

Programowanie i projektowanie obiektowe

Paweł Daniluk

Wydział Fizyki
Uniwersytet Warszawski

Jesień 2014

Spis treści

1	Wstęp	3
1.1	Polimorfizm i dziedziczenie	4
2	Analiza obiektowa	6
2.1	Obiektowe modelowanie dziedziny	6
2.2	Odnajdywanie klas pojęciowych	7
2.3	Odnajdywanie powiązań	9
2.4	Dodawanie atrybutów	10
3	Projektowanie obiektowe	13
3.1	Metody	14
3.2	Typy danych i powiązania	14
3.3	Dziedziczenie i hierarchia klas	15
3.4	Widoczność	16
3.5	Diagramy przebiegu	17
3.6	Wzorce projektowe (ang. <i>design patterns</i>)	18
4	Obiekty i klasy w Pythonie	20
4.1	Najprostszy obiekt	20
4.2	Atrybuty i metody	21
4.3	Inicjalizacja obiektów	22
4.4	Dziedziczenie	23
4.5	Ograniczenia dostępu	23
4.6	Duck typing	25
4.7	Bardzo szczególne cechy Pythona	26
5	Powiązania i tworzenie obiektów	28
5.1	Implementacja powiązań	29
5.2	Wzorzec projektowy <i>Composite pattern</i>	30
5.3	Iteratory	31
5.4	Tworzenie obiektów	34
6	Metody i dziedziczenie	38
6.1	Metody	38
6.2	Wzorzec projektowy <i>Observer</i>	40
6.3	Dziedziczenie	41
6.4	Przesłanie metod	43
6.5	Klasy abstrakcyjne	45
6.6	Wzorzec projektowy <i>Template method</i>	46

6.7	Kiedy warto rozszerzyć klasę?	46
6.8	Podsumowanie	47
7	Metody statyczne i klasowe	50
7.1	Atrybuty klas	50
7.2	Zastosowania atrybutów klasowych	51
7.3	Metody w klasach	53
7.4	Wzorzec projektowy <i>Singleton</i>	54
7.5	Wzorzec projektowy <i>Factory methods</i> – uzupełnienie	55
8	Wyjątki	57
8.1	Instrukcja <code>try</code>	58
8.2	Wyjątki standardowe	59
8.3	Własne wyjątki	60
8.4	Poprawne posługiwanie się wyjątkami	62

Rozdział 1

Wstęp

Lasciate ogni speranza, voi
ch'entrate.

Dante Alighieri

Tradycyjne języki programowania imperatywnego mają relatywnie ubogi repertuar typów danych. Zazwyczaj są to:

- typy proste – liczby, znaki, wartości logiczne, napisy
- tablice – numerowane ciągi wartości innych typów, np. wektor liczb całkowitych.
- struktury (rekordy) – zestaw nazwanych pól określonych typów.

Mimo pozornej prostoty pozwalają one na przechowywanie w zorganizowany sposób dowolnych danych. Typy proste służą do przechowywania pojedynczych wartości. Tablice znakomicie nadają się w przypadku, gdy konieczne jest składowanie wielu wartości tego samego rodzaju, które (z reguły) należą do jednego zbioru. Przykładowo, tablice można wykorzystać do przechowywania wektorów i macierzy albo danych o studentach należących do grupy ćwiczeniowej. W ostatnim przypadku należy wpięrcz dysponować typem opisującym pojedynczego studenta – jest to typ pojedynczego elementu tablicy. Do tego celu służą struktury (czasem zwane rekordami). Pozwalają one na grupowanie nazwanych elementów, które w odróżnieniu od tablic mogą mieć różne typy. W ten sposób można zdefiniować strukturę `Student`, która ma atrybuty `imie` i `nazwisko` będące napisami, `n` `indeksu` będący liczbą, oraz `dataur` będący datą.

Z reguły programy w językach imperatywnych składają się z funkcji, które pozwalają na wyróżnienie wielokrotnie wykorzystywanych elementów programu. Funkcje przyjmują argumenty (możliwe są również funkcje bezargumentowe) i na ich podstawie obliczają wynik. W programie (module programu) może występować co najwyżej jedna funkcja o danej nazwie¹.

Stosując opisany powyżej sposób programowania wielokrotnie natrafia się na trudność wynikającą z faktu, że operacja o określonej nazwie może być w różny

¹Powyższe rozważania są uogólnieniem. Istnieją języki programowania, dla których powyższe stwierdzenia nie są prawdziwe

sposób realizowana dla konkretnych typów danych. Na przykład funkcja `pole` służąca do obliczania pola figury geometrycznej musi zachowywać się różnie w zależności od figury, która została przekazana jako argument:

```
def pole(figura):
    if figura is Kolo:
        return pi * figura.r * figura.r
    elif figura is Kwadrat:
        return figura.a * figura.a
    elif figura is Prostokat:
        return figura.a * figura.b
```

Taka funkcja jest skomplikowana i wymaga aktualizacji za każdym razem, gdy w systemie pojawia się nowy rodzaj figury geometrycznej. Alternatywnie można zaimplementować niezależne funkcje dla każdej figury o nazwach `poleKola`, `poleKwadratu` itd. Wada takiego rozwiązania objawia się w sytuacji, gdy dysponujemy figurą nieznanego rodzaju i chcemy obliczyć jej pole. Wtedy wywołanie odpowiedniej funkcji należy poprzedzić szeregiem testów. Z reguły w dużym systemie funkcji podobnych do rozważanej jest wiele (w przypadku figur geometrycznych mogą to być `obwod`, `rysuj`, `srednicaokreguopisanego`), co prowadzi do programu skomplikowanego i trudnego w utrzymaniu.

Programowanie obiektowe pozwala na łatwe rozwiązanie przedstawionych trudności. W programowaniu obiektowym dane przechowywane są w formie obiektów, które do pewnego stopnia są podobne do struktur. Jednak oprócz atrybutów do obiektów mogą być przypisane również metody, czyli fragmenty programu opisujące operacje, które na konkretnym obiekcie można wykonać. Obiekty często opowiadają rzeczywistym przedmiotom (osobom, zdarzeniom), których dotyczy działanie systemu. W przykładzie opisanym powyżej obiektami byłyby figury geometryczne. Ich atrybuty zależałyby od rodzaju figury (koło – `r`; kwadrat – `a`; prostokąt – `a`, `b`). W tym zakresie rozwiązanie obiektowe nie różni się od przedstawionego powyżej. Jednakowoż poza atrybutami obiekty mogą również posiadać metody. W tym wypadku byłaby to metoda `pole`, która obliczałaby i zwracała pole obiektu, dla którego została wywołana.

Obiekty podobnego rodzaju grupuje się w *klasy*. Obiekty należące to jednej klasy mają te same metody. W ten sposób można niewielkim wysiłkiem tworzyć obiekty posiadające pożądane własności.

1.1 Polimorfizm i dziedziczenie

O ile w programowaniu nieobektowym może istnieć tylko jedna funkcja o danej nazwie, w przypadku obiektowym różne obiekty mogą mieć metody o tej samej nazwie. Pozwala to na uzyskanie efektu, w którym znaczenie instrukcji zależy od kontekstu. Przykładowo napis `f.pole()` może oznaczać zastosowanie różnego wzoru w zależności od rodzaju figury będącej wartością zmiennej `f`. Taki efekt nazywa się *polimorfizmem*. W zależności od konkretnego języka programowania identyfikuje się różne rodzaje polimorfizmu. Wyróżnia się między innymi:

polimorfizm statyczny w którym na etapie kompilacji da się przewidzieć, która funkcja zostanie wykonana, na podstawie zadeklarowanych typów zmiennych

polimorfizm dynamiczny w którym dopiero podczas wykonania programu określana jest właściwa funkcja (metoda)

polimorfizm uniwersalny polegający na tworzeniu podprogramów (np. funkcji) działających na różnych typach danych (np. iloczyn skalarny).

polimorfizm ad-hoc gdzie właściwa funkcja jest dobierana na podstawie typu argumentów.

Ponadto w programowaniu obiektowym dopuszcza się definiowanie nowych klas na podstawie już istniejących. Wymagane jest wówczas określenie różnicy pomiędzy klasą podstawową i klasą potomną. Ten sposób definiowania klas nazywa się *dziedziczeniem*, klasę definiowaną (dziedziczącą) nazywa się *podklasą*, a klasę (klasy), z której następuje dziedziczenie – *nadklasą*. Tym sposobem fragmenty definicji wspólne dla wielu klas można ulokować w nadklasie, z której będą one dziedziczyć.

Rozdział 2

Analiza obiektowa

Odpowiednie dać rzeczy słowo!

Cyprian Kamil Norwid

Realizacja każdego projektu informatycznego składa się z następujących etapów:

1. Analiza
2. Projekt
3. Implementacja
4. Wdrożenie
5. Konserwacja

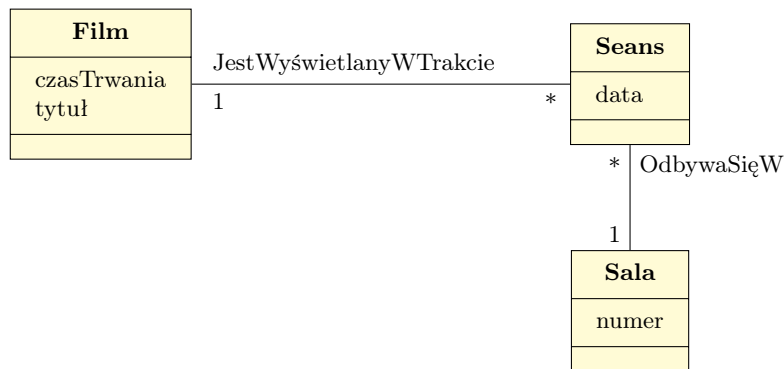
W tym rozdziale skupimy się na pierwszym etapie – *analizie* – ze szczególnym uwzględnieniem analizy obiektowej. Celem tego etapu jest precyzyjne opisanie problemu i określenie docelowych wymagań. Jego pominięcie lub zaniedbanie może mieć fatalne i nieusuwalne skutki w postaci programu, który nie spełnia założeń. Należy podkreślić, że celem analizy jest badanie i opisanie problemu. Podczas jej przeprowadzania **nie należy** rozważać konkretnych rozwiązań programistycznych.

Analiza obiektowa polega na identyfikacji i klasyfikacji *obiektów pojęciowych*. Obiekty pojęciowe nie mają związku z obiektami, które istnieją w programie obiektowym – odpowiadają one pojęciom i koncepcjom ze świata rzeczywistego. Jest to zagadnienie zasadniczo odmienne od projektowania, gdzie celem jest znalezienie konkretnego rozwiązania (programistycznego i sprzętowego), które realizuje wymagania wynikające z przeprowadzonej analizy. Elementem projektu może być schemat bazy danych albo opis klas programowych.

Analiza i projektowanie dają się krótko streścić jako “zrób co należy (analiza) oraz zrób to jak należy (projektowanie)”.

2.1 Obiektowe modelowanie dziedziny

Model dziedziny odzwierciedla pojęcia z modelowanej części świata rzeczywistego oraz zależności pomiędzy nimi. Jego elementami są *klasy pojęciowe*, które



Rysunek 2.1: Fragment modelu dziedziny systemu obsługi kin

grupują obiekty o jednakowych właściwościach. Model dziedziny określa również powiązania pomiędzy klasami pojęciowymi oraz posiadane przez nie atrybuty.

W modelu dziedziny nie zajmujemy się klasami programowymi (ang. *software class*). Może on posłużyć jako źródło inspiracji przy ich projektowaniu, ale nie w drugą stronę. W wypadku systemu informacyjnego dla Zakładu Transportu Miejskiego klasami pojęciowymi mogą być: Autobus, Linia i Kierowca.

Na rysunku 2.1 przedstawiono fragment modelu dziedziny dla systemu obsługi kina. Przykład zawiera trzy klasy pojęciowe: Film, Seans, Sala. Odpowiadają one kategoriom ze świata rzeczywistego. Film jest wytworem artystycznym, Sala fizyczną lokalizacją, a Seans zdarzeniem. Do własności filmu zalicza się m.in. tytuł i czas trwania. Każdy seans ma datę, kiedy się odbywa. Sale są identyfikowane przy pomocy numerów. Ponadto na diagramie odzwierciedlony jest fakt, że filmy są wyświetlane podczas seansu (związek JestWyświetlanyWTrakcie). Liczby i znaki * oznaczają krotność związku. Z diagramu wynika, że seans odbywa się w dokładnie jednej sali kinowej, oraz że w sali kinowej może mieć miejsce dowolnie wiele seansów (aczkolwiek oczywiście nie w jednym terminie).

2.2 Odnajdywanie klas pojęciowych

Pierwszym etapem przeprowadzania analizy obiektowej jest identyfikacja istotnych dla realizowanego zadania klas pojęciowych. Jeżeli dysponujemy tekstowym opisem zadania, dobrym pierwszym przybliżeniem jest odnalezienie fraz rzeczownikowych. W tekście poniżej podkreślono wszystkie frazy rzeczownikowe. Podwójną linią wyróżniono potencjalne klasy pojęciowe.

Towary w supermarkecie wystawione są na półkach, które pogrupowane są w regaly i alejki. Na półce mogą znajdować się produkty jednego rodzaju. Półka ma ograniczoną pojemność. Ponadto istnieje magazyn, gdzie przechowywane są zapasy towarów. Sprzedaż jest rejestrowana w kasach. Należy zaprojektować system pozwalający na rejestrowanie ilości towarów wystawionych w sklepie oraz składowanych w magazynie. Na podstawie sprzedaży należy prognozować stany na półkach i generować zlecenia uzupełnienia towarów. Należy również wprowadzić możliwość rejestrowania stanów faktycznych.

Niektóre rzeczowniki nie mają związku z problemem będąc nazwami cech lub czynności (pojemność, możliwość, rejestrowanie). Inne nie mają przełożenia na rzeczywiste obiekty, które są przedmiotem działania systemu (system, sklep, supermarket). Pomijając dwa ostatnie przyjmujemy, że oprogramowanie jest tworzone dla jednego konkretnego sklepu. To założenie nie zawsze musi być poprawne.

Po odrzuceniu rzeczowników, które nie mają związku z problemem (pojedyncze podkreślenie), należy uporządkować pojęcia znajdując synonimy:

- *towar, produkt*
- rodzaj
- półka
- regał
- alejka
- magazyn
- *zapas towaru, ilość towaru, stan na półce, stan faktyczny*
- sprzedaż
- kasa
- zlecenie uzupełnienia towaru

Na koniec należy wybrać pojęcia, które są istotne dla rozwiązania problemu oraz je dobrze zdefiniować, jeżeli ich znaczenie nie jest jasne. W przedstawionym przykładzie należy zwrócić uwagę na klasę Produkt. Pojęcie to można rozumieć na dwa sposoby: jako konkretny egzemplarz produktu albo jako rodzaj produktu oferowanego przez sklep. Pierwsze rozwiązanie jest poprawne, jeżeli towary są rozróżnialne (np. przez nr seryjny) i to rozróżnienie jest istotne z punktu widzenia działania systemu. W przeciwnym wypadku osobne ewidencjonowanie każdej paczki ciasteczek byłoby absurdalne. Doświadczenie podpowiada, że druga definicja jest dla supermarketu korzystniejsza.

Drugim problematycznym zestawem pojęć są ilości, stany i zapasy produktów. Można każdemu produktowi przypisać obiekty opisujące jego ilość w różnych lokalizacjach sklepu, albo przechowywać informację o ilości produktu jako atrybut konkretnej lokalizacji. Wskazówka, że na półce mogą znajdować się produkty jednego rodzaju, sugeruje, że drugie prostsze rozwiązanie jest wystarczające. Zawsze przy znajdowaniu klas pojęciowych należy:

- używać istniejących nazw,
- nie zajmować się niczym, co nie dotyczy modelowanej części rzeczywistości,
- nie dodawać rzeczy, których nie ma.

Zalecenie używania istniejących nazw utrudnia dodawanie klas pojęciowych, które w rzeczywistości nie występują. Dodatkowo na tym etapie należy się powstrzymać przed nadmiernymi próbami porządkowania rzeczywistości przez

Kategoria	Przykłady	Klasy z dziedziny gry w Monopol
transakcje	SprzedażBiletu, RezerwacjaMiejsc	
pozycje transakcji	PozycjaRezerwacji	
produkty bądź usługi związane z transakcjami i kontraktami lub ich pozycjami	Bilet	
gdzie transakcje są odnotowywane	Kasa, WykazDostępnychMiejsc	
role ludzi i organizacji związanych z transakcją	Kasjer	Gracz
miejsce zajścia transakcji /obsługi transakcji	Kasa, SalaKinowa	
zdarzenia (często trzeba pamiętać czas ich zajścia)	Seans	GraWMonopol
obiekty fizyczne	Bilet, Kino, Kasa, Miejsce	Plansza, Pionek, Kostka
opisy	OpisSeansu	
katalogi	KatalogSeansów, KatalogFilmów	
kontenery rzeczy fizycznych lub informacji	Kino, SalaKinowa	Plansza
rzeczy w kontenerach	SalaKinowa, Miejsce	Pole
inne współpracujące systemy	SystemAutoryzującyPłatnościElektroniczne	
potwierdzenia, rejestry, kontrakty, zagadnienia prawne	Pokwitowanie, PotwierdzenieRezerwacji	
instrumenty finansowe	Czek, Gotówka	
harmonogramy, instrukcje, dokumenty regularnie używane podczas wykonywania prac	DziennaListaPromocji	

Tablica 2.1: Często spotykane kategorie klas pojęciowych

tworzenie klas ogólniejszych niż przedmioty, które faktycznie występują w problemie (np. Pojazd zamiast Samochód i Motocykl). Należy również bezwzględnie pomijać pojęcia, które co prawda istnieją, ale nie są nieodzowne dla opisywanego procesu. Tabela 2.1 zawiera przykłady często spotykanych klas pojęciowych.

