementować tylko te metody, które są dla nich istotne.
5. D - Dependency Inversion Principle (Zasada Odwrócenia Zależności). Moduły wyższego poziomu nie powinny zależeć od modułów niższego poziomu. Oba powinny zależeć od abstrakcji. Innymi słowy, zamiast polegać na konkretnych implementacjach, nasz kod powinien polegać na abstrakcjach (np. interfejsach).

Podsumowując, SOLID to zbiór zasad, które pomagają w tworzeniu oprogramowania, które jest bardziej elastyczne, łatwiejsze do utrzymania i rozbudowy. Stosowanie tych zasad może początkowo wydawać się trudne, ale z czasem przynosi korzyści w postaci bardziej przejrzystego i trwałego kodu.

Zerknijmy sobie na pewien przykład:

- Czy dostrzegacie, która z wyżej wymenionych zasad została złamana?

Brak zastosowania reguły Single Responsibility.
Powyższa klasa, oprócz opisywania modelu książki, próbuje również dać możliwość ich wyszukiwania. Do tego celu lepiej wydzielić osobną klasę.

Ograniczenie odpowiedzialności nie oznacza, że klasa lub interfejs powinny definiować tylko jedną metodę. Ważne jest to, żeby nie mieszać odpowiedzialności klasy. Na przykład, wyszukiwarka może mieć osobne metody, które wyszukują rekordy po określonym kluczu,

ale osobno powinniśmy zaimplementować klasę odpowiedzialną za filtrowanie czy sortowanie.

Co składa się na wzorzec?

Większość wzorców posiada formalny opis, dzięki czemu każdy może odtworzyć ich ideę w różnych kontekstach. Oto sekcje na które zwykle dzieli się opis wzorca:

- Cel pobieżnie opisuje zarówno problem, jak i rozwiązanie.
- Motywacja rozszerza opis problemu i rozwiązania jakie umożliwia dany wzorzec.
- Struktura klas ukazuje poszczególne części wzorca i jak są ze sobą powiązane.
- Przykład kodu w którymś z popularnych języków programowania pomaga zrozumieć ideę wzorca.

Wzorce konstrukcyjne

Wzorce konstrukcyjne są grupą, która opisuje sposoby tworzenia obiektów. Obiekty możemy tworzyć za pomocą 'konstruktorów' (metoda __init__ w Pythonie), ale ograniczając się tylko do tej metody, w przypadku bardziej skomplikowanych obiektów, nasz kod może stać się bardzo nieczytelny.

W tej grupie wyróżniamy następujące wzorce:

- Singleton - opisuje, w jaki sposób stworzyć pojedynczą instancję obiektu w aplikacji.
- Builder - wykorzystywany do tworzenia skomplikowanych obiektów.
- Factory Method - upraszcza tworzenie skomplikowanych obiektów tej samej grupy obiektów.
- Abstract Factory - umożliwia tworzenie obiektów konkretnej rodziny (jednej wśród wielu) powiązanych ze sobą obiektów.
- Prototype - opisuje jak tworzyć obiekt, za pomocą kopii już istniejącego obiektu bazowego.

Singleton

Wzorzec projektowy Singleton to wzorzec, który zapewnia, że klasa ma tylko jedną instancję oraz udostępnia globalny punkt dostępu do tej instancji. Ma to różne zastosowania, ale jest szczególnie przydatne, gdy potrzebujemy dokładnie jednego obiektu do koordynacji działań w całym systemie. Można to porównać do korzystania z jednej kamery nadzoru w sklepie - nawet jeśli sklep ma wiele drzwi, używana jest tylko jedna kamera do monitorowania wejścia i wyjścia, zapewniając centralny punkt widzenia.

Przykład w Pythonie, który przygotowałem, demonstruje implementację Singletona za pomocą dekoratora. Dekorator Singleton modyfikuje klasę tak, aby zamiast tworzyć nowe

instancje za każdym razem, kiedy jest wywoływana, zwracała tę samą instancję. W naszym przykładzie, klasy FirstClass i SecondClass są "dekorowane" jako Singletony. To oznacza, że bez względu na to, ile razy próbujemy utworzyć instancję tych klas, faktycznie otrzymujemy za każdym razem tę samą instancję. W skrypcie, a i b są instancjami FirstClass, a c i d - SecondClass. Mimo wielokrotnego tworzenia, faktycznie otrzymujemy za każdym razem te same instancje, co demonstruje, jak a.val zmienia się na 10 i c.val na 20, co wpływa na wszystkie odwołania do tych instancji.

Takie podejście może być użyteczne w różnych scenariuszach, na przykład przy zarządzaniu połączeniem z bazą danych. Chcemy mieć tylko jedno połączenie, które jest używane przez różne części aplikacji, aby uniknąć niepotrzebnej nadmiarowości i obciążenia. Implementując bazę danych jako Singleton, zapewniamy, że każdy komponent aplikacji współdzieli to samo połączenie, co może poprawić wydajność i ułatwić zarządzanie zasobami.

Builder

Wzorzec projektowy Builder jest używany, gdy konstrukcja obiektu jest złożona i wymaga wielu kroków. Pozwala on oddzielić konstrukcję złożonego obiektu od jego reprezentacji, tak aby ten sam proces konstrukcyjny mógł tworzyć różne reprezentacje. W praktyce może to być przyrównane do budowy domu, gdzie proces budowy (budowa fundamentów, wznoszenie ścian, montaż dachu) pozostaje taki sam niezależnie od tego, czy budujemy dom jednorodzinny, czy bliźniak. Plan budowy dostosowuje się do potrzeb klienta, ale proces konstrukcyjny jest znormalizowany.

Przykład, który przedstawiłem, demonstruje implementację wzorca Builder w kontekście składania komputerów. Mamy klasę Computer, która reprezentuje produkt końcowy, tutaj komputer z różnymi komponentami. ComputerBuilder jest abstrakcyjnym interfejsem (w Pythonie realizowanym przez klasę z metodą build(), która jest zaznaczona do implementacji w klasach pochodnych) do tworzenia produktów, a GamingComputerBuilder i OfficeComputerBuilder to konkretne implementacje budowniczych, każdy składający komputer o różnych specyfikacjach - odpowiednio do gier i biura.

Klasa ComputerShop działa jako "director", kierując procesem budowy i używając odpowiedniego obiektu Builder do stworzenia komputera zgodnie z zapotrzebowaniem klienta. Ten wzorzec jest szczególnie użyteczny, gdy mamy do czynienia z obiektami, które mogą być skonfigurowane w wielu różnych wariantach, zachowując przy tym czystość kodu i separację odpowiedzialności.

Użycie tego wzorca umożliwia łatwą rozbudowę o nowe typy komputerów (np. komputer do edycji wideo, serwer itp.) poprzez dodanie nowych ConcreteBuilders bez potrzeby modyfikacji istniejącego kodu klas Computer i ComputerShop, co jest zgodne z zasadą otwarte-zamknięte z SOLID.

Factory Method

Wzorzec projektowy Factory Method (Metoda Fabryczna) jest stosowany, gdy w programie mamy do czynienia z wieloma obiektami tej samej klasy bazowej, ale różniących się typem. Pozwala on na delegowanie odpowiedzialności za tworzenie instancji obiektów do różnych "fabryk" specjalizujących się w produkcji tych obiektów. Można to przyrównać do fabryki zabawek, która ma różne linie produkcyjne dla różnych rodzajów zabawek. Każda linia produkuje zabawki według własnego specyficznego procesu, ale wszystkie one są rozpoznawalne jako produkty tej samej fabryki.