2.3 Odnajdywanie powiązań

Powiązanie (ang. *association*) między klasami wskazuje, że pomiędzy obiektami do nich należącymi może występować jakaś zależność. W modelu dziedziny istotne są powiązania, które są niezbędne do wypełnienia wymagań informacyjnych i pomagają zrozumieć dziedzinę. Warto w modelu umieszczać powiązania, o których trzeba przynajmniej przez jakiś czas “pamiętać”. Najczęściej spotyka się powiązania odzwierciedlające różne rodzaje zawierania, bycia częścią lub ele-

Kategoria	Przykłady	Klasy z dziedziny gry w Monopol
A jest transakcją związaną z inną transakcją B	Płatność–RezerwacjaMiejsc	
A jest pozycją transakcji B	PozycjaRezerwacji–RezerwacjaMiejsc	
A jest produktem lub usługą z transakcji lub pozycji transakcji B	Bilet–SprzedażBiletu	
A jest rolą związaną z transakcją B	Klient–Płatność	
A jest fizyczną lub logiczną częścią B	Miejsce–SalaKinowa, SalaKinowa–Kino	Pole–Plansza, Pole–KompletPól, GraWMonopol–Plansza, GraWMonopol–Kostka, GraWMonopol–Pionek
A fizycznie lub logicznie przechowywane w/na B	Kasa–Kino	Pionek–Pole, Pole–Plansza
A jest opisem B	OpisSeansu–Seans	
A jest rejestrowane, zgłaszane, utrwalane, pamiętane w/na B	SprzedażBiletu–Kasa	Pionek–Pole, GraWMonopol–Plansza
A jest uczestnikiem/pracownikiem/członkiem B	Kasjer–Kino	Gracz–GraWMonopol
A jest organizacyjną podjednostką B	Kino–SiećKin	
A używa, zarządza lub posiada B	Kasjer–Kasa	Gracz–Pionek
A jest obok B	PozycjaRezerwacji–PozycjaRezerwacji	

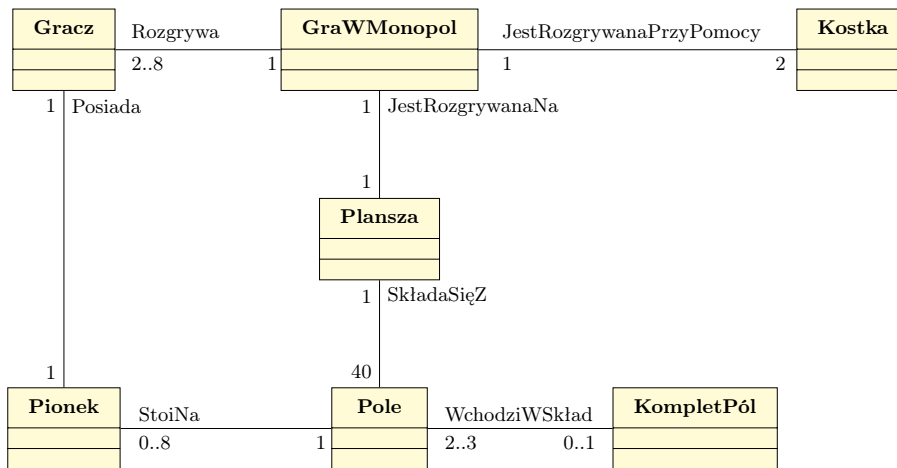
Tablica 2.2: Często spotykane kategorie powiązań

mentem, hierarchie (np. służbowe) oraz uczestnictwo w procesach. Tabela 3.2 zawiera często spotykane kategorie powiązań wraz z przykładami.

Przy wybieraniu nazw dla powiązań należy mieć na uwadze, że nie każda nazwa, która pasuje, pozwoli innym zorientować się, o co nam chodziło. Do wielu powiązań będzie na przykład pasować nazwa *Dotyczy*, ale nie przenosi ona wielu informacji. Poza określeniem faktu i nazwy powiązania podaje się jego *krotność* (liczebność). Jest to liczba obiektów, które mogą być powiązane z jednym obiektem z drugiej strony powiązania. Krotność można podawać w formie pojedynczej liczby, kilku liczb, lub przedziału. Podaje się ją niezależnie dla obydwu stron powiązania. Na rysunku 2.1 zaznaczono, że każdemu obiektowi klasy *Seans* odpowiada dokładnie jeden obiekt klasy *Film*, zaś obiektowi klasy *Film* może odpowiadać dowolnie wiele obiektów klasy *Seans* (w szczególności żaden).

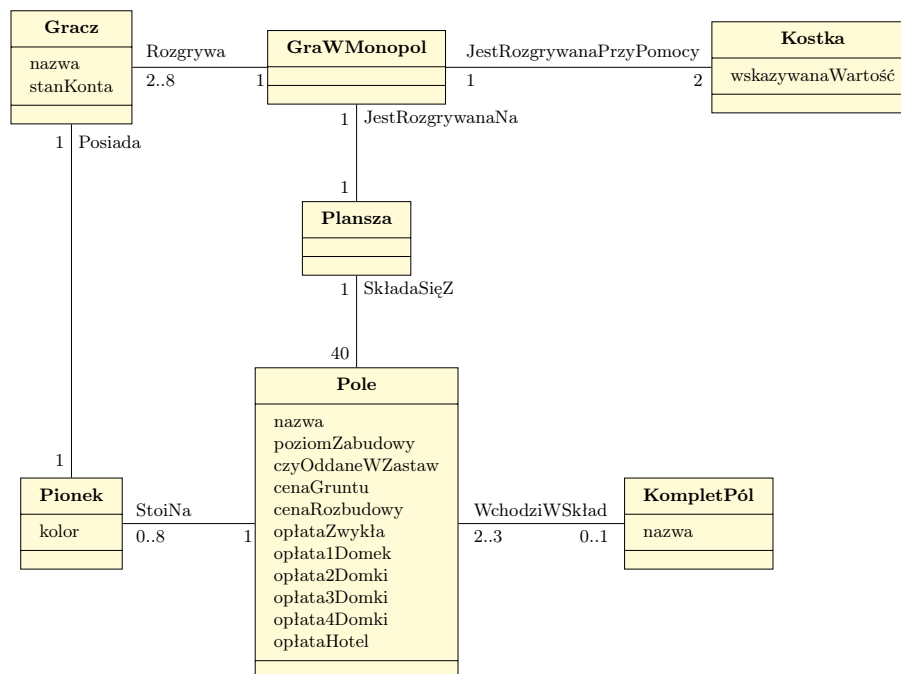
2.4 Dodawanie atrybutów

Ostatnim etapem budowania modelu dziedziny jest dodawanie atrybutów do klas pojęciowych. Atrybuty służą do opisu obiektów należących do danej klasy.



Rysunek 2.2: Powiązania w modelu dziedziny dla gry w Monopol

Typ ich wartości powinien należeć do typów prostych lub być tablicą o elementach typu prostego. Jeżeli wydaje się, że nie istnieje odpowiedni typ prosty, to najprawdopodobniej we wcześniejszym etapie pominięto klasę pojęciową. Ważne jest, aby atrybuty umieszczać we właściwych klasach. Nie wszystkie klasy pojęciowe muszą mieć atrybuty. Rysunek 2.3 przedstawia fragment obiektowego modelu dziedziny dla gry w Monopol.



Rysunek 2.3: Częściowy model dziedziny dla gry w Monopol

Rozdział 3

Projektowanie obiektowe

It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience.

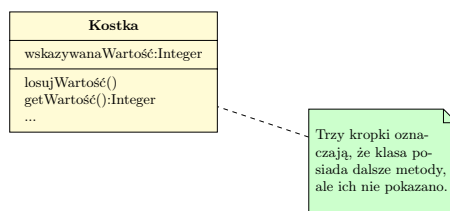
Albert Einstein

Projektowanie jest fazą rozwoju oprogramowania, której celem jest znalezienie rozwiązań technicznych prowadzących do programu/systemu spełniającego wymagania sformułowane podczas analizy. Podczas projektowania zazwyczaj pomija się niskopoziomowe lub oczywiste (dla zamierzonych odbiorców projektu) szczegóły i koncentruje się na wysokopoziomowych pomysłach oraz ideach.

Projektowanie obiektowe polega na określeniu klas programowych, ich atrybutów i metod oraz opisanu sposobu ich działania. Projekt obiektowy stanowi podstawę do tworzenia implementacji klas w konkretnym języku programowania. Na projekt obiektowy składają się informacje o:

- klasach,
- atrybutach,
- metodach,
- dziedziczeniu,
- powiązaniach (wraz ze sposobem realizacji).

Dobrym punktem wyjściowym do określenia klas programowych jest obiektowy model dziedziny i jego klasy pojęciowe. Każda klasa pojęciowa wraz ze swoimi atrybutami musi zostać odzwierciedlona w projekcie. Może się zdarzyć (szczególnie w przypadku skomplikowanych obiektów), że z różnych powodów jednej klasie pojęciowej będzie odpowiadało kilka klas programowych lub (zwłaszcza w



Rysunek 3.1: Klasa Kostka wraz z metodami

przypadku występowania powiązania jeden-do-jednego) że klasy pojęciowe zostaną scalone w jedną klasę programową. Ponadto podczas projektowania może okazać się konieczne zdefiniowanie szeregu klas pomocniczych. Atrybuty klas pojęciowych powinny zostać przypisane do odpowiadających im klas programowych. Można pominąć atrybuty, których wartość zamiast przechowywać można obliczać w razie potrzeby.

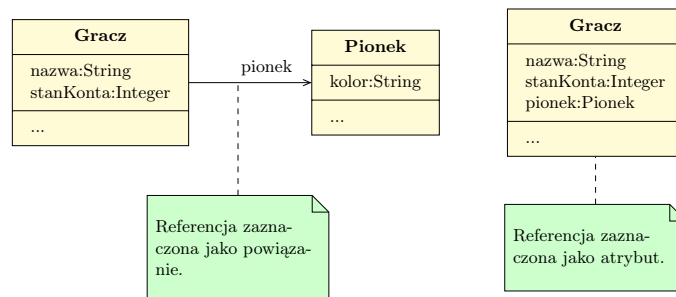
3.1 Metody

Można powiedzieć, że programowanie obiektowe polega na wyznaczaniu obiektom odpowiedzialności. Te odpowiedzialności są dwóch typów: za pamiętanie pewnych danych i za wykonywanie operacji na tych danych. Odpowiedzialność za pamiętanie jest realizowana poprzez atrybuty. Znacznie trudniejsze i istotniejsze jest właściwe rozdzielenie odpowiedzialności za wykonanie operacji. W projekcie obiektowym realizuje się to poprzez przypisywanie metod do klas programowych. Metoda jest w swojej koncepcji rodzajem funkcji przypisanej do konkretnego obiektu. Może przyjmować argumenty i zwracać wartości. Efekt działania metody zależy również od obiektu, dla którego została wywołana, i jego atrybutów. W większości przypadków nazwa metody wystarczająco precyzyjnie określa jej działanie. W przeciwnym wypadku może być konieczne dodanie do projektu dodatkowego opisu. Rysunek 3.1 przedstawia klasę Kostka z gry Monopol wraz z niektórymi metodami.

3.2 Typy danych i powiązania

Nawet jeżeli język programowania, który zostanie użyty do implementacji systemu, tego nie wymaga, warto jest w projekcie określić typ wartości atrybutów klas oraz typy argumentów i wartości zwracanych przez metody. Pozwala to między innymi na weryfikację, czy zaprojektowane atrybuty mają właściwy typ. Jeżeli nie istnieje odpowiedni typ prosty, ale istnieje w projekcie pasująca klasa, należy domniemywać, że w obiektowym modelu dziedziny atrybut należy zastąpić powiązaniem. Jeżeli nie ma pasującej klasy, najprawdopodobniej należy uzupełnić model.

Zazwyczaj powiązania pomiędzy klasami realizuje się przy pomocy atrybutów. Projektując należy zdecydować, do której z powiązanych klas będzie należał atrybut określający powiązanie. Jest technicznie możliwe utworzenie atrybutów w obydwu klasach, ale konsekwencją takiej decyzji jest konieczność utrzymywania spójności pomiędzy ich wartościami. Przy takim rozwiązaniu, bowiem, ta sama informacja jest przechowywana w dwóch miejscach i możliwe



Rysunek 3.2: Sposoby zaznaczania powiązań

jest pojawienie się sprzeczności. W przypadku powiązań wiele-do-wielu atrybut określający powiązanie musi być typu tablicowego¹. Na diagramie powiązanie realizowane przez atrybut można opatrzyć strzałką o grocie skierowanym od klasy zawierającej atrybut. Przykłady notacji oznaczającej powiązanie przedstawiono na rysunku 3.2.

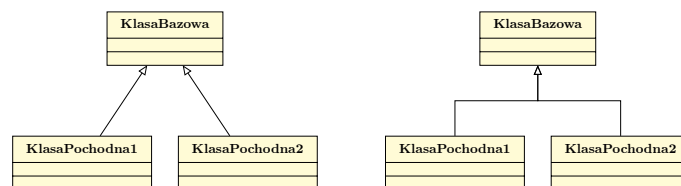
3.3 Dziedziczenie i hierarchia klas

Dziedziczenie jest fundamentalnym aspektem projektowania obiektowego. Służy ono odzwierciedleniu naturalnej zależności pomiędzy klasami, w której jedna z klas jest szczególnym przypadkiem lub rozszerzeniem drugiej. Klasa pochodna (podklasa) zachowuje wszystkie cechy i funkcjonalności klasy bazowej (nadklasy) oraz może posiadać dodatkowy zestaw cech i funkcjonalności charakterystycznych tylko dla obiektów do niej należących. W fazie projektowania należy zidentyfikować takie zależności pomiędzy klasami pojęciowymi z modelu dziedziny. Przykładowo w systemie zawierającym klasy pojęciowe Pracownik i Kierownik obiekty klasy Kierownik bez wątplenia przynależą również do klasy Pracownik, ponieważ każdy kierownik musi być równocześnie pracownikiem przedsiębiorstwa. Rzecz jasna nie zachodzi zależność odwrotna – nie każdy pracownik jest kierownikiem. Można zatem w projekcie przyjąć, że klasa Kierownik powinna dziedziczyć z klasy Pracownik. Nietrudno zauważyć, że takie działanie przyniesie korzyść, albowiem wiele funkcjonalności związanych z byciem pracownikiem (w zależności od przeznaczenia systemu: obsługa danych osobowych, obliczenie wynagrodzenia itp.) musi być również zaimplementowane dla kierowników.

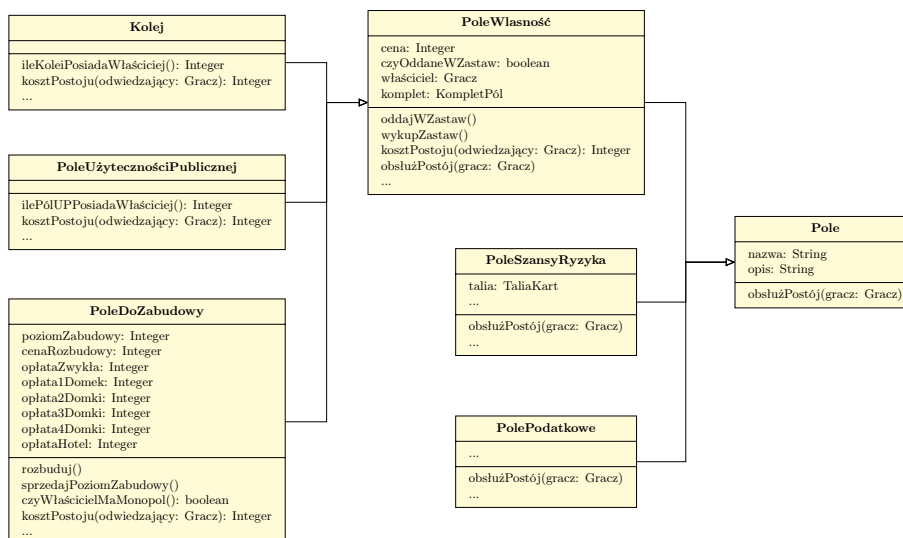
Niestety z reguły ma miejsce przypadek trudniejszy. W modelu dziedziny istnieją klasy mające część wspólnych funkcjonalności, ale nie zawierające się w sobie. Przykładem tutaj mogą być klasy SamochódOsobowy i Ciężarówka. W takiej sytuacji wskazane jest umieszczenie w projekcie dodatkowej pomocniczej klasy (o nazwie PojazdSamochodowy) realizującej funkcjonalności wspólne dla samochodów osobowych i ciężarowych. Klasa ta nie mogła pojawić się na etapie analizy, albowiem nie istnieją w świecie rzeczywistym² obiekty, które będąc pojazdami samochodowymi nie byłyby równocześnie ciężarówką lub samochodem

¹Lub równoważnego. W zależności od języka można zastosować listę, zbiór albo inną strukturę pozwalającą przechowywać wiele wartości tego samego typu.

²Przynajmniej w zakresie rozważanym w przykładzie.



Rysunek 3.3: Sposoby zaznaczania dziedziczenia



Rysunek 3.4: Pola w grze w Monopol

osobowym. Niemniej jednak dodanie takiej klasy do projektu daje natychmiastową korzyść, gdyż pozwala uniknąć dublowania definicji atrybutów i metod. Jeżeli zachodzi taka potrzeba, w klasie pochodnej można na nowo zdefiniować metody odziedziczone z klasy bazowej.

Schemat zależności wynikających z dziedziczenia nazywamy *hierarchią klas*. Przypomina ona drzewo genealogiczne. Na schemacie dziedziczenie oznacza się strzałką o trójkątnym zamkniętym niewypełnionym grocie (rys. 3.3).

Rysunek 3.4 przedstawia hierarchię klas odpowiadających polom planszy gry w Monopol.

3.4 Widoczność

Poza składowymi (funkcjonalnościami), które klasa udostępnia, może ona zawierać składowe pomocnicze niezbędne do realizacji tych funkcjonalności. Kostka z rys. 3.1 posiada dwie istotne funkcjonalności: metody `losujWartosc` i `getWartosc`. Pierwsza z nich pozwala na wylosowanie nowej wartości wskazywanej przez kostkę, czyli symuluje rzut prawdziwą kostką do gry. Druga pozwala na pobranie wskazywanej wartości, czyli odpowiada spojrzeniu gracza na kostkę. Aby możliwe było zrealizowanie tych funkcjonalności, kostka musi mieć zdolność pamiętania, jaką wartość aktualnie wskazuje. Na diagramie jest to zrealizowane

przy pomocy atrybutu `wskazywanaWartosc`. Powiemy, że ten atrybut jest składową pomocniczą.

Z reguły nie wszystkie składowe pomocnicze odnotowuje się w projekcie. Co więcej, na składowych pomocniczych nie należy polegać. W trakcie rozwoju oprogramowania mogą się one zmieniać. Dlatego w projekcie (i niektórych językach programowania obiektowego) można określać stopień widoczności składowych klasy. Mogą one być:

- publiczne (ang. *public*) – mogą ich bez ograniczeń używać obiekty wszystkich klas (oznaczenie: +)
- chronione (ang. *protected*) – mogą ich bez ograniczeń używać obiekty tej samej klasy lub jej podklas (oznaczenie: #)
- prywatne (ang. *private*) – mogą ich używać jedynie obiekty tej samej klasy (oznaczenie: -)

Określając widoczność składowych klasy projektant lub programista mogą mieć pewność, że nie są one używane w innych miejscach systemu i, co za tym idzie, są w stanie określić zasięg potencjalnych zmian, które trzeba wykonać po modyfikacji składowej.

Niekiedy praktykuje się tzw. *kapsułkowanie*, które polega na zapewnianiu dostępu do prywatnych atrybutów poprzez odpowiednie metody. Zazwyczaj stosuje się metody o nazwach rozpoczynających się od `get`, `set` i `is`. Przykładowo metoda ustawiająca wartość atrybutu `Attr` będzie się nazywała `setAttr`, pobierająca wartość – `getAttr`. W przypadku atrybutu będącego wartością logiczną (prawda/fałsz) warto zastosować nazwę `isAttr`. Posługując się kapsułkowaniem można emulować atrybuty tylko do odczytu lub tylko do zapisu. Można również zagwarantować, że przy każdej zmianie wartości atrybutu lub jego pobraniu będą wykonywane dodatkowe czynności.

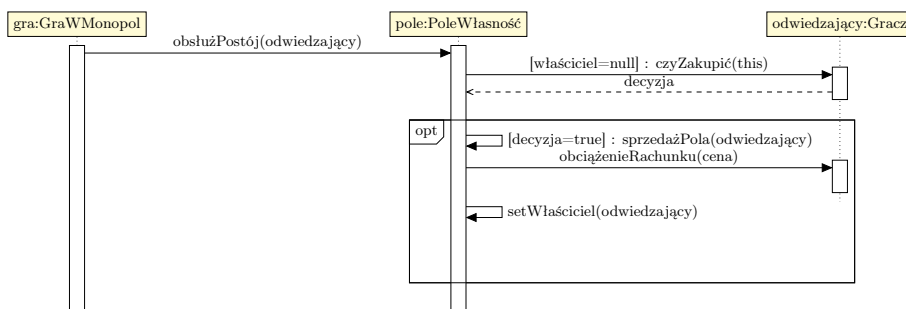
3.5 Diagramy przebiegu

Diagramy przebiegu służą do przedstawiania interakcji obiektów. Pozwalają one zaprezentować kolejność wywoływania metod podczas realizacji wybranego procesu. Tworząc lub interpretując diagramy przebiegu należy pamiętać, że poziom szczegółowości zależy od zamiaru autora i służy do przedstawienia istotnych aspektów procesu. Rysunek 3.5 ilustruje proces obsługi postoiu pionka na polu własność w grze w Monopol. Za tę funkcję odpowiada metoda `obsluzPostoj` w klasie `PoleWlasnosc`. Jeżeli pole nie ma właściciela, gracz, który je odwiedził, może je kupić. Metoda `czyZakupic` klasy `Gracz` obsługuje podjęcie tej decyzji³. Jeżeli gracz podejmie decyzję pozytywną, wykonywana jest metoda `sprzedazPola`, która obsługuje procedurę zakupu.

3.5.1 Sprzężenie, a spójność

Sprzężenie (ang. *coupling*) jest miarą jak bardzo obiekty, podsystemy lub systemy zależą od siebie nawzajem. Przykładowo obiekt wykonujący metodę innego obiektu jest z nim sprzężony. Tak samo podklasa jest sprzężona z nadklasą.

³W przypadku gracza automatycznego mamy tutaj do czynienia z jakąś implementacją AI. W przypadku gracza ludzkiego program czeka na reakcję użytkownika



Rysunek 3.5: Fragment obsługi odwiedzin pola własność w grze w Monopol.

Zbyt wysoki stopień sprzężenia jest niepożądany. Każda zmiana w klasie może pociągać za sobą konieczność dokonania zmian w klasach z nią sprzężonych.

Spójność (ang. *cohesion*) to miara jak funkcjonalnie powiązane są metody danej klasy. Warto dbać o utrzymanie wysokiej spójności. Można to osiągnąć przez tworzenie klas, które służą do obsługi tylko jednego zagadnienia lub tworzenie klas pomocniczych i korzystanie z nich w klasie, która pierwotnie miała obsługiwać wiele funkcjonalności.

Paradoksalnie może się wydawać, że zachowanie wysokiej spójności przez zlecenie odpowiedzialności ma negatywny wpływ na sprzężenie. Jednak klasy mające wiele odpowiedzialności są zazwyczaj sprzężone z wieloma innymi klasami.

3.6 Wzorce projektowe (ang. *design patterns*)

Większość problemów można rozwiązać na wiele sposobów. Wybierając konkretne rozwiązanie można kierować się następującymi kryteriami:

- prostota
- pracochłonność
- elastyczność
- łatwość w utrzymaniu

Wszystkie te cechy są pożądane, ale niestety niektóre pozostają ze sobą w sprzeczności. Często rozwiązania proste są mało elastyczne, a czas zaoszczędzony podczas tworzenia oprogramowania skutkuje zwiększonymi kosztami utrzymania. Znalezienie równowagi pomiędzy tymi i innymi istotnymi dla projektu kryteriami jest zadaniem projektanta. Szczęśliwie w wielu przypadkach można posilkować się rozwiązaniami znanymi.

Wzorce projektowe (ang. *design patterns*) stanowią dobrze znane pary problem-rozwiazanie wraz z zaleceniami odnośnie stosowania, analizą zalet i wad, sposobami implementacji itd. Projektant może wybrać wzorec projektowy pasujący do fragmentu projektowanego systemu kierując się jego cechami. Zaletą stosowania wzorców projektowych jest również standardyzacja. Tak stworzony system jest lepiej zrozumiały dla innych programistów i tym samym łatwiejszy w konserwacji. Tabela 3.1 zawiera listę wybranych wzorców projektowych.

Nazwa	Opis
Abstract factory	Sposób tworzenia rodzin powiązanych obiektów bez określania ich klas.
Builder	Oddzielenie tworzenia skomplikowanego obiektu od jego reprezentacji. Pozwala na użycie tego samego procesu do tworzenia różnych reprezentacji.
Factory method	Metoda tworząca pojedynczy obiekt. Klasa tworzonego obiektu zależy od przekazanych argumentów i definicji metody w podklasie.
Object pool	Pozwala na uniknięcie potencjalnie kosztownego tworzenia i niszczenia obiektów poprzez ponowne użycie obiektów już niepotrzebnych.
Prototype	Tworzenie nowych obiektów na podstawie prototypowej instancji.
Adapter albo Wrapper	Dostosowuje interfejs klasy tak, aby mogła być wykorzystana w miejscu, do którego normalnie by nie pasowała.
Composite	Organizuje obiekty w strukturę drzewiastą odzwierciedlającą hierarchię zawierania się obiektów w sobie.
Facade	Ujednolicony i uproszczony interfejs do złożonego podsystemu.
Module	Grupuje powiązane funkcjonalnie elementy (klasy, singletony, metody itp.).
Iterator	Umożliwia sekwencyjny dostęp do elementów składowanych w większym obiekcie bez eksponowania jego struktury wewnętrznej.
Null object	Użycie domyślnego obiektu zamiast pustych referencji.
Observer albo Publish/subscribe	Zależność jeden-do-wielu, gdzie zmiana stanu pierwszego obiektu powoduje automatyczne powiadomienie obiektów powiązanych.
Strategy	Zestaw klas implementujących różne algorytmy umożliwiające ich zamienne stosowanie.

Tablica 3.1: Wybrane wzorce projektowe

Rozdział 4

Obiekty i klasy w Pythonie

W dalszej części wykładu będziemy posługiwać się językiem Python. Jest to język obiektowy, który w ostatnich latach zdobył olbrzymią popularność, zwłaszcza w środowisku naukowym. Należy jednak pamiętać, że realizacja paradygmatu obiektowego w Pythonie znacząco odbiega od konwencji przyjętych w takich językach programowania jak C++ lub Java.

Jak każdy język obiektowy Python pozwala na definiowanie klas. W założeniu klasy te (będziemy je nazywali klasami implementacyjnymi albo programowymi) odpowiadają klasom projektowym, aczkolwiek na etapie uszczegóławiania projektu może pojawić się konieczność utworzenia dodatkowych klas pomocniczych.

Każdy obiekt w Pythonie jest instancją pewnej klasy. Jego funkcjonalność jest określona przez definicję klasy, do której należy. Ta funkcjonalność jest zdefiniowana przy pomocy należących do klasy metod. Obiekt może również posiadać atrybuty, które służą do przechowywania danych z nim związanych. Te atrybuty odpowiadają atrybutom określonym w projekcie. Zauważmy, że o ile obiekty należące do jednej klasy mają jednakowe metody, to zazwyczaj wartości ich atrybutów się różnią. Dlatego można powiedzieć, że:

Klasa odpowiada za metody obiektów do niej należących,

a

obiekty odpowiadają za przechowywanie wartości swoich atrybutów.

To rozróżnienie jest dosyć istotne, albowiem w Pythonie nie jest konieczne deklarowanie zmiennych przed ich użyciem. Ta zasada dotyczy również atrybutów. Każdy obiekt może mieć atrybuty tworzone *ad-hoc*, a w definicji klasy atrybutów nie deklaruje się wcale.

4.1 Najprostszy obiekt

Poniższy fragment programu przedstawia minimalną definicję klasy:

```
class A:  
    pass
```

składa się ona ze słowa kluczowego `class`, po którym występuje nazwa klasy (A). Instrukcja pusta (`pass`) w następnym wierszu jest konieczna ze względu na

fakt, że w Pythonie zakres bloku kodu jest określany przez wcięcia i w związku z tym nie jest możliwe umieszczenie w programie bloku pustego. Tak zdefiniowana klasa nie ma żadnych metod. Niemniej jednak można tworzyć obiekty do niej należące, które z kolei mogą mieć atrybuty (każdy obiekt inne). Poniższa sesja w interpreterze demonstruje tworzenie i zachowanie obiektu klasy A:

```
>>> a=A()
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: A instance has no attribute 'x'
>>> a.x=1
>>> a.x
1
>>>
```

Obiekt tworzy się poprzez “wywołanie” klasy tak, jakby była funkcją¹. Do składowych obiektu można się dostać przy pomocy operatora `..`. Pierwsza próba pobrania wartości atrybutu `x` kończy się niepowodzeniem (wyjątek `AttributeError`). Po wykonaniu przypisania, wartość atrybutu jest dostępna.

4.2 Atrybuty i metody

W Pythonie atrybuty obiektów działają podobnie jak zmienne. Tworzymy je przez pierwsze przypisanie. Nie jest zatem oczywiste w jaki sposób można umieścić w definicji klasy informację o odpowiedzialności obiektów za przechowywanie danych. Jest to cecha charakterystyczna języka Python².

Definicje metody, z kolei, umieszcza się w definicji klasy. Poniżej przedstawiono definicję klasy, która ma jedną bezargumentową metodę:

```
class Lew:
    def talk(self):
        print "Jestem lew"
```

Nie można nie zauważyć, że składnia definicji metody jest identyczna ze składnią definicji funkcji. Jediną różnicą jest to, że poprawnie zdefiniowana metoda musi przyjmować co najmniej jeden argument, który konwencja nakazuje nazywać `self`³. Poprzez ten argument do metody zostaje przekazany obiekt, dla którego została wywołana. Klasa `Lew` z przykładu posiada bezargumentową metodę `talk`, która akurat ignoruje również przekazywaną do niej instancję klasy `Lew`. Nie zmienia to jednak faktu, że metodę można wywołać wyłącznie dla obiektu. Poniżej przedstawiono instrukcje konieczne do utworzenia obiektu klasy `Lew` i wywołania metody `talk`.

```
>>> l=Lew()
>>> l.talk()
Jestem lew
>>>
```

¹W rzeczy samej, klasa jest obiektem wywoływalnym (ang. *callable*) tak, jak np. funkcja.

²I innych języków z typowaniem dynamicznym.

³Nie jest to obowiązkowe, aczkolwiek zalicza się do dobrej praktyki programowania.

Rozszerzymy teraz klasę `Lew` o kolejne metody:

```
class Lew:
    def talk(self):
        print "Jestem lew"

    def setHungry(self, val):
        self.hungry=val

    def talkMore(self):
        self.talk()
        if self.hungry:
            print "glodny lew"
```

Metoda `setHungry` kapsułkuje atrybut `hungry`. Warto zwrócić uwagę, że dostęp do atrybutu odbywa się poprzez argument `self`. Próba przypisania postaci:

```
hungry=val
```

spowoduje przypisanie na zmienną lokalną `hungry`. Metoda `talkMore` rozszerza funkcjonalność metody `talk`. Wykorzystuje atrybut `hungry` do określenia, czy lew ma ogłosić również informację o posiadanym apetycie. Poniższa sesja interpretera ilustruje działanie powyższych metod:

```
>>> l=Lew()
>>> l.talk()
Jestem lew
>>> l.setHungry(True)
>>> l.talkMore()
Jestem lew
glodny lew
>>>
```

Należy jeszcze zauważyć, że do atrybutu `hungry` można również dostawać się bezpośrednio bez korzystania z metody `setHungry`, oraz że na początku lew nie działa poprawnie:

```
>>> l=Lew()
>>> l.talkMore()
Jestem lew
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in talkMore
AttributeError: Lew instance has no attribute 'hungry'
```

4.3 Inicjalizacja obiektów

Nowo utworzony obiekt nie posiada żadnych atrybutów. W szczególności lew nie posiada atrybutu `hungry`, więc metoda `talkMore` próbując pobrać jego wartość powoduje wystąpienie wyjątku `AttributeError`. Do określenia początkowych wartości atrybutów służy metoda specjalna `__init__`. Jest ona wywoływana automatycznie po utworzeniu nowego obiektu i są do niej przekazywane argumenty, które pojawiły się w "wywołaniu" klasy tworzącym obiekt. Metoda

`__init__` w klasie `Lew`, której zadaniem byłoby utworzenie atrybutu `hungry` wygląda następująco:

```
def __init__(self):
    self.hungry=False
```

4.4 Dziedziczenie

Dziedziczenie jest niezwykle istotną w programowaniu obiektowym konstrukcją, która niewielkim kosztem pozwala tworzyć nowe klasy bez powielania definicji metod, które są współdzielone z innymi klasami. W języku Python klasę, z której nowodefiniowana klasa ma dziedziczyć podaje się w nawiasie następującym po nazwie klasy definiowanej. Poniżej przedstawiono kompletną (pozostałe metody włącznie z `__init__` są dziedziczone) definicję klasy `GroznyLew`:

```
class GroznyLew(Lew):
    def talkMore(self):
        self.talk()
        print "grozny lew,"
        print "spotkac mnie znaczy pech."
        if self.hungry:
            print "Wszystkich zjem, az do dna."
            print "Rety, lepiej nie spotkac lwa."
```

Obiekty klasy `GroznyLew` od obiektów klasy `Lew` odróżnia metoda `talkMore`, w której groźny lew szczegółowo się przedstawia. Poniżej przedstawiono działanie obiektu tej klasy:

```
>>> gl=GroznyLew()
>>> gl.talk()
Jestem lew
>>> gl.setHungry(True)
>>> gl.talkMore()
Jestem lew
grozny lew,
spotkac mnie znaczy pech.
Wszystkich zjem, az do dna.
Rety, lepiej nie spotkac lwa.
```

4.5 Ograniczenia dostępu

W Pythonie nie ma możliwości określania poziomu dostępności metod i atrybutów. Istnieje jedynie konwencja, której przestrzeganie nie jest w żaden sposób wymuszane przez interpreter. Składowe o nazwach zaczynających się od znaku `_` są uznawane za niedostępne publicznie i/lub zależne od implementacji. W szczególności nie należy zakładać, że w kolejnych wersjach programów pozostaną niezmienione.