Przykład, który przygotowałem, ilustruje implementację wzorca Factory Method w kontekście tworzenia gier. Mamy abstrakcyjną klasę bazową Game, która definiuje wspólne cechy i zachowania gier. Od niej dziedziczą klasy BoardGame i PCGame, każda reprezentująca różne rodzaje gier.

GameFactory to klasa bazowa (interfejs) dla fabryk, które będą tworzyć konkretne gry. MonopolyGameCreator i ValorantGameCreator to konkretne implementacje tej fabryki, odpowiedzialne za tworzenie konkretnych instancji gier: Monopoly (gra planszowa) i Valorant (gra komputerowa). Dzięki temu, kiedy użytkownik wybierze typ gry, odpowiednia fabryka zostanie użyta do stworzenia obiektu tej gry bez konieczności bezpośredniego odwoływania się do konstruktorów klas gier w kodzie klienta.

Wzorzec Factory Method ułatwia rozbudowę aplikacji o nowe typy produktów (w tym przypadku gier), ponieważ dodanie nowej gry wymaga tylko utworzenia nowej klasy konkretnego produktu i odpowiadającej jej klasy fabryki, bez konieczności modyfikowania istniejącego kodu klienta. To podejście sprzyja zasadzie otwarte/zamknięte z SOLID, zgodnie z którą klasy powinny być otwarte na rozszerzenia, ale zamknięte na modyfikacje.

Abstract Factory

Wzorzec projektowy Abstract Factory (Fabryka Abstrakcyjna) umożliwia tworzenie rodzin powiązanych ze sobą obiektów bez określania ich konkretnych klas. Jest to rozwinięcie wzorca Factory Method, które pozwala na działanie na wielu fabrykach. Można to porównać do zarządzania różnymi fabrykami produkującymi różne rodzaje pojazdów, gdzie każda fabryka może produkować samochody, rowery, motocykle itp., ale wszystkie produkty z danej fabryki pasują do siebie pod względem stylu, materiałów itd.

W przedstawionym przykładzie mamy do czynienia z fabrykami samochodów różnych marek, gdzie każda fabryka może produkować różne warianty tego samego modelu (Combi, Hatchback, Sedan). CarFactory jest klasą bazową (interfejsem) dla wszystkich fabryk samochodów, definiując metody, które muszą być zaimplementowane przez konkretne fabryki, np. ToyotaCorollaFactory i AudiA4Factory. Te konkretne fabryki są odpowiedzialne za tworzenie określonych wariantów modeli samochodów. Dzięki temu, niezależnie od tego, którą fabrykę wybierzemy, możemy być pewni, że wszystkie samochody będą miały wspólne cechy charakterystyczne dla danej marki.

FactoryProvider jest klasą pomocniczą, która pozwala na wybór odpowiedniej fabryki na podstawie wejścia użytkownika. W ten sposób użytkownik decyduje, z jakiej "fabryki" chce skorzystać, a następnie za pomocą tej fabryki tworzone są konkretnie obiekty (w tym przypadku hatchbacki), bez bezpośredniego odwoływania się do konstruktorów klas produktów.

Wzorzec Abstract Factory jest szczególnie przydatny, gdy systemy powinny być niezależne od sposobu tworzenia, komponowania i reprezentacji produktów, które produkują. Pozwala to na wprowadzanie nowych wariantów produktów (np. nowych modeli samochodów) bez zmiany kodu klienta, co jest zgodne z zasadą otwarte/zamknięte z SOLID. Dzięki temu wzorcowi możemy łatwo rozszerzać nasz system o nowe fabryki i produkty, utrzymując przy tym porządek i modularność kodu.

Wzorce projektowe Builder, Factory Method i Abstract Factory faktycznie rozwiązują podobne problemy i często są używane do tworzenia obiektów w oprogramowaniu. Jednak mają one różne cele i są stosowane w różnych scenariuszach. Oto główne różnice między nimi oraz wskazówki, kiedy który wzorzec może być najbardziej odpowiedni:

Builder

Cel: Oddziela konstrukcję złożonego obiektu od jego reprezentacji, tak aby ten sam proces konstrukcyjny mógł tworzyć różne reprezentacje.

Kiedy używać:

- Gdy tworzenie obiektu wymaga wielu kroków, które mogą być realizowane w różnych kombinacjach lub sekwencjach.
- Gdy konfiguracja obiektu musi być bardzo elastyczna i istnieje wiele opcji konfiguracji.

Przykład: Tworzenie złożonego obiektu `Dokument`, który może mieć różne formaty, nagłówki, stopki itd.

Factory Method

Cel: Definiuje interfejs do tworzenia obiektu, ale pozwala klasom pochodnym decydować, która klasa ma być instancjonowana. Factory Method pozwala klasie przekazać proces tworzenia instancji do klas pochodnych.

Kiedy używać:

- Gdy klasa nie może przewidzieć klasy obiektów, które ma tworzyć.
- Gdy klasa chce, aby jej podklasy określały tworzone obiekty.

- Gdy klasy delegują odpowiedzialność do jednej z wielu pomocniczych podklas, a lokalizacja tej wiedzy o tym, która klasa pomocnicza jest delegatem, jest istotna.

Przykład: Klasa `Dokument` może wymagać tworzenia różnych typów dokumentów, takich jak `DokumentTekstowy` lub `DokumentGraficzny`, ale dokładny typ tworzonego dokumentu jest znany tylko w czasie wykonania.

Abstract Factory

Cel: Zapewnia interfejs do tworzenia rodzin powiązanych lub zależnych obiektów bez określania ich konkretnych klas.

Kiedy używać:

- Gdy system powinien być niezależny od sposobu tworzenia, komponowania i reprezentacji swoich produktów.
- Gdy system ma wiele rodzin produktów do obsłużenia i chce je konfigurować w czasie wykonania.
- Gdy rodzina produktów jest zaprojektowana tak, aby razem współpracować, a wymagasz tej gwarancji w czasie kompilacji.

Przykład: Tworzenie interfejsu użytkownika, gdzie każdy element (przyciski, okna dialogowe) może mieć różne style w zależności od systemu operacyjnego (Windows, MacOS, Linux) i wszystkie elementy jednego GUI muszą być spójne.

Podsumowanie:

- Builder skupia się na konstruowaniu złożonych obiektów krok po kroku.
- Factory Method tworzy jedną instancję konkretnej klasy, ale dokładna klasa jest ukryta przed klientem.
- Abstract Factory tworzy rodziny powiązanych obiektów bez określania ich konkretnych klas.

Wybór pomiędzy tymi wzorcami zależy głównie od problemu, który próbujesz rozwiązać, oraz od złożoności obiektów, które tworzysz.

Prototype

Wzorzec projektowy Prototype umożliwia kopiowanie istniejących obiektów bez konieczności zależności od ich klas. Jest to szczególnie przydatne, gdy tworzenie kopii obiektu jest bardziej efektywne lub wygodne niż tworzenie nowego od zera, zwłaszcza gdy mamy do

czynienia z obiektami o złożonym stanie lub konfiguracji. Można to przyrównać do sytuacji, gdy zamiast hodować nową owcę od narodzin, bierzemy istniejącą owcę i "kopiujemy" ją, aby uzyskać nową o identycznych cechach, ale z możliwością wprowadzenia modyfikacji.

Przykład, który przedstawiłem, ilustruje użycie wzorca Prototype za pomocą klasy Sheep. Klasa ta ma metodę clone, która wykorzystuje funkcję deepcopy z modułu copy do utworzenia głębokiej kopii obiektu, co oznacza, że każdy obiekt zagnieżdżony wewnątrz zostanie również skopiowany. Dzięki temu, gdy modyfikujemy sklonowany obiekt (na przykład zmieniając imię owcy z "Jolly" na "Dolly"), oryginalny obiekt pozostaje niezmieniony.