Składowe o nazwach zaczynających się od znaków `__` są "prywatne" w specyficzny sposób. Występujący wewnątrz definicji klasy identyfikator postaci

`__ident` jest zamieniany na `_Klasa__ident`, gdzie `Klasa` jest nazwą definiowanej klasy. Poniższy program:

```
class Lew:
    ...

    def __talkMore(self):
        self.talk()
        if self.hungry:
            print "glodny_lew"

    def talkMore(self):
        self.__talkMore()

    def talkSafe(self):
        self.__talkMore()

class GroznyLew(Lew):
    def __talkMore(self):
        self.talk()
        print "grozny_lew,"
        print "spotkac_mnie_znaczy_pecz."
        if self.hungry:
            print "Wszystkich_zjem,_az_do_dna."
            print "Rety,_lepiej_nie_spotkac_lwa."

    def talkMore(self):
        self.__talkMore()

    def talkMean(self):
        self.__talkMore()
```

zostanie automatycznie zinterpretowany tak, jakby wyglądał następująco:

```
class Lew:
    ...

    def _Lew__talkMore(self):
        self.talk()
        if self.hungry:
            print "glodny_lew"

    def talkMore(self):
        self._Lew__talkMore()

    def talkSafe(self):
        self._Lew__talkMore()

class GroznyLew(Lew):
```

```

def _GroznyLew__talkMore(self):
    self.talk()
    print "grozny lew,"
    print "spotkac mnie znaczy pech."
    if self.hungry:
        print "Wszystkich zjem, az do dna."
        print "Rety, lepiej nie spotkac lwa."

def talkMore(self):
    self._GroznyLew__talkMore()

def talkMean(self):
    self._GroznyLew__talkMore()

```

Poniżej przedstawiono przykład jego działania:

```

>>> l=Lew()
>>> l.talkMore()
Jestem lew
glodny lew
>>> l.talkSafe()
Jestem lew
glodny lew
>>> gl=GroznyLew()
>>> gl.talkMore()
Jestem lew
grozny lew,
spotkac mnie znaczy pech.
Wszystkich zjem, az do dna.
Rety, lepiej nie spotkac lwa.
>>> gl.talkSafe()
Jestem lew
glodny lew
>>> gl.talkMean()
Jestem lew
grozny lew,
spotkac mnie znaczy pech.
Wszystkich zjem, az do dna.
Rety, lepiej nie spotkac lwa.
>>>

```

Warto zauważyć, że gdyby nazwa metody `__talkMore` była poprzedzona jednym znakiem `_`, metoda `talkSafe` w klasie `GroznyLew` działałaby inaczej. W takim przypadku, wywoływana byłaby metoda `_talkMore` należąca do klasy obiektu, dla którego metoda `talkSafe` jest wywoływana (czyli `GroznyLew._talkMore`), zamiast metody z klasy, w której metoda `talkSafe` jest zdefiniowana.

4.6 Duck typing

W języku Python obowiązuje typowanie dynamiczne oraz mechanizm *duck typing*. Programując nie określa się typów zmiennych (oraz atrybutów obiektów i

argumentów funkcji). Weryfikacja zgodności typu przekazanej wartości z oczekiwanym następuje w momencie próby jej użycia. Jeżeli się to powiedzie, czyli np. przekazany obiekt ma atrybut albo metodę o określonej nazwie i liczbie argumentów, to niezależnie od jego typu wszystko jest w porządku. W przeciwnym wypadku następuje błąd. Takie działanie można przyrównać do skrajnie fenomenologicznego podejścia streszczonego zdaniem:

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Duck typing posiada szereg zalet. Między innymi:

- Można używać w jednym kontekście obiektów niemających wspólnej nadklasy zawierającej wymagane składowe.
- Nie trzeba stosować rzutowań, interfejsów, typów generycznych ani wzorców.
- Projektowanie jest znacznie łatwiejsze.

Niestety wiąże się z nim również fundamentalna wada, do której trzeba się przyzwyczaić: Proste błędy (np. literówki), które mogłyby zostać wykryte na etapie kompilacji, zostaną odnalezione dopiero w momencie próby wykonania instrukcji, w której występują. Ścisły system typów również skutecznie zabezpiecza przed podawaniem niewłaściwych argumentów do funkcji lub metod.

4.7 Bardzo szczególne cechy Pythona

Omówione powyżej zagadnienia występują w wielu językach obiektowych, albo są (jak prefiks `__` przed nazwą metody) charakterystycznymi dla Pythona rozwiązaniami typowych problemów. W tym podrozdziale zwrócimy uwagę na kilka własności tego języka, które są dla niego bardzo charakterystyczne i raczej niespotykane w innych językach obiektowych.

W większości języków programowania w klasie wyróżnia się metodę odpowiedzialną za tworzenie obiektów tej klasy. Jest ona nazywana *konstruktorem*. Opisana powyżej metoda `__init__` nie jest konstruktorem w ścisłym tego słowa znaczeniu. W momencie jej wywołania tworzony obiekt już istnieje (i jest do niej przekazywany przez argument `self`). Metoda `__init__` odpowiada wyłącznie za jego inicjalizację, dlatego najlepiej nazywać ją *inicjalizatorem*. Python nie posiada, żadnego wsparcia dla automatycznego wywoływania inicjalizatorów nadklas. Jeżeli zachodzi taka potrzeba, należy wprost wywołać inicjalizator nadklasy w inicjalizatorze podklasy:

```
class GroznyLew(Lew):
    def __init__(self):
        self.angry = True
        Lew.__init__(self)
```

W powyższym przykładzie widać, w jaki sposób można uzyskać dostęp do metod nadklasy, które są przesłonięte przez podklasy. W tym zakresie metoda przypomina funkcję. Można ją pobrać z klasy jak każdą inną składową. Trzeba jednak wówczas pamiętać o przekazaniu obiektu, dla którego ma być wykonana,

jako pierwszego argumentu. Jeżeli `a` jest obiektem klasy `A`, które ma metodę `m`, to poniższe instrukcje są równoważne i skutkują wywołaniem `m` na obiekcie `a`:

```
a.m()
A.m(a)
```

Podobnie w przypadku lwów:

```
>>> l=Lew()
>>> l.talkMore()
Jestem lew
>>> Lew.talkMore(l)
Jestem lew
>>> Lew.setHungry(l, True)
>>> Lew.talkMore(l)
Jestem lew
glodny lew
>>>
```

Python pozwala również na posługiwanie się w definicji funkcji domyślnymi wartościami argumentów np.:

```
def __init__(self, hungry=True):
    ...
```

oraz podawaniem nazw argumentów w wywołaniu funkcji, co umożliwia podawanie ich w kolejności innej niż w definicji funkcji i/lub pomijanie argumentów, dla których ma zostać przyjęta ich wartość domyślna. Powyższy przykład pochodzi z dokumentacji autorów języka, gdzie to zagadnienie jest szeroko omówione:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

```
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='V00000M') # 2 keyword args
parrot(action='V00000M', voltage=1000000) # 2 keyword args
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 key
```

Rozdział 5

Powiązania i tworzenie obiektów

Można wyróżnić dwa rodzaje odpowiedzialności obiektów za pamiętanie informacji. Pierwszy, to wartości skalarne (pojedyncze) i ewentualnie wektory (tablice, listy) typów prostych. W tym przypadku nie ma żadnych wątpliwości, w jaki sposób należy zaplanować ich przechowywanie. Drugim rodzajem są wartości reprezentowane przez obiekty lub ich zbiory. Są one z reguły odnotowane w obiektowym modelu dziedziny i w projekcie jako powiązania. Można je zakwalifikować do jednej z trzech kategorii:

- jeden do jednego
- jeden do wielu
- wiele do wielu

Najłatwiejsze to zaimplementowania są powiązania typu “jeden do jednego”. Można je łatwo zrealizować przy pomocy atrybutu. Trzeba jedynie zdecydować, w której z powiązanych klas ten atrybut powinien być umieszczony. Można to rozstrzygnąć kierując się zasadą, że musi być zachowana możliwość dojścia do każdego obiektu w systemie. Jedynie część obiektów jest dostępna bezpośrednio poprzez zmienne. Dostęp do pozostałych odbywa się poprzez przechodzenie sieci powiązań. Istnienie obiektów, do których nie ma możliwości dostępu, mija się z celem.

Podobna sytuacja zachodzi w przypadku powiązań typu “jeden do wielu”. Można je zaimplementować przy pomocy pojedynczego atrybutu po stronie “wielu”, lub listy, tablicy lub zbioru po stronie “jeden”. Pierwszy wariant wydaje się znacznie łatwiejszy, ale nie zawsze jest stosowny. Przykładowo pomiędzy klasami `Faktura` i `PozycjaFaktury` zachodzi powiązanie typu “jeden do wielu” – faktura może mieć wiele pozycji; pozycja należy tylko do jednej faktury. Gdyby to powiązanie było zrealizowane przy pomocy atrybutu `faktura` w klasie `PozycjaFaktury`, najbardziej naturalny przypadek zastosowania klasy `Faktura`, jakim jest odczytanie jej treści, byłby bardzo trudny do zrealizowania. Dodatkowo należałoby zapewnić inny sposób dostania się do instancji klasy `PozycjaFaktury` np. poprzez utrzymywanie listy wszystkich pozycji występujących w systemie.

Odwzorowanie typu “wiele do wielu” można realizować podobnie jak “jeden do wielu” przy pomocy list (lub innych podobnych typów) przypisanych do obiektu i przechowujących informację z jakimi ten jest powiązany. Można również zauważyć, że najbardziej ogólnym przypadkiem powiązania “wiele do wielu” jest graf skierowany i zastosować jedną ze znanych metod reprezentowania grafów (czyli listy sąsiadów albo macierz sąsiedztwa).

5.1 Implementacja powiązań

Czasem nawet w przypadku pozornie łatwego związku “jeden do jednego” trudno jest podjąć decyzję, w której klasie warto umieścić atrybut. Rozważmy klasy `Man` i `Woman`, których obiekty mogą być powiązane, jeżeli osoby im odpowiadające znajdują się w związku małżeńskim. Fakt ten można odnotowywać przy pomocy atrybutu `wife` w klasie `Man`, lub atrybutu `husband` w klasie `Wife`. Oba te rozwiązania są całkowicie równoważne i wybór któregośkolwiek z nich będzie skutkować trudnym dostępem do powiązanego obiektu w kierunku przeciwnym. W takiej sytuacji naturalne wydaje się utrzymywanie atrybutów w obydwu klasach. Niestety przy takim rozwiązaniu informacja o powiązaniu obiektów byłaby przechowywana w dwóch miejscach, co wiąże się z ryzykiem rozspójnienia i pojawienia się informacji sprzecznych. Pewną ochronę może stanowić kapsułkowanie atrybutów. Rozważmy poniższe implementacje metod `set_wife` i `set_husband`:

```
class Man:
    ...

    def set_wife(self, wife):
        self.wife = wife
        wife.set_husband(self)

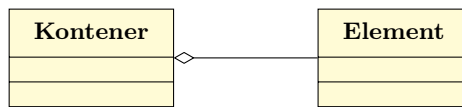
class Woman:
    ...

    def set_husband(self, husband):
        self.husband = husband
        husband.set_wife(self)
```

Metody `set_husband` i `set_wife` służą do aktualizacji odpowiadających im atrybutów i równocześnie zapewniają aktualizację atrybutu w obiekcie po drugiej stronie powiązania. Niestety przedstawiona implementacja ma poważny błąd. Wywołanie którejkolwiek z metod prowadzi do wystąpienia nieskończonego łańcucha wywołań rekurencyjnych. Poniżej przedstawiono rozwiązanie poprawne:

```
class Man:
    ...

    def set_wife(self, wife):
        if(self.wife == wife):
            return
```



Rysunek 5.1: Powiązanie Kontener-Element

```

self.wife.set_husband(None)
self.wife = wife
if wife is not None:
    wife.set_husband(self)
  
```

Pierwszy warunek służy sprawdzeniu, czy wartość atrybutu wymaga zmiany i w ten sposób przerywa rekursję. Dodatkowo “poprzedni” mąż jest “pozbawiany” żony. Poprawnie jest również obsługiwany przypadek ustawiania wartości atrybutu na `None`. Widać, że implementując kapsułkowanie, które pociąga za sobą kaskadę aktualizacji innych atrybutów, należy zachować szczególną staranność. Trzeba dbać, ażeby nie pojawił się nieskończony ciąg wywołań rekurencyjnych oraz poprawnie obsługiwać wartości nietypowe (jak `None`).

W przypadku powiązań “jeden do wielu” kapsułkowanie pomaga ukryć techniczne szczegóły związane z rodzajem struktury danych zastosowanej do przechowywania zbioru powiązanych obiektów. Implementując klasę `Mother`, której obiekty mogą być powiązane z dowolną liczbą dzieci, zamiast dawać bezpośredni dostęp do atrybutu `_children`, lepiej zaimplementować metody `add_child` i `remove_child`:

```

class Mother:
    def __init__(self):
        self._children = []

    def add_child(self, child):
        self._children.append(child)

    def remove_child(self, child):
        self._children.remove(child)
  
```

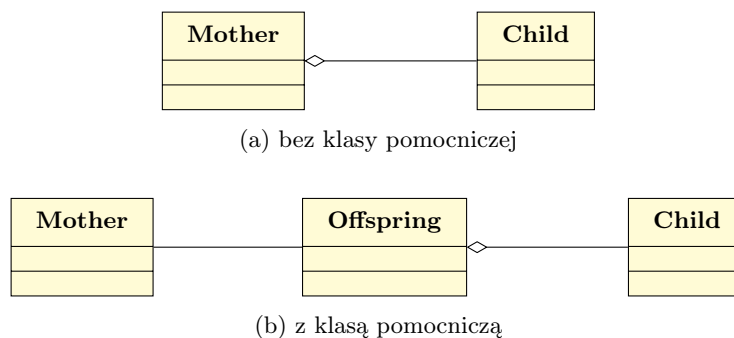
Szczególnym przypadkiem powiązania typu “jeden do wielu” jest zawieranie się. Obiekty, które mogą zawierać inne, nazywamy *kontenerami*. Powiązanie będące zawieraniem oznaczamy na diagramie przy pomocy rombu (rys. 5.1).

Jeżeli rozszerzanie klasy o funkcjonalność kontenera (rys. 5.2a) jest niekorzystne, można zastosować klasę pomocniczą powiązaną “jeden do jednego” (rys. 5.2b).

W Pythonie wszystkie obiekty typów sekwencyjnych są kontenerami.

5.2 Wzorzec projektowy *Composite pattern*

Composite pattern jest wzorcem projektowym stosowanym w przypadku hierarchii obiektów jednego rodzaju, które mogą się w sobie zawierać. Kanonicznym przykładem są tutaj wszelkiego rodzaju wyrażenia arytmetyczne albo logiczne; elementy urządzenia, które mogą składać się z mniejszych części itp. W tym wzorcu występują co najmniej trzy klasy:



Rysunek 5.2: Kontenery

- **Component** – odpowiada dowolnemu elementowi
- **Leaf** – odpowiada elementom niepodzielnym
- **Composite** – odpowiada elementom składającym się z innych elementów klasy **Component**

Na rysunku 5.3 przedstawiono schemat zależności pomiędzy klasami. Metoda `operation` reprezentuje funkcjonalność charakterystyczną dla wszystkich obiektów, które są implementowane. Dodatkowo klasa **Composite** posiada metody pozwalające na dostęp do komponentów, z których się składa (które zawiera).

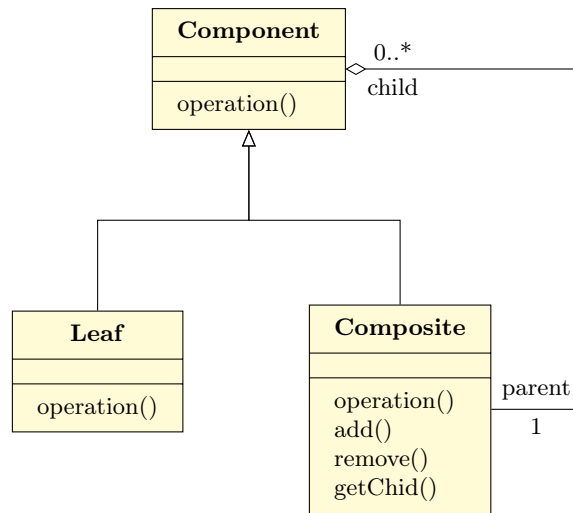
5.3 Iteratory

W przypadku wszelkiego rodzaju obiektów, które mogą zawierać inne (kontenerów), konieczne jest zapewnienie dobrego dostępu do ich zawartości. Najlepiej jest, aby dostęp odbywał się w sposób niezależny od wewnętrznej organizacji kontenera. Można wówczas wymieniać implementacje konterera bez zmiany sposobu dostępu do zawartości. Obiekty, które pozwalają sekwencyjnie przeglądać zawartość konterera, nazywamy *Iteratorami*.

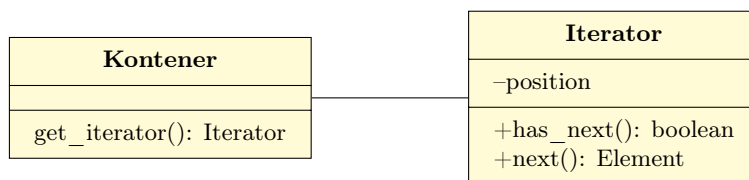
Rysunek 5.4 przedstawia schemat wzorca projektowego *Iterator*. Klasa **Kontener** posiada metodę `get_iterator`, która tworzy i zwraca nowy iterator dla danego kontenera. Iterator pozwala pobierać kolejne elementy z konterera przy pomocy metody `next`. Metoda `has_next` z kolei pozwala sprawdzić, czy istnieją jeszcze nieodwiedzone elementy. Dodatkowo iterator pamięta, który element został ostatnio pobrany, przy pomocy prywatnego atrybutu `position`. Dlatego każdy element konterera może zostać odwiedzony dokładnie raz. Równocześnie może istnieć dowolnie wiele iteratorów dla danego kontenera. Iterator, który nie ma kolejnych elementów (czyli wszystkie już odwiedził), jest “zużyty”.

Python pozwala definiować specjalne metody, które pozwalają na tworzenie iterowalnych obiektów w taki sposób, żeby mogły być używane tak samo jak obiekty typów iterowalnych wbudowanych w język (np. w instrukcji `for`). Jeżeli obiekty danej klasy mają być iterowalne, należy w niej zdefiniować metodę `__iter__`, która zwraca nową instancję iteratora.

Klasa **Przedzial** reprezentuje przedział liczb naturalnych o określonych granicach (atrybuty `od` i `do`). Metoda `__iter__` tworzy i zwraca nową instancję klasy **PrzedzialIter**, która jest iteratorem dla klasy **Przedzial**:



Rysunek 5.3: Schemat wzorca *Composite*



Rysunek 5.4: Schemat wzorca *Iterator*.

```

class Przedzial:
    def __init__(self, od, do):
        self.od = od
        self.do = do

    def __iter__(self):
        return PrzedzialIter(self.od, self.do)

```

Klasa `PrzedzialIter` ma metodę `next`, która zwraca kolejną liczbę z przedziału. Aktualna liczba jest przechowywana w atrybucie `pos`. W momencie dojścia do końca przedziału zgodnie z wymaganiami Pythona podnoszony jest wyjątek `StopIteration`. Dodatkowo iteratory muszą same być iterowalne (posiadać metodę `__iter__`), która nie tworzy nowego iteratora lecz zwraca ten, dla którego została wywołana:

```

class PrzedzialIter:
    def __init__(self, od, do):
        self.pos = od
        self.do = do

    def next(self):
        if self.pos <= self.do:
            self.pos += 1
            return self.pos - 1
        else:
            raise StopIteration

    def __iter__(self):
        return self

```

Pozornie obiekt klasy `Przedzial` nie różni się od swojego iteratora:

```

>>> list(PrzedzialIter(1,3))
[1, 2, 3]
>>> list(Przedzial(1,3))
[1, 2, 3]
>>> przedzial = Przedzial(1,3)
>>> for i in przedzial:
...     print i
...
1
2
3

```

Sytuacja zmienia się, gdy próbujemy użyć tego samego iteratora w kilku kontekstach równocześnie. Poniżej przedstawiono dwie zagnieżdżone pętle po obiekcie klasy `Przedzial`, które posiadają (automatycznie tworzone) dwa niezależne iteratory i działają zgodnie z intuicyjnym oczekiwaniem:

```

>>> for i in 1:
...     for j in 1:
...         print i,j
...

```

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Gdyby podobna operacja miała zostać przeprowadzona na obiekcie klasy `PrzedzialIter` wynik byłby diametralnie różny:

```
>>> przedzaliter = przedzial.__iter__()
>>> for i in przedzaliter:
...     print i
...
1
2
3
>>> for i in przedzaliter:      # tutaj iterator jest juz wyczerpany
...     print i
...
>>> przedzaliter = przedzial.__iter__()
>>> for i in przedzaliter:
...     for j in przedzaliter:
...         print i,j
...
1 2
1 3
>>>
```

5.4 Tworzenie obiektów

Standardowo obiekty tworzymy wywołując konstruktor:

```
>>> Lew('Ziutek')
<__main__.Lew instance at 0x10edfec20>
```

Nie zawsze takie podejście jest wystarczająco elastyczne. Na przykład jeżeli:

- są różne algorytmy tworzenia/inicjalizacji obiektu,
- nie wiadomo jakiej klasy obiekt chcemy stworzyć,
- chcemy parametry inicjalizacji podawać na raty (np. wczytując je z pliku),
- chcemy obiekty ewidencjonować.

Wymienionych operacji nie można wykonać w metodzie `__init__`, ponieważ obiekt jest już wtedy utworzony (więc nie można podjąć decyzji, do jakiej klasy ma należeć) i jest ona wywoływana raz (podawanie argumentów na raty nie wchodzi w rachubę). Z poziomu metody `__init__` trudno również byłoby dostać

się do obiektu, który miałby nowoutworzony obiekt w jakiś dodatkowy sposób przetworzyć. Istnieje kilka użytecznych wzorców projektowych, które w takich przypadkach można zastosować.

5.4.1 Wzorce projektowe *Factory* i *Factory method*

Jeżeli z jakiegoś powodu nie da się wskazać wprost klasy obiektu, który ma zostać utworzony, można zdefiniować metodę (lub funkcję), która na podstawie przekazanych parametrów dokonuje wyboru klasy i tworzy odpowiedni obiekt. Klasę, która zawiera taką metodę, będziemy nazywać fabryką (ang. *factory*). Przykładem może być funkcja tworząca obiekty reprezentujące figury geometryczne na podstawie listy wierzchołków:

```
def makeFigure(points):
    if len(points) == 1:
        return Point(points[0])
    elif len(points) == 2:
        return Segment(points[0], points[1])
    elif len(points) == 3:
        return Triangle(points[0], points[1], points[2])
    else:
        return Polygon(*points)
```

W pewnych przypadkach wybór klasy tworzonego obiektu może następować wskutek wywołania metody tworzącej w odpowiedniej fabryce. Jeżeli wykorzystuje się do tego dziedziczenie, mamy do czynienia ze wzorcem projektowym *Factory method*.

```
class MazeGame:
    def __init__(self):
        room1 = self.make_room()
        room2 = self.make_room()
        room1.connect(room2)
        self.add_room(room1)
        self.add_room(room2)

    def make_room(self):
        return OrdinaryRoom()

class MagicMazeGame(MazeGame):
    def make_room(self):
        return MagicRoom()
```

W powyższym przykładzie występują klasy odpowiadające labiryntowi zwykłemu (*MazeGame*) i magicznemu (*MagicMazeGame*). W obydwu przypadkach początkowy labirynt składa się z dwóch połączonych pomieszczeń. Jednakże w zwykłym labiryntcie występują zwykłe pomieszczenia (*OrdinaryRoom*), a w magicznym magiczne (*MagicRoom*). Metoda `__init__`, która odpowiada za tworzenie labiryntu, jest wspólna dla obu klas¹. Jest to możliwe ponieważ, aby utworzyć nowe pomieszczenie, wywołuje ona metodę `make_room`, która tworzy obiekt odpowiedniej klasy.

¹Klasa *MagicMazeGame* dziedziczy ją z klasy *MazeGame*.

5.4.2 Wzorzec projektowy *Builder*

Nie zawsze łatwo jest zdefiniować konstruktor/inicjalizator/metodę tworzącą obiekty ze względu na złożoność i różnorodność argumentów. W takiej sytuacji opłaca się tworzyć obiekt “na raty”. Taki efekt można osiągnąć tworząc obiekt pośredniczący, który gromadzi dane o obiekcie, który ma zostać utworzony, a następnie po wywołaniu odpowiedniej metody tworzy ten obiekt i go zwraca. Rozwiązanie to może mieć szereg zastosowań:

- liczba argumentów do inicjalizatora jest zbyt wielka
- konieczne jest tworzenie wielu obiektów na podstawie tych samych parametrów
- nie jest możliwe posiadanie wszystkich argumentów w tym samym momencie (bo są wczytywane lub obliczane na bieżąco)

Przykładem może być klasa `CarBuilder` tworząca obiekty klasy `Car`. Do ustawiania konkretnych cech tworzonego samochodu służą metody o nazwach zaczynających się od prefiksów `set` lub `unset`:

```
class Car:
# Can have GPS, trip computer and a various number
# of seats.
# Can be a city car, a sport car or a cabriolet.

class CarBuilder:
    def getResult(self):
#         output: a Car with the right options
# Construct and return the car.

    def setSeats(self, number):
#         input: the number of seats the car may have.
# Tell the builder the number of seats.

    def setCityCar(self):
        ...

    def setCabriolet():
        ...

    def setSportCar():
        ...

    def setTripComputer():
        ...

    def unsetTripComputer():
        ...

    def setGPS():
        ...
```

```
def unsetGPS():  
    ...
```

Klasy `CarBuilder` można używać w następujący sposób:

```
carBuilder = CarBuilder()  
carBuilder.setSeaters(2)  
carBuilder.setSportCar()  
carBuilder.setTripComputer()  
carBuilder.unsetGPS()  
car1 = carBuilder.getResult()  
carBuilder.setGPS()  
car2 = carBuilder.getResult()
```

5.4.3 Inne przydatne wzorce projektowe

Istnieje wiele innych wzorców projektowych związanych z tworzeniem obiektów. Zaliczają się do nich:

- *Abstract Factory* – Definiuje się wiele fabryk służących do tworzenia rodzin obiektów, które dziedziczą z tej samej klasy. W zależności od kontekstu używa się jednej z nich.
- *Leniwa inicjalizacja* – Obiekt lub jego fragmenty są tworzone dopiero w momencie pierwszego użycia.
- *Singleton* – Istnieje dokładnie jedna instancja danej klasy. Jeżeli klasy są obiektami to można je traktować jak singletony.
- *Multiton* – Istnieje ściśle określona liczba instancji danej klasy, zazwyczaj dostępnych za pośrednictwem słownika.

Rozdział 6

Metody i dziedziczenie

6.1 Metody

W rozdziale 4 powiedzieliśmy, że obiekty odpowiadają za przechowywanie informacji, a klasy za operacje na tych danych, które realizowane są przy pomocy metod. Mówiliśmy też, że można z pewnym uproszczeniem przyjąć, że metoda¹ to funkcja zdefiniowana w klasie i operująca na obiekcie, dla którego została wywołana (i dodatkowych argumentach).

Zasadniczą różnicą pomiędzy metodą a funkcją jest to, że metoda jest za pośrednictwem klasy przypisana do obiektu. Zatem pod tą samą nazwą mogą kryć się różne metody, w zależności od obiektu (właściwie jego klasy), dla którego są wywoływane. Metody definiuje się, aby zaimplementować funkcjonalności, do realizacji których niezbędne są dane zawarte w obiekcie, dla którego metoda miałaby być wykonywana. Do standardowych funkcjonalności metod należą:

- pobieranie i aktualizacja właściwości obiektu (kapsułkowanie),
- pobieranie przetworzonych właściwości obiektu,
- zmiana właściwości w wyniku obliczenia,
- obliczenie funkcji z dodatkowymi argumentami wynikającymi ze stanu obiektu,
- wykonanie operacji na innym obiekcie,
- kaskada wywołań/zdarzeń.

Najprostszym rodzajem operacji na właściwościach jest kapsułkowanie, czyli pobieranie i ustawianie wartości atrybutu przy pomocy metod. Na rysunku 6.1 przykładami takich metod są `set_side` i `get_side` w klasie `Square` oraz `set_abc` i `get_abc` w klasie `Quadratic`. Warto zwrócić uwagę, że metody kapsułkujące w klasie `Quadratic` opakowują równocześnie trzy atrybuty. Metody kapsułkujące oprócz aktualizacji wartości atrybutów mogą wykonywać dodatkowe czynności takie jak weryfikacja podanych wartości albo aktualizacja innych obiektów².

¹W niektórych językach programowania używa się pojęcia *komunikat*.

²Patrz klasy `Man` i `Woman` w rozdziale 5.1.

Square	Quadratic
-side -position	-a,b,c
+set_side(side) +get_side() +get_area() +get_perimeter() +covers(x,y)	+set_abc(a,b,c) +get_abc() +get_solutions() +intersects(curve)

Rysunek 6.1: Przykładowe metody.

Metody kapsułkujące mogą również służyć do ukrycia faktu, że atrybut do którego rzekomo się odwołują, w rzeczywistości nie istnieje. Na rysunku 6.1 taką własność mają metody `get_area`, `get_perimeter` i `get_solutions`. Pole i obwód kwadratu są łatwe do obliczenia, jeżeli znana jest długość jego boku. Nie ma zatem żadnego racjonalnego powodu, aby te wartości przechowywać. Wiązałoby się to wręcz z kłopotliwą komplikacją, albowiem przy każdej aktualizacji długości boku należałoby je na nowo obliczyć. Jednakże taki szczegół implementacyjny jest zupełnie nieistotny dla programisty posługującego się klasą `Square`. Z jego punktu widzenia obwód i pole powierzchni są takimi samymi właściwościami kwadratu jak bok z zastrzeżeniem, że nie można ich zmieniać.

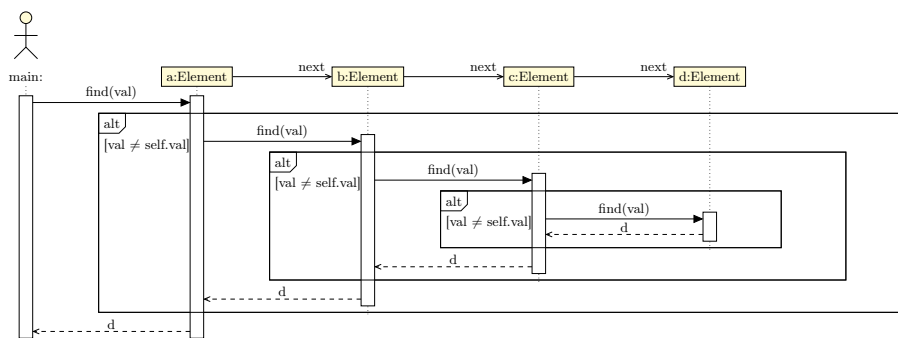
Dalszym naturalnym rozszerzeniem jest sytuacja, w której do obliczenia wartości poza atrybutami obiektu potrzebne są dodatkowe argumenty. Przykładem tutaj jest metoda `covers` w klasie `Square`, która rozstrzyga czy punkt o podanych współrzędnych jest pokryty przez kwadrat, albo metoda `intersects` w klasie `Quadratic`, która rozstrzyga czy parabola przecina podaną krzywą. Oczywiście jest, że takie metody poza dodatkowymi argumentami potrzebują również obiektu (kwadratu lub paraboli). Równocześnie jasne jest, że umieszczenie ich w klasie jest wygodniejsze niż definiowanie poza klasą funkcji, która przyjmowałaby zarówno obiekt i dodatkowe argumenty.