W przykładzie najpierw tworzymy instancję Sheep o imieniu "Jolly". Następnie wykorzystujemy metodę clone do stworzenia jej kopii, którą modyfikujemy, zmieniając jej imię na "Dolly". Na końcu sprawdzamy, że oryginalna owca i jej klon to dwa różne obiekty, co pokazuje, że klonowanie zostało wykonane poprawnie.

Wzorzec Prototype jest szczególnie użyteczny w aplikacjach wymagających tworzenia kopii obiektów z dynamicznie zmieniającym się stanem lub w systemach, gdzie koszt inicjalizacji instancji jest wysoki. Ułatwia on zarządzanie stanem obiektów i pozwala na elastyczne modyfikowanie sklonowanych instancji, co sprzyja lepszej organizacji kodu i może przyczynić się do zwiększenia wydajności aplikacji.

Wzorce strukturalne

Wzorce strukturalne to jedne z kategorii wzorców projektowych w programowaniu. Pomagają one określić, w jaki sposób różne obiekty współpracują ze sobą, tworząc struktury. Głównym celem wzorców strukturalnych jest ułatwienie projektowania przez zapewnienie bardziej elastycznych i efektywnych sposobów łączenia obiektów.

W prostych słowach, można to porównać do klocków LEGO. Wyobraź sobie, że masz różne kształty klocków, ale niektóre z nich nie pasują do siebie bezpośrednio. Wzorce strukturalne to takie "adaptery" lub "mosty", które pozwalają ci połączyć te klocki w sposób, który wcześniej byłby niemożliwy.

Przykłady wzorców strukturalnych to m.in.:

- Adapter (Adapter) - umożliwia współpracę obiektów o niekompatybilnych interfejsach.
- Most (Bridge) - rozdziela abstrakcję od jej implementacji, tak aby obie mogły być modyfikowane niezależnie.
- Kompozyt (Composite) - pozwala traktować pojedyncze obiekty i ich zgrupowania w ten sam sposób.
- Dekorator (Decorator) - dodaje nowe funkcjonalności do obiektów dynamicznie, bez modyfikacji ich kodu.

Adapter

Wzorzec projektowy Adapter jest jak tłumacz między dwoma osobami, które nie mówią tym samym językiem. Pozwala jednemu obiektowi (w naszym przykładzie NewPrinter), który ma pewną funkcjonalność, komunikować się i współpracować z innym obiektem (OldPrinter), który oczekuje innej funkcjonalności, bez konieczności zmiany kodu obu klas. Adapter "przetłumaczy" więc wezwania z jednego interfejsu na drugi.

Zastosowania wzorca Adapter:
Przykład z życia: Możemy porównać to do sytuacji, gdzie masz urządzenie elektryczne z innego kraju, które wymaga innego typu wtyczki niż te dostępne w twoim domu. Zamiast wymieniać wtyczkę urządzenia lub gniazdka, używasz adaptera, który umożliwia podłączenie urządzenia do istniejącego gniazdka. Adapter nie zmienia niczego w urządzeniu ani w gniazdku; po prostu umożliwia im współpracę.

W tym kodzie PrinterAdapter dziedziczy po OldPrinter, ale zamiast używać oryginalnej implementacji metody show, przekierowuje wywołanie do metody display z NewPrinter. Dzięki temu, gdy stara część aplikacji próbuje użyć metody show na adapterze, rzeczywiste działanie odbywa się poprzez metodę display nowej drukarki. Adapter umożliwia więc współpracę między starym a nowym kodem bez konieczności ich bezpośredniej modyfikacji.

Decorator

Wzorzec projektowy Dekorator pozwala na dynamiczne dodawanie nowych funkcjonalności do obiektów bez zmiany ich struktury. To trochę jak ubieranie choinki w święta: startujesz z podstawową choinką i dodajesz do niej ozdoby, każda ozdoba zmienia wygląd choinki, ale sama choinka pozostaje choinką. Dekorator "ubiera" nasz obiekt w dodatkowe funkcjonalności.

Zastosowania wzorca Dekorator:
Przykład z życia: Możemy porównać wzorzec Dekorator do wyboru dodatków do pizzy. Zaczynasz od podstawowej pizzy, a następnie dodajesz składniki, takie jak pepperoni, pieczarki czy oliwki, w zależności od preferencji. Każdy dodatek zmienia smak i wygląd pizzy, ale wciąż jest to pizza.

Przykład z Pythona: Jak w przedstawionym kodzie, zaczynasz od prostej kawy. Możesz następnie "udekorować" ją dodatkami, takimi jak mleko, karmel czy bita śmietana, dodając do niej nowe właściwości (koszt i opis) bez zmiany oryginalnej klasy Coffee. Dzięki temu możesz tworzyć różne kombinacje kawy, używając tych samych klas.

Każdy dekorator "opakowuje" oryginalny obiekt kawy, dodając do niego nowe właściwości (koszt i opis dodatków). Wzorzec Dekorator umożliwia elastyczne i łatwe dodawanie nowych funkcjonalności do obiektów, co jest szczególnie przydatne w sytuacjach, gdy modyfikacje muszą być dokonywane dynamicznie, na etapie działania programu.

Facade

Wzorzec projektowy Fasada to jak pilot do domowego centrum rozrywki. Zamiast wstawać i manualnie włączać telewizor, odtwarzacz DVD i zmniejszać natężenie światła, używasz pilota (fasady), aby zrobić to wszystko jednym kliknięciem. Fasada ukrywa skomplikowane interakcje między różnymi urządzeniami i oferuje prosty interfejs do obsługi całego zestawu.

Zastosowania wzorca Fasada:
Przykład z życia: Używanie pilota do zarządzania wszystkimi urządzeniami w salonie. Pilot działa jak fasada, która umożliwia prostą interakcję z złożonym systemem urządzeń bez konieczności wchodzenia w szczegóły ich działania.

Wzorzec Fasada jest szczególnie przydatny w systemach, gdzie istnieje wiele zależnych lub skomplikowanych klas, a my chcemy zapewnić prosty sposób na ich obsługę. Umożliwia to klientom korzystanie z systemu na wysokim poziomie abstrakcji, bez konieczności zagłębiania się w szczegóły implementacji.

Proxy

Wzorzec projektowy Proxy działa jak dozorca, który kontroluje dostęp do jakiegoś obiektu, opóźnia jego tworzenie lub wykonuje dodatkowe operacje przy dostępie do tego obiektu, na przykład logowanie. To trochę jak mieć ochroniarza przed drzwiami klubu, który decyduje, kto może wejść, i zajmuje się wszystkimi sprawami, zanim pozwoli Ci porozmawiać z właściwą osobą wewnątrz.

Zastosowania wzorca Proxy:
Przykład z życia: Można to porównać do korzystania z karty płatniczej podczas zakupów. Karta działa jako pośrednik (proxy) między Tobą a Twoim kontem bankowym; pozwala na dokonanie płatności bez konieczności fizycznego posiadania gotówki. Operacje sprawdzające, autoryzacja i finalizacja transakcji są odroczone do momentu, gdy są rzeczywiście potrzebne.

Przykład z Pythona: W podanym kodzie, zamiast ładować obraz z dysku w momencie tworzenia obiektu RealImage, co może być czasochłonne, używamy ProxyImage do opóźnienia ładowania. Obraz jest ładowany tylko wtedy, gdy jest to konieczne (np. przy pierwszym wyświetleniu). Dzięki temu, jeśli obraz nie jest w ogóle potrzebny, nigdy nie dochodzi do kosztownej operacji ładowania.