Skutkiem działania metody mogą być nie tylko zmiany atrybutów obiektu, dla którego została wywołana. Metody mogą również wykonywać operacje na innych obiektach poprzez zmienianie wartości ich atrybutów albo wywoływanie metod. Przykładowo w klasie `Treser` można zdefiniować metodę `nakarm`, której skutkiem będzie nakarmienie zadanego lwa:

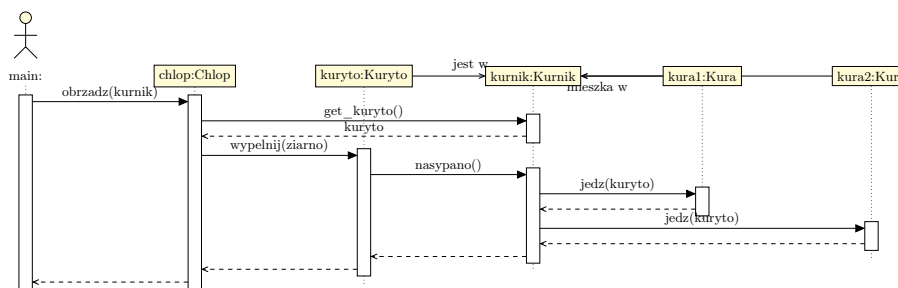
```
class Treser:
    def nakarm(self, lew):
        pasza=self.przygotuj_pasze()
        lew.jedz(pasza)

class Lew:
    def jedz(self, pasza):
        print "Mniam, □pyszna□", pasza
```

Może się wreszcie zdarzyć, że metoda wywołuje następną metodę, która wywołuje kolejną itd. Powstaje w ten sposób tzw. *kaskada wywołań*. W szczególności metoda może wywoływać samą siebie tak, jak ma to miejsce w przypadku funkcji rekurencyjnych. Rysunek 6.2 ilustruje działanie metody `find` zdefiniowanej



Rysunek 6.2: Rekurencyjne przeszukiwanie listy



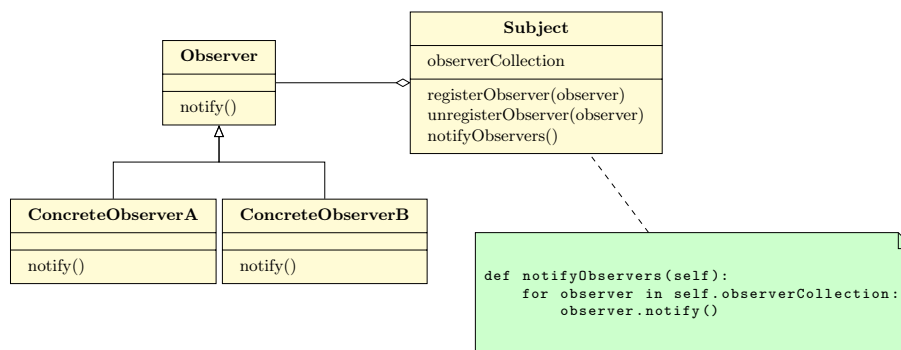
Rysunek 6.3: Rekurencyjne przeszukiwanie listy

w klasie `Element`, której obiekty tworzą łańcuch (listę jednokierunkową). Wyszukiwanie elementu o zadanej wartości atrybutu `val` w łańcuchu polega na sprawdzeniu, czy pierwszy element łańcucha jest poszukiwanym i, jeżeli nie, przeszukaniu łańcucha zaczynającego się od elementu następującego po pierwszym. Projektując taką metodę rekurencyjną postępuje się podobnie jak w przypadku funkcji rekurencyjnej. Koniecznie trzeba zadbać o zakończenie rekursji. W przedstawionym przykładzie wyszukiwanie będzie działać dopóki elementy nie utworzą cyklu, czyli dopóki pierwszy element łańcucha nie jest następnikiem ostatniego.

Znacznie częściej w kaskadzie wywołań występują różne metody. Rysunek 6.2 przedstawia kaskadę mającą miejsce podczas obrządzania kurnika. W klasie `Chłop` jest zdefiniowana metoda `obrzadz`, której argumentem jest kurnik (obiekt klasy `Kurnik`). Z kurnika chłop pobiera (metoda `get_kuryto`) naczynie, z którego jedzą kury (obiekt klasy `Kuryto`). Następnie wypełnia je ziarnem (metoda `wypełnij` w klasie `Kuryto`). Wówczas kuryto informuje kurnik (metoda `nasypano` w klasie `Kurnik`), że zostało napełnione. Kurnik z kolei powiadamia mieszkające w nim kury (metoda `jedz` w klasie `Kura`).

6.2 Wzorzec projektory *Observer*

Przedstawiony na rysunku 6.3 schemat jest prosty ale niestety nienaturalny. W świecie rzeczywistym kurnik, ani kuryto nie będąc przedmiotami nieożywionymi nie inicjują działań podejmowanych przez kury. To kury obserwują koryto i w



Rysunek 6.4: Schemat wzorca *Observer*

odpowiednim momencie przystępują do jedzenia.

Wzorec projektowy *Observer* (rys. 6.4) służy do odwzorowania takiego działania w systemie obiektowym. Występują w nim dwie klasy: obserwowana (*Subject*) i obserwująca (*Observer*). Klasa *Subject* pozwala na rejestrowanie obiektów (instancji klasy *Observer*), które obserwują jej stan i powinny być informowane o jego zmianie. W momencie, gdy następuje zmiana stanu obiektu klasy *Subject* wywoływana jest metoda `notifyObservers`. Żeby to było możliwe konieczne jest kapsułkowanie atrybutów, które są obserwowane. Metoda `notifyObservers` wywołuje metodę `notify` wszystkich obiektów, które należą do zbioru `observerCollection`. Ten atrybut jest kapsułkowany przez metody `registerObserver` i `unregisterObserver`.

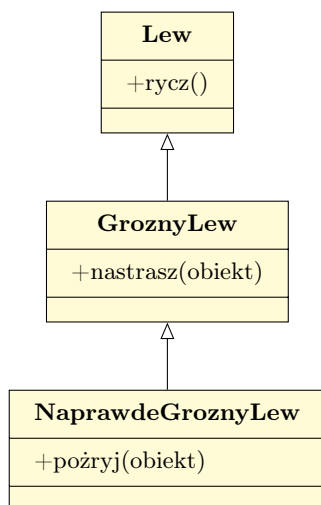
Oczywiście możliwe jest zdefiniowanie wielu różnych klas obiektów obserwujących. W zależności od języka programowania muszą one dziedziczyć z jednej klasy nadrzędnej albo (jak w Pythonie) po prostu mieć metodę `notify`. Dodatkowo obserwowany obiekt wywołując metodę `notify` może do niej przekazać argumenty niosące informację o naturze zmiany.

6.3 Dziedziczenie

Jak już kilkakrotnie wspominaliśmy, dziedziczenie jest fundamentalnym mechanizmem w projektowaniu obiektowym. Dzięki niemu można zaoszczędzić wiele wysiłku związanego z tworzeniem i utrzymaniem programu, albowiem funkcjonalności wspólne dla kilku klas wystarczy zaimplementować tylko raz – w klasie, z której dziedziczą. Ponadto dziedziczenie pozwala na przesłanianie metod w podklasach. W systemie występuje wtedy wiele metod o tej samej nazwie (i być może przeznaczeniu), które jednakowoż różnią się swoim działaniem. Właściwa metoda zostanie dobrana na podstawie klasy obiektu, dla którego została wywołana.

Dziedziczenie stosuje się w trzech przypadkach:

- rozszerzenie istniejącej klasy o nowe funkcjonalności (uszczegółowienie),
- grupowanie klas o wspólnych funkcjonalnościach,
- oznaczenie przynależności do wspólnej, nadrzędnej kategorii.



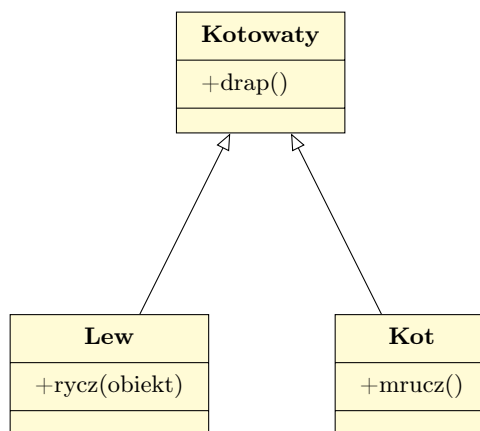
Rysunek 6.5: Przykład rozszerzania klasy o nowe funkcjonalności.

Przykład rozszerzania funkcjonalności zaprezentowano na rys. 6.5. Lew umie ryczeć. Groźny lew umie to samo co zwykły lew i dodatkowo potrafi nastraszyć. Naprawdę groźny lew potrafi to samo co groźny lew (ryczeć i straszyć) oraz pożerać.

Częstą praktyką jest rozszerzanie funkcjonalności klas pochodzących z różnego rodzaju bibliotek i modułów, które w ten sposób mogą zostać dostosowane do konkretnej potrzeby.

Grupowanie klas o współdzielonych funkcjonalnościach pozwala na zmniejszenie rozmiaru programu i usunięcie powtarzających się fragmentów. Na rysunku 6.6 przedstawiono klasy `Lew` i `Kot`. Mimo istotnych różnic w funkcjonalności (lew ryczy, a kot mruczy) obydwa zwierzęta potrafią podrapać. Jest to funkcjonalność wspólna dla wszystkich zwierząt należących do kotowatych. Zdefiniowanie klasy `Kotowaty` pozwoli na umieszczenie w niej metody `drap` i pozbycie się jej z klas, które z niej dziedziczą. Oczywiście jest to możliwe wyłącznie pod warunkiem, że lew i kot drapią tak samo. Doświadczenie podpowiada, że obiekty należące do klasy `Kotowaty`, które równocześnie nie należą do żadnej z jej podklas, nie powinny móc istnieć w systemie. Dlatego `Kotowaty` jest przykładem klasy, która nie znajdzie się w obiektywnym modelu dziedziny, a w projekcie zostanie umieszczona ze względów technicznych. Sam fakt, że nie zamierzamy tworzyć obiektów należących bezpośrednio do danej klasy, nie powinien budzić zaniepokojenia. Jak widać, takie klasy mogą być użyteczne.

Rozważmy jeszcze sytuację, w której mamy do czynienia z klasami należącymi do jednej kategorii i mającymi różniące się metody o tej samej nazwie. Na rysunku 6.7 przedstawiono klasy `Sledz` i `Mo1`, które mają metodę `zeruj`. Najprawdopodobniej te metody zasadniczo się od siebie różnią działaniem oraz typem pobieranego argumentu. Niemniej jednak można zdefiniować klasę `Zwierze` z metodą `zeruj`. Wprawdzie metoda `zeruj` w klasie `Zwierze` nie może mieć żadnej użytecznej funkcjonalności, ale fakt jej występowania niesie informację, że w każdej podklasie należy zdefiniować metodę o tej nazwie. Innymi słowy każde zwierzę żeruje. Metodę bez implementacji, która musi być przesłonięta w każdej



Rysunek 6.6: Przykład grupowania ze względu na funkcjonalności

podklase oraz klasę zawierającą takie metody nazywamy *abstrakcyjną*³.

W językach programowania z typowaniem statycznym (C++, Java) stosowanie klas abstrakcyjnych jest konieczne, jeżeli ma istnieć możliwość operowania w tym samym kontekście na obiektach różnych klas, które nie mają naturalnej (obejmującej wspólną funkcjonalność) nadklasy. Dzieje się tak, ponieważ dla każdej zmiennej i każdego argumentu należy z góry określić jej typ. Jeżeli jest to obiekt, trzeba podać jego klasę, która musi zawierać wszystkie potrzebne składowe, nawet jeżeli w rzeczywistości zawsze będą występowały obiekty podklas z własnymi definicjami tych składowych.

W przypadku typowania dynamicznego (duck-typing) (Python, Smalltalk) klasy abstrakcyjne nie są konieczne, ponieważ dowolne obiekty mogą występować we wszystkich kontekstach. Jeżeli nie posiadają wymaganych składowych, zostanie to wykryte dopiero podczas wykonania programu. Jednakowoż stosowanie klas abstrakcyjnych ułatwia projektowanie i późniejsze utrzymanie kodu. W szczególności występowanie metody abstrakcyjnej nie pozwala zapomnieć o podaniu jej implementacji w definicji podklasy.

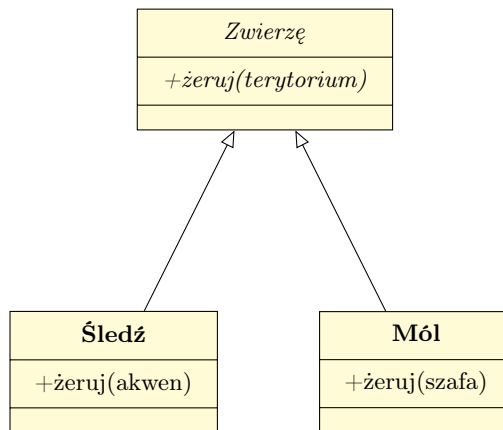
6.4 Przesłanianie metod

Jeżeli w podklasie występuje metoda o tej samej nazwie co w nadklasie, przesłania ona metodę zdefiniowaną wyżej. Mogłoby się wydawać, że w takiej sytuacji metoda z nadklasy jest bezpowrotnie stracona. W rzeczywistości jednak tak nie jest. Przesłanianie metody stosuje się w dwóch przypadkach:

- metoda z nadklasy robi coś zupełnie innego niż potrzeba,
- metoda z nadklasy nie robi wszystkiego co potrzeba.

W pierwszym przypadku metoda w podklasie przesłoni metodę z nadklasy i cel zostanie osiągnięty. W poniższym przykładzie implementacje metody `odpowiedz` w klasach `Student` i `DobryStudent` różnią się zasadniczo i nie posiadają wspólnych funkcjonalności:

³Oczywiście nic nie stoi na przeszkodzie, aby klasa abstrakcyjna zawierała również zwykłe metody.



Rysunek 6.7: Przykład grupowania ze względu na przynależność do jednej kategorii. Kursywa oznacza metodę lub klasę abstrakcyjną.

```

class Student:
    def odpowiedz(self, pytanie):
        self.zrob_karpia()
        self.dukaj(pytanie)

class DobryStudent(Student):
    def odpowiedz(self, pytanie):
        self.odpowiedz_spiewajaco(pytanie)
  
```

Mimo to uzasadnione jest, aby klasa `DobryStudent` dziedziczyła z klasy `Student` (z powodu innych pominiętych w przykładzie funkcjonalności). Należałoby oczywiście rozważyć implementację dwóch klas: `KiepskiStudent` i `DobryStudent` dziedziczących z abstrakcyjnej klasy `Student`.

W drugim przypadku warto implementując metodę w podklasie wywołać metody z nadklasy, aby wykorzystać już zaimplementowaną funkcjonalność, a następnie wykonać pozostałe czynności. Przykładem mogą być klasy `Lew` i `GroznyLew`, które różnią się tym, że groźny lew przed jedzeniem zawsze ryczy:

```

class Lew:
    def pozryj(self, antylopa):
        print "Mniam."

class GroznyLew(Lew):
    def pozryj(self, antylopa):
        self.rycz()
        Lew.pozryj(self, antylopa)
  
```

Sytuacja, w której implementując metodę w podklasie wywołuje się metodę z nadklasy, czyli rozszerza się funkcjonalność odziedziczonej metody, jest bardzo pożądana. W szczególności opłacalne jest umieszczenie w nadklasie metody o niekompletnej funkcjonalności z założeniem, że zostanie ona uzupełniona w podklasach.

6.5 Klasy abstrakcyjne

Jak wspomniano, w Pythonie nie ma potrzeby definiowania metod abstrakcyjnych. Jednak chociażby w celu dokumentacyjnym może być to celowe. Można to zrobić na dwa sposoby. Pierwszy polega na zdefiniowaniu zwykłej metody, której jedynym efektem będzie podniesienie wyjątku `NotImplementedError`. Taka metoda będzie mogła zostać wywołana wyłącznie dla obiektów należących do klas, w których nie została przesłonięta. Próba jej wywołania zakończy się wystąpieniem wyjątku informującego o braku implementacji. W poniższym przykładzie w ten sposób zaimplementowano metodę `speak` w klasie `Animal`:

```
class Animal():
    def speak(self):
        raise NotImplementedError #abstract

class Dog(Animal):
    def speak(self):
        return "bark"

class MuteAnimal(Animal):
    pass
```

W klasie `MuteAnimal` nie ma definicji metody `speak`. Jednak można tworzyć obiekty do niej należące. Kłopot wystąpi dopiero w momencie próby wywołania metody `speak`.

Alternatywnie, korzystając z modułu `ABCMeta` można zdefiniować klasy prawdziwie abstrakcyjne, czyli takie, których obiektów nie można tworzyć. Definicję metody abstrakcyjnej poprzedza się dekoratorem `abstractmethod`. Nadal jednak może ona mieć nietrywialną implementację i być wywoływana (np. w celu rozszerzenia jej funkcjonalności).

```
from abc import ABCMeta, abstractmethod

class Animal():
    __metaclass__ = ABCMeta

    @abstractmethod
    def speak(self):
        pass

class MuteAnimal(Animal):
    pass

class Dog(Animal):
    def speak(self):
        print "Bark"
```

W powyższym przykładzie nie można tworzyć obiektów klas `Animal` i `MuteAnimal`:

```
>>> Animal()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```

TypeError: Can't instantiate abstract class Animal with abstract methods speak
>>> MuteAnimal()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class MuteAnimal with abstract methods speak
>>> Dog()
<__main__.Dog object at 0x10ade20d0>

```

ponieważ zawierają one metodę abstrakcyjną (`MuteAnimal` ją dziedziczy).

6.6 Wzorzec projektowy *Template method*

Niekiedy metoda w klasie i metody ją przesłaniające w podklasach działają według wspólnego schematu. Różnice mogą być jednak na tyle istotne, że nie jest opłacalne zaimplementowanie jednej łatwo parametryzowalnej metody. Wówczas można rozważyć posłużenie się wzorcem projektowym *Template method*. Polega on na utworzeniu w nadklasie metody, która nie będzie przesłaniana w podklasach, implementującej algorytm w ogólnym (wspólnym dla wszystkich podklas) zakresie. Jego szczegółowe aspekty są realizowane przez konkretne metody zaimplementowane w podklasach. Przykładem realizacji tego wzorca może być metoda `poluj` w klasie `Drapieżnik`:

```

class Drapieżnik:
    def poluj(self, ofiara):
        if not self.czy_smaczna(ofiara):
            return

        lup = self.gon(ofiara)

        if lup is not None:
            self.jedz(lup)

```

Metoda `poluj` zawiera ogólny schemat polowania wspólny dla wszystkich drapieżników. Najpierw oceniana jest przydatność potencjalnej ofiary (metoda `czy_smaczna`). Następnie następuje pościg (metoda `gon`). Jeżeli polowanie powiedzie się ofiara jest zjadana (metoda `jedz`). Te metody są już charakterystyczne dla konkretnego drapieżnika (lub grupy drapieżników) i muszą być zaimplementowane w podklasach.

Stosowanie wzorca *Template method* daje bardzo dużą swobodę w rozmieszczeniu funkcjonalności w podklasach. Każda z metod wywoływanych w metodzie wzorcowej, może zostać zaimplementowana na odpowiednim poziomie w hierarchii.

6.7 Kiedy warto rozszerzyć klasę?

Dziedziczeniem należy posługiwać się rozważnie. Dosty często spotykanym błędem jest tworzenie podklas, które mają mieć w założeniu funkcjonalność mniejszą od nadklasy. Podejmując decyzję o dziedziczeniu warto wziąć pod uwagę czy:

- wszystkie składowe nadklasy zostaną wykorzystane lub przysłonięte w użyteczny sposób,
- obiekt podklasy ma udawać (być używany tak samo jak) obiekt nadklasy.

Jeżeli żaden z powyższych warunków nie jest spełniony (czyli np. zachodzi konieczność przysłonięcia wielu metod w sposób trywialny), należy stworzyć wspólną nadklasę. Rozważmy często przytaczany przykład figur geometrycznych. Klasa `Prostokat` posiada dwa atrybuty (`a` i `b`) przechowujące długości boków oraz metody `pole` i `obwod`:

```
class Prostokat:
    __init__(self, a, b):
        self.a = a
        self.b = b

    def pole(self):
        return self.a * self.b

    def obwod(self):
        return (self.a + self.b) * 2
```

Kwadrat jest szczególnym przypadkiem prostokąta o bokach równej długości. Jeżeli jednak chcielibyśmy porównywać funkcjonalność klasy `Prostokat` i `Kwadrat` zauważymy, że prostokąt jest uogólnieniem kwadratu. Paradoksalnie mogłoby się wydawać, że w takiej sytuacji klasa `Prostokat` powinna dziedziczyć z klasy `Kwadrat`, skoro rozszerza jej funkcjonalność. Jednak nie da się wykorzystać metod obliczających pole i obwód kwadratu w obliczaniu tych wielkości dla prostokąta. Możliwe jest to w drugą stronę. Wystarczy stworzyć prostokąt o równych bokach:

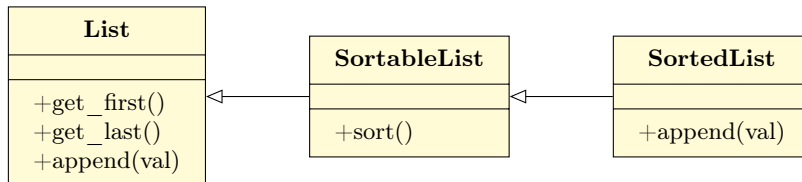
```
class Kwadrat(Prostokat):
    __init__(self, a):
        self.a = a
        self.b = a
```

Powyższe rozwiązanie ma jednak istotną wadę. Obiekty klasy `Kwadrat` mają atrybut `b`. Co więcej, nie ma mechanizmu gwarantującego, że oba atrybuty (`a` i `b`) mają tę samą wartość. Ten efekt można jednak osiągnąć stosując kapsułkowanie. Pewną niedoskonałością będzie jedynie przesłonięcie w klasie `Kwadrat` funkcjonalnych metod `set_b` i `get_b` z klasy `Prostokat` przez metody zgłaszające próbę wykonania niedozwolonej operacji lub takie ich zaimplementowanie, aby atrybut `b` w kwadracie zachowywał się tak samo jak `a`.

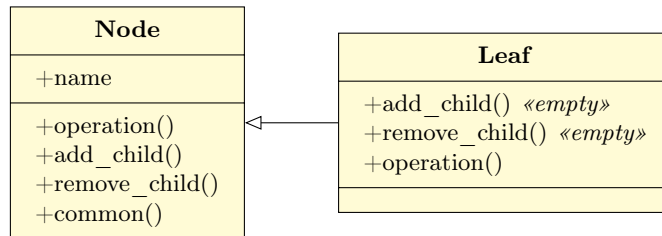
Na rysunku 6.8 przedstawiono przykłady poprawnego i niepoprawnego wykorzystania dziedziczenia.

6.8 Podsumowanie

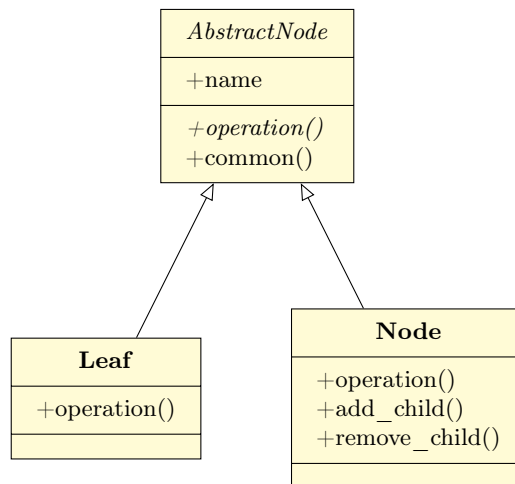
Projektowanie hierarchii klas jest dosyć newralgicznym etapem tworzenia programu. Błędy popełnione na tym etapie będą wymuszały niezgodne z intuicją sposoby wykorzystania zaprojektowanych klas, a implementacja metod może być złożona ze względu na konieczność obsługi wielu sytuacji wyjątkowych. Dlatego warto jest kierować się kilkoma zasadami:



(a) Rozszerzanie funkcjonalności – dobrze.



(b) Ograniczanie funkcjonalności – źle.



(c) Przynależność do wspólnej kategorii – dobrze.

Rysunek 6.8: Przykłady dziedziczenia

1. Klasy dziedziczące po sobie powinny zachowywać swoje właściwości. W każdym miejscu, gdzie może wystąpić nadklasa lub jej obiekt, powinno być możliwe użycie odpowiednio podklasy lub jej obiektu.
2. Nie należy obawiać się stosowania klas abstrakcyjnych w celu umieszczenia w nich funkcjonalności wspólnej dla wszystkich ich podklas. To pozwala ograniczyć ilość powtarzającego się kodu i co za tym idzie eliminować potencjalne powtórzenia błędów.
3. Jeżeli klasy mają wspólną funkcjonalność, ale nie mogą mieć wspólnej nadklasy, należy rozważyć utworzenie klasy pomocniczej zawierającej tę funkcjonalność. To pozwoli ograniczyć ilość powtarzającego się kodu.

Rozdział 7

Metody statyczne i klasowe

W poprzednich rozdziałach przyjmowaliśmy naturalny podział odpowiedzialności pomiędzy klasy i obiekty, w którym do obiektów były przypisane ich atrybuty (odpowiedzialność za przechowywanie informacji), a do klas metody (odpowiedzialność za funkcjonalności). W rzeczywistości nie ma racjonalnego powodu, aby utrzymywać to ograniczenie i nie dopuszczać do istnienia atrybutów w klasach. Co więcej nic nie stoi na przeszkodzie, aby definiować metody, które ignorują wartość swojego pierwszego argumentu (czyli działają tak samo niezależnie od obiektu).

Co więcej w języku Python klasy są obiektami, aczkolwiek takimi, które w pewnych sytuacjach zachowują się w sposób szczególny.

7.1 Atrybuty klas

Dlatego nic nie stoi na przeszkodzie, aby podobnie jak dla zwykłych obiektów, tworzyć w klasach atrybuty:

```
>>> class A:
...     pass
...
>>>
>>> A.class_attr = "classA"
>>>
>>> A.class_attr
'classA'
```

Atrybuty klasy różnią się od atrybutów obiektów¹ tym, że są również dostępne w ich instancjach:

```
>>> a = A()
>>> a.class_attr
'classA'
```

Co więcej są one również dziedziczone:

```
>>> class B(A): pass
```

¹Chodzi tu o obiekty, które nie są klasami.

```

...
>>> B.class_attr
'class_A'
>>>

```

Klasa B dziedziczy atrybut `class_attr` z klasy A dokładnie tak samo, jakby dziedziczyła metodę.

Powiemy, że z każdym obiektem (i klasą) w Pythonie jest związana *przestrzeń nazw*, która zawiera wszystkie atrybuty² należące do tego obiektu. Odwołanie do składowej obiektu (lub klasy) powoduje przeszukanie w ściśle ustalonej kolejności następujących przestrzeni nazw:

1. obiekt,
2. klasa obiektu,
3. nadklasa,
4. kolejna nadklasa,
5. ...

Przeszukiwanie odbywa się do skutku, czyli do momentu znalezienia poszukiwanej składowej, lub dojścia do klasy, która z niczego nie dziedziczy. Należy zauważyć, że przejście od obiektu do jego klasy w ciągu poszukiwania odbywa się tylko raz. Innymi słowy, nie są sprawdzane składowe klasy, do której należy klasa obiektu, który jest przeszukiwany.

Z przedstawionego algorytmu wynika, że składowe obiektu przesłaniają składowe klasy, podobnie jak składowe podklasy przesłaniają składowe nadklasy.

Cechą szczególną Pythona (wynikającą z typowania dynamicznego) jest to, że przypisanie wartości atrybutu zawsze odbywa się w najbliższej przestrzeni nazw. Może to być do pewnego stopnia zaskakujące, że przypisanie:

```
>>> a.class_attr = "obj_a"
```

nie spowoduje zmiany wartości atrybutu w klasie A, lecz utworzenie przesłaniającego go atrybutu w obiekcie a.

Podobnie przypisanie:

```
>>> B.class_attr = "classB"
```

nie będzie miało wpływu na klasę A. Spowoduje jedynie utworzenie nowego atrybutu w klasie B.

Atrybuty należące do przestrzeni nazw związanej z danym obiektem (lub klasą) znajdują się w słowniku będącym wartością atrybutu specjalnego `__dict__`.

7.2 Zastosowania atrybutów klasowych

Atrybuty klasowe mają kilka szczególnych cech, które nadają im użyteczność. Przede wszystkim mogą być tworzone w deklaracji klasy. To znaczy, że są dostępne zanim utworzony jakakolwiek instancja. Ponadto istnieje dokładnie jedna "kopia" atrybutu klasowego w danej klasie. W tym aspekcie są

²Metody są w istocie atrybutami klas.

one podobne do zmiennych globalnych. Istnieją zawsze i mogą służyć do przekazywania informacji w obrębie tych fragmentów systemu, w których dana klasa jest widoczna. Dzięki temu, że podlegają dziedziczeniu i przesłaniani pozwalają na tworzenie wartości wspólnych dla wybranych grup klas. Opłaca się je wykorzystywać po przechowywania:

- wartości charakteryzujących podklasy;
- wartości wspólnych dla wszystkich instancji (takich jak stałe i parametry konfiguracyjne);
- wszelkich wartości, do przechowywania których w innym przypadku należałoby posłużyć się zmienną globalną.

Rozważając pierwszy przypadek można zauważyć, że niekiedy stosując wzorec *Template method* zamiast metody definiowanej w podklasach wystarczy posłużyć się atrybutem:

```
class Zwierze:
    def daj_glos(self):
        print self.glos

class Pies(Zwierze):
    glos="Woof , woof"

class Swinka(Zwierze):
    glos="Oink , oink"

class LewHoward(Zwierze):
    glos="Roark"
    def daj_glos(self):
        Ziemia.trzes_sie()
        Zwierze.daj_glos(self)
```

Atrybut `glos` służy do określenia głosu wydawanego przez zwierzęta. Takie rozwiązanie pomijając to, że zaoszczędza wielokrotnego implementowania metody `daj_glos`, pozwala na łatwe zmienianie głosu w każdej instancji klas dziedziczących z klasy `Zwierze`.

Nigdy nie umieszcza się w programie stałych, ani parametrów jako wartości, ponieważ ich zmiana wiązałaby się z koniecznością odnalezienia wszystkich ich wystąpień. Należy posługiwać się zmiennymi³ i zawsze do danej wartości odwoływać się przez nazwę zmiennej. Ze względów projektowych dobrze jest umieszczać stałe i parametry jak najbliżej metod, które z nich korzystają. Atrybuty klasowe znakomicie się do tego nadają.

Przykładami innych “globalnych” wartości przechowywanych na poziomie klasy mogą być:

- licznik utworzonych instancji,
- ostatnio obliczona wartość,
- kontener na instancje,
- połączenie z bazą danych.

³W niektórych językach programowania można również definiować stałe.

7.3 Metody w klasach

Czasem można mieć pokusę, aby dodać do klasy metodę, która ignoruje swój pierwszy argument (`self`). Jej działanie oczywiście nie może zależeć od obiektu, dla którego jej wywoływana. Dlatego w zasadzie nie różni się ona od zwykłej funkcji. Ale jeżeli funkcjonalność jest w jakiś sposób powiązana z klasą, to może to służyć uporządkowaniu kodu. Przykładem mogą być metody służące do tworzenia obiektów. Znaczącym mankamentem jest brak możliwości wywołania takiej metody bez uprzedniego utworzenia instancji klasy. Skoro jednak możemy tworzyć atrybuty klasowe, dlaczego nie mielibyśmy móc definiować metod, które nie wymagają istnienia obiektów?

W większości języków obiektowych występuje pojęcie *metody statycznej*. Jest to metoda przypisana do klasy, która nie operuje na żadnym obiekcie. Odpowiada ona opisowi z początku podrozdziału i w zasadzie nie jest niczym więcej, jak tylko funkcją przypisaną do klasy. W Pythonie deklaracje metod statycznych poprzedza się dekoratorem `@staticmethod`:

```
class A:
    attr="class_A"

    @staticmethod
    def m():
        print A.attr
```

Metody statyczne nie pobierają obiektu, więc nie muszą posiadać argumentu `self`. Mogą za to odwoływać się do atrybutów klasy⁴.

Pozornie może wydawać się, że jedynym zastosowaniem metod statycznych jest lepsza organizacja kodu. Nie należy jednak zapominać, że jak wszystkie inne składowe klasy podlegają one dziedziczeniu i mogą być przesłaniane. Pewną trudnością w efektywnym wykorzystaniu dziedziczenia może być trudność w odwoływaniu się do atrybutów klasy. Jeżeli bowiem do atrybutów klasy można się dostać jedynie przez wskazanie tej klasy wprost, metoda statyczna wywołana w podklasie będzie, wbrew intuicji, korzystała z atrybutów nadklasy, co ilustruje przykład:

```
>>> class A:
...     attr="class_A"
...
...     @staticmethod
...     def print_attr():
...         print A.attr
...
>>> A.print_attr()
class A
>>>
>>> class B(A):
...     attr="class_B"
...
>>> B.print_attr()
class A
```

⁴Oczywiście każda inna funkcja i metoda też może.

```
>>>
```

Szczęśliwie w Pythonie istnieje dość nietypowy rodzaj metod, które stanowią remedium na opisaną niedogodność – *metody klasowe*. Różnią się one od zwykłych metod tym, że ich pierwszym argumentem nie jest instancja klasy, dla której zostały wywołane, lecz sama klasa. Mogą one być wywoływane dla klasy, jak również dla obiektu. W drugim przypadku jako argument przekazywana jest klasa obiektu, a nie on sam. Zwyczajowo pierwszy argument metod klasowych nazywa się `cls`. Definicje metod klasowych poprzedza się dekoratorem `@classmethod`. Rozważmy modyfikację przykładu przedstawionego powyżej:

```
>>> class A:
...     attr="class_A"
...
...     @classmethod
...     def print_attr(cls):
...         print cls.attr
...
>>> A.print_attr()
class A
>>>
>>> class B(A):
...     attr="class_B"
...
>>> B.print_attr()
class B
>>>
```

Metoda `print_attr` jest metodą klasową. Poprzez pierwszy argument można odwołać się do klasy, dla której została wywołana. W ten sposób klasa, z której atrybutów metoda korzysta, nie musi być sztywno określona w jej kodzie. Takie metody dobrze zachowują się w dziedziczeniu.

7.4 Wzorzec projektowy *Singleton*

Singleton to wzorzec projektowy, który polega na ograniczeniu liczby instancji danej klasy do jednej i zapewnieniu łatwego dostępu do jedynej instancji. Stosuje się go, jeżeli z różnych powodów wiele instancji wzajemnie by sobie przeszkadzało. *Singleton* może znaleźć zastosowanie przy implementowaniu fabryki obiektów, jeżeli jej działanie miałyby zależeć od uprzednio wyprodukowanych obiektów i potrzebna byłaby gwarancja, że wszystkie pochodzą z jednej fabryki. Przydatny może być również do przechowywania stanu aplikacji, albo w innej sytuacji, gdy nie można uniknąć zastosowania zmiennych globalnych i na dodatek ich struktura jest skomplikowana.

Implementacja prawdziwego *singletona*, którego jedyną instancję tworzy się i otrzymuje w taki sam sposób jak zazwyczaj tworzy się obiekty (czyli przez wywołanie klasy), wykracza poza zakres tego rozdziału. Można jednak łatwo osiągnąć podobny efekt posługując się atrybutami klasowymi i metodą statyczną:

```
class Singleton:
    _instance = None
```

```

@staticmethod
def get_instance():
    if Singleton._instance == None:
        Singleton._instance = Singleton()

    return Singleton._instance

```

Metoda `get_instance` zwraca instancję singletona jeżeli już została stworzona. W przeciwnym wypadku tworzy ją i zachowuje w klasowym atrybucie `_instance`. Takie rozwiązanie nie gwarantuje, że kolejne instancje nie zostaną stworzone bezpośrednio. Można się przed tym zabezpieczyć wprowadzając w metodzie `__init__` sprawdzenie wartości atrybutu `_instance`.