Wzorzec Proxy jest przydatny, gdy chcemy kontrolować dostęp do pewnych zasobów lub operacji. Może być używany do zarządzania kosztami związanymi z tworzeniem obiektów, do kontroli dostępu do nich, do ładowania obiektów na żądanie lub do dodawania logiki (np. logowania, liczenia odwołań) przy dostępie do obiektów.

Bridge

Wzorzec projektowy Most (Bridge) pozwala rozdzielić duży klasę lub grupę powiązanych klas na dwie oddzielne hierarchie: abstrakcję i implementację, które mogą być rozwijane niezależnie od siebie. To trochę jak mieć zdalne sterowanie (abstrakcję), które może obsługiwać różne urządzenia (implementacje), takie jak telewizor czy radio, bez konieczności wiedzy o szczegółach działania tych urządzeń.

Zastosowania wzorca Most:
Przykład z życia: Pomyśl o uniwersalnym pilocie do telewizora, który może sterować różnymi markami telewizorów. Pilot jest zaprojektowany do wykonywania ogólnych działań (włącz/wyłącz, zmiana kanału), niezależnie od specyfiki danego modelu telewizora. Model telewizora (implementacja) może być zmieniany, ale sposób, w jaki używasz pilota (abstrakcja), pozostaje taki sam.

Przykład z Pythona: W podanym kodzie, klasa RemoteControl działa jako abstrakcja, zapewniając ogólne działanie (w tym przypadku włączanie i wyłączanie), które można zastosować do dowolnego urządzenia. Klasy TV i Radio są konkretnymi implementacjami, które definiują, co dokładnie oznacza włączenie lub wyłączenie dla danego urządzenia. Wzorzec Most pozwala na niezależne rozwijanie klas urządzeń i klas abstrakcji.

Kluczowym elementem tutaj jest rozdzielenie "co" ma być zrobione (tj. włączanie/wyłączanie urządzenia) od "jak" to ma być zrobione (tj. specyficzna implementacja dla TV lub radia). Dzięki temu możesz łatwo dodawać nowe urządzenia do systemu, tworząc kolejne klasy implementacji, bez konieczności zmiany kodu abstrakcji. Jest to szczególnie przydatne w dużych systemach, gdzie zmiany w jednym miejscu mogą mieć niepożądane skutki w innym.

Composite

Wzorzec projektowy Kompozyt pozwala na traktowanie pojedynczych obiektów i złożonych struktur obiektów w ten sam sposób. To trochę jak patrzeć na drzewo genealogiczne: możesz spojrzeć na pojedynczą osobę (liść) lub całą rodzinę (gałąź z liśćmi), ale sposób, w jaki interpretujesz ich informacje, jest podobny. Kompozyt umożliwia zarządzanie grupami obiektów tak, jakby były one pojedynczym obiektem.

Zastosowania wzorca Kompozyt:
Przykład z życia: Możesz pomyśleć o organizacji firmy, gdzie masz pracowników (np. programistów) i menedżerów. Menedżerowie mogą mieć pod sobą innych pracowników lub menedżerów. Używając wzorca Kompozyt, możesz traktować każdego pracownika, niezależnie od ich pozycji, w jednolity sposób - na przykład, wyświetlając ich strukturę organizacyjną.

Przykład z Pythona: W podanym kodzie, klasa Manager może zawierać inne obiekty Employee (zarówno Developer jak i inne obiekty Manager z ich podwładnymi). Dzięki temu możesz wywołać metodę show_details() na obiekcie Manager, aby wyświetlić informacje zarówno o menedżerze, jak i o wszystkich jego podwładnych, niezależnie od ich poziomu w hierarchii.

Wzorzec Kompozyt jest szczególnie przydatny w scenariuszach, gdzie masz do czynienia z hierarchicznymi strukturami, takimi jak drzewa decyzyjne, struktury organizacyjne w firmach, systemy plików i inne. Umożliwia on jednolite i łatwe zarządzanie oraz dostęp do skomplikowanych struktur obiektów.

Flyweight

Wzorzec projektowy Flyweight ma na celu zmniejszenie zużycia pamięci przez współdzielenie jak największej ilości danych między podobnymi obiektami. Jest to szczególnie przydatne, gdy masz do czynienia z dużą liczbą obiektów o podobnym stanie. Można to porównać do systemu transportu publicznego, gdzie jedno pojazd (np. autobus) jest współdzielone przez wielu pasażerów, zamiast aby każdy z nich kierował własnym pojazdem do tego samego miejsca.

Zastosowania wzorca Flyweight:
Przykład z życia: Biblioteka w mieście może mieć tylko jedną kopię każdej unikalnej książki, którą może wypożyczać wielu czytelnikom. Zamiast kupować nową kopię książki dla każdej osoby, książka jest współdzielona, co oszczędza miejsce i pieniądze.

Przykład z Pythona: W przedstawionym kodzie, zamiast tworzyć nowy obiekt Circle dla każdego wystąpienia okręgu o tym samym kolorze, fabryka ShapeFactory sprawdza, czy okrąg o danym kolorze już istnieje. Jeśli tak, to ten istniejący okrąg jest używany ponownie. Dzięki temu, jeśli stworzysz tysiące okręgów o ograniczonej liczbie kolorów, rzeczywiście będziesz miał tylko jedną instancję dla każdego unikalnego koloru.

Tutaj, ShapeFactory działa jako centralny punkt, w którym obiekty Circle są tworzone i zarządzane. Kiedy program prosi o okrąg o danym kolorze, fabryka najpierw sprawdza, czy taki okrąg już istnieje. Jeśli tak, to zwraca referencję do istniejącego obiektu zamiast tworzyć nowy, co oszczędza zasoby systemowe.

W Pythonie dekorator @classmethod jest używany do oznaczania metody jako metody klasy. Oznacza to, że metoda jest powiązana z klasą, a nie z instancją klasy. Główna różnica pomiędzy zwykłą metodą (czyli metodą instancji) a metodą klasy polega na tym, do czego odnoszą się one za pomocą pierwszego argumentu: zwykła metoda przyjmuje instancję jako pierwszy argument (self), podczas gdy metoda klasy przyjmuje klasę jako pierwszy argument (cls).

Metody instancji: To są standardowe metody, które mają dostęp do konkretnej instancji (przy użyciu `self`) i jej atrybutów. Muszą być wywołane na rzecz instancji klasy.

```
def metoda_instancji(self, arg1, arg2, ...):
    ...
```

Metody klasy (`@classmethod`): Są powiązane z klasą, a nie z konkretną instancją. Przyjmują jako pierwszy argument klasę (`cls`) zamiast instancji. Mogą być wywołane zarówno na rzecz klasy, jak i jej instancji.

```
@classmethod
def metoda_klasy(cls, arg1, arg2, ...):
    ...
```

Metody statyczne (`@staticmethod`): Nie mają dostępu ani do instancji (`self`), ani do klasy (`cls`). Działają jak zwykłe funkcje, ale są definiowane w obrębie klasy i można je wywołać zarówno na rzecz klasy, jak i jej instancji. Metody statyczne nie mają specjalnych argumentów związanych z klasą lub instancją.

```
@staticmethod
def metoda_statyczna(arg1, arg2, ...):
    ...
```

- Argumenty:
  - `@classmethod` przyjmuje jako pierwszy argument klasę (zwykle nazywaną `cls`).
  - `@staticmethod` nie przyjmuje specjalnych pierwszych argumentów związanych z instancją lub klasą.
- Dostęp:
  - Metody klasy mają dostęp do atrybutów klasy i mogą je modyfikować.
  - Metody statyczne nie mają bezpośredniego dostępu do atrybutów klasy ani instancji.
- Zastosowanie:
  - `@classmethod` jest często używany do tworzenia metod fabrycznych lub alternatywnych konstruktorów.
  - `@staticmethod` jest używany, gdy metoda nie wymaga dostępu do atrybutów instancji ani klasy, ale ma sens, aby była związana z klasą (np. funkcje pomocnicze).

W praktyce wybór między `@classmethod` a `@staticmethod` zależy od konkretnych potrzeb. Jeśli metoda powinna interakcjonować z atrybutami klasy, używaj `@classmethod`. Jeśli metoda działa niezależnie od atrybutów klasy i instancji, ale ma sens, aby była częścią klasy, używaj `@staticmethod`.

Wzorce behawioralne

Wzorce behawioralne to jedna z trzech głównych kategorii wzorców projektowych obok wzorców strukturalnych i kreacyjnych. Skupiają się one na algorytmach oraz przydziale odpowiedzialności między obiektami.

Wzorce behawioralne opisują sposoby komunikacji pomiędzy obiektami, czyli jak obiekty współdziałają i jak dzielą się odpowiedzialnością. Wzorce te są szczególnie przydatne, gdy chodzi o złożone przepływy kontroli i decyzji, które są trudne do przewidzenia w wyniku dynamicznych interakcji pomiędzy obiektami.

Chain of responsibility

Wzorzec projektowy Chain of Responsibility (łańcuch odpowiedzialności) pozwala uniknąć sprzęgania nadawcy żądania z jego odbiorcami, dając szansę na obsłużenie żądania więcej niż jednemu obiektowi. Wyobraź sobie, że jest to jak gra w "gorącego ziemniaka", gdzie zadanie lub prośba jest przekazywana od jednej osoby do drugiej, aż znajdzie się ktoś, kto może się nią zająć.

Zastosowania wzorca Chain of Responsibility:
Przykład z życia: Procedura składania reklamacji w sklepie może być przykładem Chain of Responsibility. Najpierw składasz reklamację sprzedawcy, jeśli sprzedawca nie może jej rozwiązać, jest przekazywana do menedżera, a następnie, jeśli nadal pozostaje nierozwiązana, do działu obsługi klienta centrali. Każdy poziom ma szansę rozwiązać problem przed przekazaniem go dalej.

Przykład z Pythona: W przedstawionym kodzie mamy prosty przykład łańcucha odpowiedzialności. Mamy dwa konkretne obiekty obsługi (ConcreteHandlerA i ConcreteHandlerB), z których każdy obsługuje różne typy żądań: ConcreteHandlerA obsługuje liczby parzyste, a ConcreteHandlerB liczby nieparzyste. Jeśli żaden z nich nie może obsłużyć żądania, przekazuje je dalej w łańcuchu.

W tym kodzie, żądania są przekazywane wzdłuż łańcucha - najpierw do ConcreteHandlerA, który obsługuje liczby parzyste. Jeśli liczba jest nieparzysta, przekazuje ją do ConcreteHandlerB. To pokazuje elastyczność wzorca w obsłudze różnych rodzajów żądań przy minimalnym sprzężeniu między nadawcą a odbiorcami.

Template Method

Wzorzec Template Method definiuje szkielet algorytmu w metodzie szablonowej w superklasie, ale pozwala podklasom nadpisać określone etapy algorytmu bez zmiany jego struktury. To trochę jak przepis kulinarny, który podaje kroki przygotowania dania,

pozostawiając miejsce na własną interpretację niektórych kroków, np. dodawanie składników według własnego gustu.

Zastosowania wzorca Template Method:
Przykład z życia: Przygotowanie napoju - podobnie jak w podanym kodzie. Masz ogólny przepis na przygotowanie napoju (gotowanie wody, zaparzanie, nalewanie do filiżanki, dodawanie dodatków), ale sposób zaparzania i dodatki mogą się różnić w zależności od tego, czy przygotowujesz herbatę, czy kawę.

Przykład z Pythona: Jak w przedstawionym kodzie, Beverage to klasa bazowa, która definiuje szkielet algorytmu przygotowania napoju. Metody brew_or_steep() i add_condiments() są zdefiniowane jako puste (do nadpisania przez podklasy), ponieważ różne napoje wymagają różnych metod parzenia i dodatków. Klasy Tea i Coffee nadpisują te metody, dostarczając szczegółową implementację dla każdego napoju.

Wzorzec Template Method jest użyteczny, gdy mamy do czynienia z algorytmami, które mają zdefiniowaną sztywną strukturę kroków, ale niektóre z tych kroków mogą być realizowane na różne sposoby w zależności od kontekstu. Umożliwia to podklasom precyzyjne dostosowanie pewnych aspektów algorytmu, zachowując jego ogólną strukturę.

Memento

Wzorzec projektowy Memento pozwala zapisać i przywrócić poprzedni stan obiektu bez ujawniania szczegółów jego implementacji. To trochę jak funkcja "cofnij" w edytorze tekstu, która pozwala wrócić do poprzedniego stanu dokumentu.

Prosty przykład wzorca Memento:
Załóżmy, że mamy prosty edytor tekstu, który pozwala zapisywać i przywracać swoje stany (tekst).

EditorMemento: Klasa Memento przechowuje stan obiektu Editor. Stan ten jest niezmienialny po utworzeniu memento.
Editor: To klasa, której stan chcemy zapisywać i do której chcemy móc powrócić. Posiada metody type() do dodawania tekstu, save() do zapisywania obecnego stanu w memento, oraz restore() do przywracania stanu z memento.
W przykładowym użyciu, obiekt Editor jest używany do pisania tekstu. Stan edytora jest zapisywany przed dodaniem trzeciego zdania. Następnie tekst jest przywracany do ostatniego zapisanego stanu, co skutkuje usunięciem "Trzeciego zdania." z zawartości.
Wzorzec Memento jest przydatny, gdy potrzebujesz oferować funkcje cofania operacji lub zapisywania stanów w aplikacjach, zapewniając przy tym wysoki poziom enkapsulacji.

State

Wzorzec State pozwala obiektowi zmieniać swoje zachowanie w zależności od wewnętrznego stanu. Jest to trochę jak zmiana kanałów w telewizorze: w zależności od tego, jaki kanał jest wybrany, telewizor pokazuje różne treści. W kontekście wzorca State, "kanały" to różne stany, które mogą zmieniać sposób, w jaki obiekt reaguje na zewnętrzne bodźce.

Zastosowania wzorca State:
Przykład z życia: Światła uliczne działające na zasadzie wzorca State. Światło zmienia się z czerwonego na zielone, na żółte, a następnie z powrotem na czerwone, gdzie każdy kolor reprezentuje różny stan i zachęca do różnych działań (stop, go, prepare to stop).

Przykład z Pythona: Jak w przedstawionym kodzie, mamy system świateł drogowych, gdzie TrafficLight zmienia swój stan między RedLight, GreenLight, i YellowLight. Każdy stan wykonuje różne działanie i przygotowuje system do przejścia do następnego stanu.

Wzorzec State jest przydatny, kiedy masz do czynienia z obiektami, których zachowanie musi zmieniać się dynamicznie w zależności od ich wewnętrznego stanu. Umożliwia to płynne przejścia między różnymi stanami bez potrzeby zmiany warunków wewnątrz metod obiektu, zachowując zarazem czystość kodu i łatwość zarządzania różnymi stanami.