Singletonów należy używać wyłącznie, gdy jest to naprawdę konieczne, czyli da się udowodnić, że nie można postąpić inaczej.

7.5 Wzorzec projektowy *Factory methods* – uzupełnienie

Metody klasowe znakomicie nadają się do tworzenia obiektów. Można się nimi posługiwać implementując wzorzec *Factory method*.

Załóżmy, że konieczne jest zliczanie utworzonych instancji klasy, a informacja o liczbie instancji ma być przechowywana w atrybucie klasowym `count`. Efekt ten można uzyskać umieszczając w metodzie `__init__` zwiększenie wartości atrybutu:

```
self.__class__.count += 1
```

Niestety to rozwiązanie wymaga szczególnej staranności w implementowaniu metody `__init__` w podklasach. Każda metoda musi albo wywołać `__init__` z nadklasy, albo zawierać tę linijkę. Znacznie korzystniej byłoby umieścić tę funkcjonalność tam, gdzie należy czyli u szczytu hierarchii. Rozważmy klasę `Abstract`, która zawiera metodę `create`, która zwiększa licznik w klasie, dla której została wywołana oraz tworzy i zwraca nowy obiekt:

```

class Abstract:
    @classmethod
    def create(cls):
        cls.count+=1
        return cls()

class A(Abstract):
    count=0
    def __init__(self):
        print "init_A"

class B(A):
    count=0

```

Nowe obiekty tworzyć będziemy wywołując metodę `create` w odpowiedniej klasie:


```
>>> A.create()
init A
<A instance at 0x10a5c0cf8>
>>> A.create()
init A
<A instance at 0x10a5c0d88>
>>> B.create()
init A
<B instance at 0x10a5c0cf8>
>>> A.count
2
>>> B.count
1
```

Rozdział 8

Wyjątki

It's easier to ask forgiveness than
it is to get permission.

kadm. prof. Gracja Hopper

Im bardziej skomplikowany program, tym większa szansa, że znajdzie sytuacja nieprzewidziana przez programistę, która nie pozwala na kontynuowanie pracy programu. Takie zdarzenie można nazwać błędem. Usuwanie błędów jest pracą żmudną i z reguły mało satysfakcjonującą. Co więcej, wielokrotnie istnieje możliwość wznowienia pracy programu po wystąpieniu błędu, ale nie ma technicznej możliwości, aby to zrobić. Rozważmy sytuację, gdy w obiekcie brakuje atrybutu. Wówczas instrukcja:

```
val = obj.attr
```

nie może zostać wykonana. Wystarczyłoby jednak próbę dostępu do atrybutu poprzedzić sprawdzeniem, czy istnieje (służy do tego funkcja `hasattr`):

```
if hasattr(obj, 'attr'):
    val = obj.attr
else:
    val = None
```

Takie rozwiązanie jest niezmiernie żmudne i aby było skuteczne, każdą potencjalnie newralgiczną instrukcją należałoby opakować szeregiem sprawdzeń. Ponadto nie zawsze w miejscu wykrycia problemu można go właściwie obsłużyć. Czasem trzeba po prostu przerwać wykonywaną funkcję lub metodę przekazując w wyniku informację o rodzaju anomalii. Dlatego bardzo pożądany jest mechanizm, który pozwoliłby na rozdzielenie miejsca, gdzie problem występuje, od miejsca gdzie jest obsługiwany.

Podnoszenie (rzucanie) i przechwytywanie wyjątków (ang. *exceptions*) jest rozwiązaniem tego problemu. W momencie wystąpienia nieprzewidzianej sytuacji tworzony jest obiekt zwany wyjątkiem, a wykonywanie bieżącego fragmentu programu zostaje przerwane. Rozpoczyna się poszukiwanie miejsca, gdzie wyjątek mógłby zostać obsłużony. Jeżeli takowe zostanie znalezione, wznowia się od niego wykonanie programu. W przeciwnym wypadku czyli w sytuacji, gdy programista nie zapewnił obsługi danego wyjątku, wykonanie programu kończy się definitywnie.

W języku Python wyjątki są obiektami klasy `Exception`. Podnosi się je przy pomocy instrukcji `raise`, zaś instrukcja `try` służy do definiowania miejsc ich przechwytywania. Niektóre wyjątki podnoszone są automatycznie, bez używania instrukcji `raise`. Ma to miejsce w sytuacji, gdy interpreter nie jest w stanie poprawnie wykonać instrukcji np. na skutek braku odpowiedniego atrybutu. Ostatni przykład z zabezpieczeniem przed próbą dostępu do nieistniejącego atrybutu zaimplementowany przy pomocy wyjątków wygląda następująco:

```
try:
    val = obj.attr
except AttributeError:
    val = None
```

Na pierwszy rzut oka to rozwiązanie nie różni się znacząco od poprzedniego – w miejscu instrukcji `if` pojawia się instrukcja `try` lecz struktura programu pozostaje bez zmian. Sposób działania jest jednak zasadniczo odmienny. W rozwiązaniu pierwszym w instrukcji warunkowej sprawdza się czy atrybut istnieje, dopiero potem następuje próba dostępu do niego, w której to sprawdzenie musi być powtórzone¹. W rozwiązaniu z instrukcją `try` wykonywane jest tylko jedno sprawdzenie (przez interpreter). Jeżeli zakończy się ono niepowodzeniem, wyjątek, który w przeciwnym wypadku spowodowałby przerwanie pracy programu, zostaje przechwycony i wykonywana jest instrukcja zastępcza.

8.1 Instrukcja try

Instrukcja `try` może mieć dosyć skomplikowaną strukturę i składać się z wielu klauzul poprzedzających bloki kodu, które mają zostać wykonane w określonych okolicznościach. W najbardziej rozbudowanej formie ma ona postać:

```
try:
    # Instrukcje, w obrebie ktorych moga wystapic
    # wyjatki do przechwycenia.
except ExceptionClass1 as e:
    # Instrukcje, ktore zostana wykonane, jezeli
    # wystapil wyjatek nalezacy do klasy
    # ExceptionClass1.
    # Wyjatek znajduje sie na zmiennej e.
except ExceptionClass2 as e:
    # Instrukcje, ktore zostana wykonane, jezeli
    # wystapil wyjatek nalezacy do klasy
    # ExceptionClass2.
else:
    # Instrukcje, ktore zostana wykonane, jezeli
    # nie wystapil zaden wyjatek. Wyjatki w obrebie
    # tego bloku nie beda przechwytywane przez
    # te instrukcje try.
finally:
    # Instrukcje, ktore zostana wykonane na samym
    # koncu niezaleznie od tego, czy wystapil wyjatek.
```

¹Interpreter Pythona w jakiś sposób musi obsługiwać przypadek próby dostępu do nieistniejącego atrybutu.

W instrukcji `try` może wystąpić dowolnie wiele bloków `except`. Wewnątrz takiego bloku można mieć dostęp do obiektu reprezentującego wyjątek. Jest to przydatne, jeżeli obiekt ten zawiera dodatkowe informacje potrzebne do obsłużenia wyjątku. Jeżeli nie jest on konieczny, podawanie nazwy zmiennej po słowie kluczowym `as` można pominąć. Można również całkowicie pominąć podawanie klasy po klauzuli `except`. Będą wówczas przechwytywane wszystkie wyjątki.

Klauzula `else` służy do podania instrukcji, które mają być wykonane wyłącznie wtedy, gdy wyjątek nie nastąpił, a które nie mogą być dołączone do bloku po klauzuli `try`, ponieważ ewentualne wyjątki, które mogłyby w nich wystąpić, nie powinny być przez tę instrukcję przechwycone.

Klauzula `finally` jest przeznaczona do umieszczenia instrukcji, które muszą być wykonane zawsze niezależnie od wystąpienia wyjątku. Zazwyczaj służą one do wykonania czynności porządkowych takich jak zamknięcie plików, albo likwidacja innych zasobów.

Instrukcje `try` mogą być w sobie zagnieżdżone w obrębie jednej funkcji. Może się również zdarzyć, że w funkcji wywołanej w bloku `try` znajduje się kolejna instrukcja `try`. W takim przypadku, jeżeli wystąpi wyjątek, przeszukiwane są po kolei wszystkie aktualnie wykonywane instrukcje `try` w kolejności od najbardziej zagnieżdżonej. Pierwsza klauzula `except`, która pasuje do wyjątku, zostanie wykorzystana do obsłużenia go.

Instrukcja `try`, która przechwytuje wyjątki mogące wystąpić w funkcji `doSomething` wygląda następująco:

```
try:
    f=open(...)
    # kod, który może wygenerować wyjątek
    doSomething(...)
except Exception1 as e:
    # przechwycono wyjątek klasy Exception1
    process_exception(e)
except Exception2:
    # przechwycono wyjątek klasy Exception2
    print "Olaboga!!!"
else:
    jestSuper() # nie wystąpił żaden wyjątek
finally:
    # niezależnie co się stało, trzeba posprzątać
    f.close()
```

W przykładzie warto zwrócić uwagę na klauzulę `finally`, w której umieszczono instrukcję zamykającą plik. Zostanie ona wykonana zawsze, niezależnie od wyjątków, które mogłyby wystąpić².

8.2 Wyjątki standardowe

W Pythonie występuje szereg wyjątków, które zostaną automatycznie podniesione przez interpreter w przypadku napotkania na okoliczności uniemożliwiające dalsze wykonywanie programu. Są to między innymi:

²W szczególności jeżeli wystąpi wyjątek, którego ta instrukcja `try` nie przechwytuje.

ArithmeticError nadklasa obejmująca wyjątki związane z błędami numerycznymi

AssertionError niespełniona asercja w instrukcji `assert`

AttributeError brak atrybutu

IndexError brak elementu o podanym indeksie

KeyError brak klucza w słowniku

NameError brak zmiennej

NotImplementedError nie zaimplementowana funkcjonalność

OverflowError przepełnienie

StopIteration koniec iteracji (podnoszony w metodzie `next()` iteratora)

ZeroDivisionError dzielenie przez 0

Przyglądając się liście można zaobserwować, że istotnie we wszystkich przypadkach nie da się racjonalnie określić sposobu dalszego działania. Jaką wartość miałby mieć nieistniejący element listy lub słownika? Jaki jest wynik dzielenia przez 0? Bardzo wygodne jest posługiwanie się w takim wypadku wyjątkiem zamiast sztuczną wartością specjalną. Szczęśliwie pisząc program z reguły nie trzeba przewidywać z góry wszystkich możliwych wyjątków (aczkolwiek nie jest to niewskazane). Podczas testowania wszystkie nieprzechwycone wyjątki będą powodowały przerwanie programu. Jeżeli w konkretnym przypadku okaże się to działaniem niepożądanym, będzie możliwe usunięcie usterki przez dodanie obsługi wyjątku w odpowiednim miejscu.

8.3 Własne wyjątki

W poprzednim podrozdziale omówiliśmy wyjątki, które są automatycznie podnoszone przez interpreter. Można przyjąć, że odpowiadają one zdarzeniom “niskopoziomowym” czyli takim, które z przyczyn technicznych uniemożliwiają działanie programu. W każdym systemie istnieje jednak wiele sytuacji, gdy problem może pojawić się na wyższym poziomie organizacji np. wskutek sprzeczności w danych. Obsługa takich błędów wymaga zdefiniowania własnych wyjątków.

Rozważmy program, który wczytuje dane z pliku tworząc na podstawie kolejnych linii na przemian obiekty klasy A i B. Wczytywanie należy zakończyć w momencie, gdy ostatnia para wczytanych obiektów ma określone własności. Taki program zaimplementowany bez użycia wyjątków mógłby wyglądać następująco:

```
def czytaj1(f):
    line=f.readline()
    if line=='': # koniec pliku
        return (None, 'EOF')
    else:
        return (A(line), 'OK')
```

```

def czytaj2(f):
    ...

def czytaj(f):
    done=False
    while not done:
        (resA, status)=czytaj1(f)
        if status=='EOF':
            return (None, 'EOF')
        (resB, status)=czytaj2(f)
        if status=='EOF':
            return (None, 'EOF')
        res.extend((resA, resB))
        done=przetworz(resA, resB)

    return (res, 'OK')

```

Funkcje `czytaj1` i `czytaj2`, których zadaniem jest wczytywanie obiektów odpowiednich klas, musiałyby oprócz obiektu, zwracać informację, czy wczytywanie się powiodło³. Funkcja `czytaj`, która na przemian wywołuje `czytaj1` i `czytaj2`, musi każdorazowo sprawdzać poprawność operacji oraz przekazywać wyżej informacje o powodzeniu całej operacji. Program wywołujący funkcję `czytaj` mógłby wyglądać jak poniżej:

```

f=...
(res, status)=czytaj(f)
if status=='EOF':
    print "Blad"
else:
    print 'OK'

```

Zauważmy, że w funkcji `czytaj` nie zaimplementowano żadnego przeciwdziałania na wypadek przedwczesnego napotkania na koniec pliku. Wczytywanie jest przerywane i informacja o porażce jest natychmiast zwracana.

Rozważmy rozwiązanie z wyjątkiem o nazwie `EOFException` zdefiniowanym na potrzebę napotkania na koniec pliku:

```

class EOFException(Exception):
    pass

def czytaj1(f):
    line=f.readline()
    if line=='': # koniec pliku
        raise EOFException()

    return A(line)

def czytaj2(f):
    ...

```

³Zwracanie wartości `None` w przypadku niepowodzenia nie zawsze jest dopuszczalne. Być może jest to wartość, która jest dopuszczalna również w sytuacji poprawnej.

```

def czytaj(f):
    done=False
    while not done:
        resA=czytaj1(f)
        resB=czytaj2(f)
        res.extend((resA, resB))
        done=przetworz(resA, resB)

    return (res)

```

Nowa funkcja `czytaj` nie zawiera, żadnej kontroli skuteczności działania wywoływanych przez nią funkcji. Nie musi, ponieważ nawet gdyby napotkano na koniec pliku, na poziomie funkcji `czytaj` nic nie można z tym zrobić. Funkcje `czytaj1` i `czytaj2` również są uproszczone. Zamiast pary obiekt-status zwracają tylko utworzony obiekt. W przypadku końca pliku podnoszą wyjątek.

Z nowej funkcji `czytaj` można korzystać w następujący sposób:

```

f=...
try:
    res=czytaj(f)
    print 'OK'
except EOFException:
    print "Blad"

```

Tutaj również mamy uproszczenie, albowiem zwracana jest pojedyncza wartość zamiast pary. Podniesienie wyjątku `EOFException` podczas wykonywania funkcji `czytaj` (w funkcjach przez nią wywoływanych) spowoduje natychmiastowy przeskoczenie do bloku po klauzuli `except`.

Każdy wyjątek jest obiektem klasy dziedziczącej z klasy `Exception`. Nie ma żadnych ograniczeń na funkcjonalność takich obiektów. Mogą one zawierać informacje związane z sytuacją, w której zostały podniesione oraz metody przydatne do jej obsługi. Możliwe (i wskazane) jest również korzystanie z mechanizmu dziedziczenia. Klauzula `except` przechwytywa wyjątki należące do podanej klasy i wszystkich jej podklas. Sposób ich obsługi można różnicować przez umieszczenie odpowiednich metod (odpowiednio przesłanianych) w podklasach.

8.4 Poprawne posługiwanie się wyjątkami

Wyjątki są mechanizmem bardzo ułatwiającym programowanie, ale mogą przyczynić się do powstawania trudnych do wykrycia błędów. Dlatego bezwarunkowo należy stosować się do zasady:

Catch only what you can handle.

W żadnym wypadku nie należy przechwytywać wyjątków, których nie można w sposób użyteczny obsłużyć. Przykładem takiego postępowania jest niewinnie wyglądający kod:

```

try:
    a = f(lista[i])

```

```

except:
    print "Lista nie ma elementu o indeksie:", i

```

Intencją programisty było oczywiście wyświetlenie komunikatu diagnostycznego w wypadku braku elementu w liście. Niestety tak sformułowana klauzula `except` będzie przechwytywała również wszystkie wyjątki, które mogą wystąpić podczas wykonywania funkcji `f`. Uzupełnienie klauzuli `except` o klasę wyjątku `IndexError` nie rozwiązuje problemu, albowiem przechwycone zostaną również wszystkie wyjątki tego typu, które mogłyby zaistnieć w funkcji `f`. Poprawna implementacja przechwytuje wyjątek `IndexError` związany z dostępem do listy, a gdyby nie wystąpił wywołuje funkcję `f`:

```

try:
    val = lista[i]
except IndexError:
    print "Lista nie ma elementu o indeksie:", i
else:
    a = f(val)

```

Jeżeli obsługa błędów w programie ma być realizowana przy pomocy wyjątków, można opóźnić jej implementację. W pierwszej kolejności warto przygotować prototyp, który nie jest wyposażony w obsługę błędów. Można wówczas przeprowadzić wstępne testy dobierając dane wejściowe tak, aby ominąć potencjalne problemy. Następnie należy program rozszerzyć o podnoszenie wyjątków tam, gdzie obliczenia nie mogą być kontynuowane. Jeżeli zostanie to prawidłowo zrobione, wszystkie przebiegi kończące się przerwaniem programu, będą opatrzone komunikatem dającym pełną informację o sytuacji, która spowodowała jego przerwanie. Wreszcie ostatnim krokiem jest umieszczenie w odpowiednich miejscach obsługi podnoszonych wyjątków.