Command

Wzorzec projektowy Command pozwala zamknąć wszystkie informacje potrzebne do wykonania danej akcji lub wywołania zdarzenia w jednym obiekcie polecenia. Ten wzorzec oddziela obiekt wykonujący operację od obiektu, który wie, jak operacja ma być wykonana. To trochę jak używanie pilota do telewizora: przycisk na pilocie (invoker) wywołuje określoną funkcję na telewizorze (receiver), ale nie musisz wiedzieć, jak ta funkcja jest wewnątrz telewizora implementowana.

Zastosowania wzorca Command:
Przykład z życia: Zamówienie w restauracji. Klient (invoker) składa zamówienie (command) u kelnera, który przekazuje je do kuchni (receiver), gdzie jest realizowane. Klient nie musi wiedzieć, jak przygotować danie - to zadanie kuchni.

Przykład z Pythona: W przedstawionym kodzie, mamy prosty system zarządzania światłem, gdzie Light to klasa reprezentująca światło (receiver), a LightOnCommand i LightOffCommand to konkretne polecenia, które mogą być wywołane przez RemoteControl (invoker). W ten sposób RemoteControl może włączyć lub wyłączyć światło bez bezpośredniego odwoływania się do jego implementacji.

Wzorzec Command jest szczególnie przydatny, gdy potrzebujesz oddzielić obiekty wykonujące operacje od obiektów decydujących, kiedy te operacje mają być wykonane. Umożliwia to większą elastyczność w zarządzaniu operacjami, ich kolejkowaniu, logowaniu czy nawet wspieraniu cofania operacji.

Visitor

Wzorzec projektowy Visitor pozwala na dodanie nowych operacji do istniejących klas obiektów bez ich modyfikowania. Jest to przydatne, gdy mamy do czynienia z operacjami na obiektach różnych klas, ale chcemy uniknąć zanieczyszczania ich kodu dodatkowymi funkcjonalnościami. Można to porównać do wizyty w różnych miastach: zamiast każde miasto przygotowywało własny plan zwiedzania dla każdego turysty, turysta (visitor) przybywa z własnym planem (operacją) i realizuje go w każdym mieście (elemencie), które odwiedza.

Zastosowania wzorca Visitor:
Przykład z życia: Inspektor odwiedzający różne stacje pracy w fabryce. Każda stacja pracy (Element) ma swoje specyficzne zadania, ale inspektor (Visitor) ma własną agendę kontroli lub działań do przeprowadzenia na każdej stacji, niezależnie od jej specyfiki.

Przykład z Pythona: W przedstawionym kodzie mamy do czynienia z klasami ElementA i ElementB, które reprezentują różne elementy, które mogą być odwiedzane. ConcreteVisitor jest przykładem Visitora, który wykonuje różne akcje w zależności od typu elementu, który odwiedza. Dzięki temu wzorcowi, logika specyficzna dla typu elementu jest przeniesiona do klasy Visitora, a elementy mogą pozostać niezmienione.

W tym kodzie, metoda accept w klasach ElementA i ElementB pozwala ConcreteVisitor na wykonanie odpowiedniej operacji, w zależności od typu elementu. Jest to przykład na to, jak wzorzec Visitor pozwala na dodanie nowych operacji do klas bez ich modyfikacji, co jest szczególnie przydatne w aplikacjach, gdzie obiekty różnych klas muszą być przetwarzane w różny sposób.

Strategy

Wzorzec projektowy Strategy pozwala zdefiniować rodzinę algorytmów, umieszcza je w oddzielnych klasach i umożliwia ich wymienne używanie. Dzięki temu możesz zmieniać algorytmy niezależnie od klientów, którzy z nich korzystają. To jak wybór środka transportu do pracy: możesz iść pieszo, jechać rowerem, samochodem czy komunikacją miejską. Wybór zależy od wielu czynników, ale każda z tych opcji dostarcza Cię do celu, chociaż w różny sposób.

Zastosowania wzorca Strategy:
Przykład z życia: Planowanie trasy w aplikacji mapowej. Możesz wybrać najszybszą trasę, najkrótszą, unikając autostrad, lub trasę dla rowerów. Każda strategia prowadzi Cię do celu, ale korzysta z innej logiki.

Przykład z Pythona: W przedstawionym kodzie, mamy system obliczający podatek dla różnych krajów (USA, Polska, Niemcy), gdzie każdy kraj ma inną stawkę podatkową. Strategie podatkowe (USATaxStrategy, PolandTaxStrategy, GermanyTaxStrategy) są wymienialne i zależne od kontekstu zamówienia (Order), umożliwiając obliczenie całkowitej kwoty z zamówieniem w zależności od kraju.

Wzorzec Strategy jest użyteczny, gdy chcesz umożliwić użytkownikowi wybór z różnych algorytmów lub zachowań w czasie wykonania programu. Umożliwia to łatwe rozszerzanie i dodawanie nowych strategii bez modyfikowania kontekstu, w którym są używane, co jest kluczem do utrzymywania niskiego sprzężenia i wysokiej spójności w aplikacjach.

Mediator

Mediator to wzorzec projektowy, który ma na celu zredukowanie powiązań pomiędzy klasami poprzez przeniesienie tych powiązań do jednej klasy centralnej, zwaną mediatorem. Dzięki temu łatwiej jest modyfikować, utrzymywać i testować system.

Mediator definiuje interfejs dla komunikacji z obiektami kolegów (czyli obiektami komunikującymi się za pośrednictwem mediatora).

Poniżej znajduje się prosty przykład implementacji wzorca Mediator:

1. Definiujemy interfejs Mediator, który wymaga metody notify.
2. Następnie tworzymy konkretną klasę mediatora, ChatRoom, która reaguje na powiadomienia od kolegów (użytkowników w tym przypadku).
3. Klasa Colleague to klasa bazowa dla wszystkich kolegów, którzy będą komunikować się za pośrednictwem mediatora. W tym przypadku posiada metodę send_message, która powiadamia mediatora o wysłaniu wiadomości.
4. Klasa User to konkretny kolega, który wysyła wiadomości.

Główną ideą wzorca Mediator jest to, że zamiast wielu powiązań między różnymi klasami (kolegami), mamy jedno centralne miejsce (mediator), które zarządza komunikacją i zachowaniem systemu.

Stosowany w systemach komunikacji między komponentami, gdzie mediator zarządza interakcją, redukując bezpośrednie zależności między komponentami (np. w GUI, gdzie mediator zarządza komunikacją między panelami).

Observer

Wzorzec projektowy Observer pozwala obiektom na subskrybowanie zdarzeń innego obiektu i otrzymywanie powiadomień o tych zdarzeniach. Jest to jak otrzymywanie powiadomień na telefonie: aplikacje (obserwatorzy) nasłuchują na powiadomienia od systemu (podmiotu) i reagują na nie, gdy tylko się pojawią.

Zastosowania wzorca Observer:
Przykład z życia: Subskrypcja gazety. Jako subskrybent (obserwator) otrzymujesz nowe wydanie gazety (zdarzenie) zaraz po jego wydaniu przez wydawcę (podmiot).

Przykład z Pythona: W przedstawionym kodzie, TemperatureSensor działa jako podmiot (Subject), powiadamiając wszystkich swoich subskrybentów (obserwatorów), gdy temperatura się zmienia. TemperatureDisplay jest obserwatorem, który reaguje na zmiany temperatury, aktualizując wyświetlane informacje.

Wzorzec Observer jest szczególnie przydatny, gdy musisz utrzymać spójność między powiązanymi obiektami, ale nie chcesz ich ściśle wiązać. Dzięki temu, że obserwatorzy są powiadamiani o zmianach w podmiocie, mogą odpowiednio zareagować, utrzymując system zaktualizowany i zsynchronizowany.

Wzorce projektowe są schematami, które opisują najlepsze praktyki i dobre rozwiązania projektowe często napotykanych problemów w programowaniu obiektowym. Wzorce te można podzielić na trzy główne kategorie: strukturalne, behawioralne i kreacyjne. Oto podstawowe różnice między nimi:

Wzorce kreacyjne:

- Cel: Koncentrują się na procesach tworzenia obiektów.
- Zastosowanie: Są używane, gdy system musi być niezależny od sposobu tworzenia, komponowania i reprezentowania obiektów.
- Przykłady:
  - Singleton (Pojedynczy): Zapewnia, że klasa ma tylko jedną instancję i zapewnia punkt dostępu do niej.
  - Factory Method (Metoda Fabryczna): Definiuje interfejs do tworzenia obiektu, ale pozwala klasom pochodnym zdecydować, która klasa ma być instancjonowana.
  - Prototype (Prototyp): Tworzy obiekt na podstawie istniejącego obiektu przez klonowanie.
  - Builder (Budowniczy): Oddziela konstrukcję skomplikowanego obiektu od jego reprezentacji.
  - Abstract Factory (Fabryka Abstrakcyjna): Tworzy obiekt z kilku klas bazujących na rodzinach klas.

Wzorce strukturalne:

- Cel: Dotyczą kompozycji obiektów, czyli sposobu, w jaki obiekty są łączone, aby tworzyć większe struktury.
- Zastosowanie: Używane, gdy chcemy zdefiniować nowe sposoby łączenia obiektów lub jeśli chcemy zapewnić, by pewne klasy pracowały razem, chociaż ich interfejsy nie są kompatybilne.

- Przykłady:
    - Adapter (Adapter): Przystosowuje interfejs jednej klasy do interfejsu drugiej.
    - Bridge (Most): Oddziela abstrakcję od implementacji, dzięki czemu obie mogą ewoluować niezależnie.
    - Composite (Kompozyt): Pozwala traktować pojedyncze obiekty i ich kompozycje w sposób jednolity.
    - Decorator (Dekorator): Dodaje nowe odpowiedzialności do obiektu dynamicznie.
    - Facade (Fasada): Dostarcza uproszczony interfejs do grupy interfejsów w podsystemie.
    - Flyweight (Pylkowy): Wykorzystuje współdzielenie, aby wsparć efektywnie dużą liczbę obiektów o drobnych rozmiarach.
    - Proxy: Reprezentuje inny obiekt, kontrolując dostęp do niego.

Wzorce behawioralne:

- Cel: Dotyczą algorytmów i przydziału odpowiedzialności między obiektami.
- Zastosowanie: Skupiają się na komunikacji między obiektami i są używane, gdy chcemy zdefiniować skomplikowane przepływy kontroli i komunikacji między obiektami.
- Przykłady: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

Każda z tych kategorii wzorców rozwiązuje specyficzny zestaw problemów i koncentruje się na różnych aspektach projektowania oprogramowania. Wiedza o tych wzorcach i umiejętność ich stosowania może znacząco poprawić jakość i elastyczność projektowanego oprogramowania.

Dobre praktyki

Wprowadzenie

Czysty kod to termin wprowadzony przez Roberta C. Martina zwanego również jako Uncle Bob. Taki kod:

- Powinien być elegancki - czyli przede wszystkim łatwy do czytania i zrozumienia.

- Powinien być spójny - każda metoda, klasa czy moduł, powinien reprezentować to samo podejście do rozwiązania. Kod nie powinien być rozproszony i "zanieczyszczony" przez zależności i niepotrzebne detale.
- Powinien być zadbany. Programista tworzący czysty kod poświęcił sporo czasu, aby pozostał on uporządkowany i prosty.
- Przechodzi przez wszystkie testy.
- Nie zawiera duplikatów.
- Nie zawiera niepotrzebnych (nieużywanych) klas i metod.

Jak pisać czysty kod?

Samoświadome nazwy

Nazwy zmiennych, klas czy metod powinny wskazywać za co odpowiadają.

Nazwy zmiennych

```
d = 10 # czas, który upłynął liczony w dniach
```

Jeśli musisz dodawać komentarz, aby wyjaśnić do czego służy dana zmienna czy metoda, to znak, że robisz to źle.

Jak powinna nazywać się poprawnie taka zmienna? Np. tak:

```
elapsed_time_in_days = 10
```

Nazwy klas

Powinny składać się z rzeczownika lub frazy z rzeczownikiem, np.: `Account`, `Customer` czy `AddressParser`. Klasa w swojej nazwie nie powinna zawierać czasowników.

Nazwy metod

Powinny zawierać czasownik, np.: `post_payment`, `delete_page` czy `save`. Metody odpowiadające za pozyskanie czy zmianę wartości pola w klasie powinny mieć przedrostek `get` lub `set`.

Liczba argumentów

Metody powinny mieć niewielką liczbę argumentów. Jeśli potrzebujesz użyć dużej ilości argumentów, prawdopodobnie kod można podzielić na mniejsze części, stosując w tym celu pewne wzorce projektowe. Dobrym przykładem może być wzorzec Builder.

Długość metody

Ciało metody powinno być jak najkrótsze. Zazwyczaj, jeżeli ciało przekracza długość około 20 linii (lub umownie, nie mieści się na jednym ekranie), powinniśmy upewnić się, czy tworzona metoda nie zawiera zbyt dużej odpowiedzialności, tzn. nie robi czynności, które powinny wykonywać osobne metody, lub czy taką metodę można podzielić na kilka mniejszych.

Twórcy Pythona zdefiniowali zestaw zaleceń i dobrych praktyk pisania kodu tak, żeby był on czytelny i spójny. Dokumenty zawierające *Python Style Guide* to:

- `PEP 8` - dotyczący kodu źródłowego
- `PEP 257` - dotyczący dokumentowania kodu (tzw. `docstrings`)

Kod jest częściej czytany niż pisany, a dodatkowo zazwyczaj czytany **nie** przez autora, dlatego istotne jest trzymanie się ogólnie przyjętych zasad.

Konwencje nazewnicze

Każdy nietrywialny program zawiera w sobie wiele nazw różnych bytów: zmiennych, funkcji, klas itp. Kod powinien jak najlepiej się dokumentować, dlatego ważnym jest stosowanie znaczących nazw. Dodatkowo, zaleca się stosowanie następujących konwencji:

- funkcje - używamy *snake case* (czyli poszczególne słowa składają się z małych liter i połączone są ze sobą znakiem podkreślenia) - np. `execute`, `open_window`
- zmienne - "snake case" - np. `x`, `current_user`
- klasy - "camel case" (poszczególne słowa zaczynają się od wielkich liter i pisane są jednym ciągiem bez żadnych znaków łączących) - np. `Handler`, `MainWindow`
- metody klas - "snake case" - np. `undo`, `parse_answer`
- stałe - "SNAKE CASE" (jednak tym razem wyłącznie wielkie litery) - np. `TAU`, `MAX_VALUE`
- moduły - "snake case" - np. `parser.py`, `gsm_modem.py`
- pakiety - małe litery bez separatorów - np. `lte`, `supersort`

Układ kodu

Zaleca się, żeby kod Pythonowy miał *dużo przestrzeni* dla większego komfortu czytania - szczegółowe zasady:

- każda funkcja globalna i klasa powinna być otoczona dwoma pustymi liniami
- wszystkie metody klasy powinny być otoczone pojedynczymi pustymi liniami (nie licząc pierwszej)
- puste linie powinny rozdzielać logiczne kroki algorytmu
- w linii nie powinno być więcej niż 79 znaków, tak, żeby cała mieściła się na ekranie
- powinno się unikać `\` do dzielenia linii na krótsze

Wcięcia

Bloki kodu w Pythonie definiowane są wcięciami, więc nie można pozwolić sobie na zbyt dużą swobodę w formatowaniu kodu, ale warto trzymać się kilku zasad:

- spacje są preferowane nad tabulatory
- jeden poziom wcięcia to 4 spacje

- przy dzieleniu długiej linii zaleca się dodatkowe wcięcia dla zwiększenia czytelności

Komentarze

Zasadniczo kod powinien być *samodokumentujący* jednak w praktyce nie da się uniknąć komentarzy. Przydatne uwagi:

- do opisu jednej operacji stosujemy komentarze `inline` na końcu linii (zaczynające się od znaku `#`)
- większe fragmenty kodu objaśniamy przy pomocy komentarzy blokowych
- funkcje, klasy, metody, moduły powinny być opisane przy pomocy tzw. `docstrings` (PEP 257 - więcej informacji poniżej)

Spacje w wyrażeniach i instrukcjach

Oprócz przestrzeni dodanej przez puste linie, zaleca się też uzupełnianie spacjami kodu w wokół operatorów:

- przypisania (`=`, `+=`, ...): `a = 10`
- porównania (`==`, `<`, `is`, `not in`, ...): `if value == 10:`
- logicznych (and, not, ...): `if value > 10 and value < 20:`

Programowanie

Kilka, zdawało by się, mało istotnych zasad dotyczących programowania znalazło swoje miejsce w dokumencie `PEP 8`

- nie porównujemy wartości zmiennych logicznych do `True` i `False`: `if valid:` a nie `if valid == True:`
- używamy kontekstu logicznego (np. pusta lista ma wartość `False`): `if lista_prac:` a nie `if len(lista_prac) > 0:`

- używamy `is not` zamiast `not ... is` w instrukcji `if`: `if x is not None:` a nie `if not x is None`
- nie używamy `if x:` jeśli mamy na myśli `if x is not None`
- używamy `.startswith()` i `.endswith()` zamiast *slicing'u*

PEP 257 - docstring

Do komentowania większych fragmentów kodu używa się tzw *docstringów*. Szczegółowe zasady ich użycia opisuje dokument `PEP 257`. Na początku przygody z dokumentacją warto trzymać się podstawowych zasad:

- dokumentuj nimi wszystkie publiczne moduły, funkcje, klasy i metody
- zaczynaj i kończ docstringa potrójnymi cudzysłowami
- jednoliniowego docstringa otaczaj potrójmymi cudzysłowami w tej samej linii
- wieloliniowego docstringa kończ potrójnymi cudzysłowami w nowej linii

Przykład użycia pokazujący zalecaną zawartość:

```
def my_function(name):
    """Przeznaczenie funkcji.

    Opis parametrów (opcjonalnie)

    Zwracane wartości (opcjonalnie)

    Warunki wstępne (opcjonalnie)

    Efekty uboczne (opcjonalnie)

    Dodatkowe informacje (dalsze wyjaśnienia, odnośniki do
bibliografii, przykłady użycia) (opcjonalnie)
    """

    pass
```

Reguły

Podczas pisania kodu powinniśmy starać się eliminować częste błędy, jakie popełniają programiści. Możemy to zrobić pamiętając o kilku regułach opisanych poniżej.

DRY - Don't repeat yourself

Jednym z największych grzechów programisty jest **duplikowanie** kodu. Jak poradzić sobie z powtórzeniami? Prawie zawsze najlepszym rozwiązaniem jest nowa metoda czy nowa klasa (w połączeniu z zastosowaniem pewnych wzorców). Na przykład, możemy stworzyć nową metodę i przenieść do niej powtarzający się kod, a duplikacje zastąpić jej wywołaniem.

Dzięki temu skracamy kod źródłowy, ułatwiamy jego zrozumienie i testowanie, oraz pozostaje on łatwiejszy do zmodyfikowania w przyszłości.

KISS - Keep it simple stupid

Kod powinien być tak **prosty**, jak to tylko możliwe. Stworzenie metody, która wykonuje skomplikowaną logikę i działa, wcale nie oznacza, że jest ona dobrze napisana, jeżeli tylko jej autor jest w stanie zrozumieć tę logikę. Dobry programista jest w stanie napisać skomplikowany algorytm w taki sposób, że nie zawiera on niepotrzebnych, nadmiarowych elementów oraz osoba pierwszy raz patrząca na takie rozwiązanie, jest w stanie je zrozumieć.

YAGNI - You Aren’t Gonna Need It

Reguła `YAGNI` mówi o tym, że **nie powinniśmy** tworzyć kodu, który w aktualnym kodzie jest **nieużywany**. Jeżeli z kolei w danym projekcie taki kod widzimy, powinniśmy kod usunąć. W dobie takich rozwiązań jak `git`, bez problemu jesteśmy w stanie go przywrócić w razie potrzeby.

Manifesto for Software Craftsmanship

Manifest tworzenia oprogramowania powstał jako rozwinięcie manifestu Agile. Stwierdza on, iż w pracy nad wytwarzaniem oprogramowania ważne jest:

- nie tylko działające oprogramowanie, lecz również jego **staranne wykonanie**,
- nie tylko reagowanie na zmiany, lecz również ciągłe wytwarzanie wartości dodanej,
- nie tylko ludzie oraz interakcje między nimi, lecz cała społeczność profesjonalistów,
- nie tylko współpraca z klientem, lecz również produktywne partnerstwo.

linters

Zasady dotyczące stylu kodowania w Pythonie są zdefiniowane na tyle przejrzyście, że da się je automatycznie sprawdzić. Najpopularniejsze programy (zwane *linterami*) analizujące kod i wskazujące problemy stylistyczne to:

- `pycodestyle`
- `flake8` - dodatkowo zawierający funkcje debuggera
- `pylint` - szukający błędów programistycznych, problematycznego kodu, sugerujący poprawki

Często oprogramowanie tego typu dostępne jest jako wtyczka do edytora lub IDE.

Autoformatery

Idąc dalej tropem automatycznej kontroli kodu trafiamy na *autoformatery*. Dodają one do funkcjonalności *linterów* automatyczne nanoszenie poprawek w kodzie. Najpopularniejsze autoformatery:

- `autopep8` - korzystający z `pycodestyle`
- `yapf` - stworzony przez Google'a

- `black`
    - reklamowany przez twórców jako *"bezkompromisowy formater kodu"*
    - stosuje reguły formatowania z `PEP 8` + dodatki
    - ma limitowane możliwości konfiguracji co przekłada się na wyjątkowo spójny styl formatowania i pozwala skupić się na zawartości kodu, a nie na opcjach jego formatowania

Installation

First, you need to install Black. You can do this using pip, Python's package manager. Open your terminal or command prompt and run the following command:

pip install black

Basic Usage

Once Black is installed, you can start formatting your Python files. To format a file, simply run Black from the command line with the file name. For example:

black myfile.py

This will format `myfile.py` according to Black's rules.

Formatting an Entire Directory

If you want to format all Python files in a directory, you can run Black with the directory name. For example:

```
black my_directory/
```

This will format all `.py` files in `my_directory` and its subdirectories.