Projekt Python – Serwery – wskazówki

Rozwiązanie – koncepcja

Produkty

Czy klasę ''Product'' można zrealizować jako nazwaną krotkę?

Każda krotka (w tym tzw. *nazwana krotka*) jest obiektem niemutowalnym, a w przypadku produktów ich cena zapewne będzie ulegać zmianie – dlatego lepiej zastosować "zwykłą" klasę.

Gdyby dane produktów nie ulegały zmianie po ich utworzeniu, wówczas przy obecnych wymaganiach uzasadnione byłoby użycie nazwanej krotki.
Zwróć uwagę, że począwszy od wersji 3.6 języka Python zmienił się zalecany sposób definiowania nazwanych krotek – zob. `typing.NamedTuple`.

Relacja z serwerem: kompozycja czy agregacja?

Przeczytaj ponownie informacje na stronie poświęconej diagramom UML w sekcji **Relacje miedzy klasami**.

Ponieważ w naszym systemie produkty istnieją wyłącznie w ramach danego serwera (produkty stanowią nieodłączną część serwera), dlatego poprawnym typem relacji jest kompozycja.

Warto oznaczyć krotności relacji jako `serwer 1:* produkty` – *"(każdy) serwer posiada dowolną liczbę produktów"*.

Czy należy tworzyć gettery i settery dla pól klasy?

W przypadku klasy `Product` pola powinny być publiczne! Co więcej, na chwilę obecną nie weryfikujemy poprawności otrzymywanych danych (co było powiedziane wprost w poleceniu).

W przyszłości zapewne będziemy chcieli dodać weryfikację nazwy produktu, ale... od tego są dekoratory `@property` (zob. skrypt do Pythona rozdz. *"Właściwości"*). Na tym polega urok języka Python, że pewnych decyzji projektowych nie musimy podejmować przedwcześnie – w razie czego możemy stosowne zmiany wprowadzić w sposób bezbolesny!

Serwery

Czy tworzyć hierarchię klas serwerów? Jeśli tak, to jak powinna ona wyglądać?

Każdy typ serwera powinien udostępniać pewną identyczną funkcjonalność:

- wyszukiwanie produktów z użyciem metody `f(n: int) -> List[Product]`
- rzucanie wyjątku, gdy zbyt duża liczba produktów zawiera nazwy pasujące do zadanego wzorca

Ponieważ klient nie musi znać szczegółów implementacji serwera z którego korzysta (powinien być w stanie korzystać z dowolnego typ serwera wymiennie), należy określić rodzaj "kontraktu", który powinien być spełniony przez wszystkie klasy potencjalnie używane w charakterze serwera. W języku Python do określania takiego kontraktu służą klasy abstrakcyjne – zatem tak, będziemy tworzyli hierarchię klas.

Ponieważ część implementacji będzie wspólna dla wszystkich serwerów, np.:

- logika procedury weryfikacji zgodności nazwy danego produktu ze wzorem jest identyczna dla wszystkich typów serwerów
- logika wykrywania sytuacji, w której należy rzucić wyjątek "za dużo znalezionych produktów"

ta wspólna część implementacji powinna się znaleźć w abstrakcji serwera (klasa o nazwie `Server`) – stąd używając terminologii z zakresu inżynierii oprogramowania: klasa macierzysta będzie *klasą abstrakcyjną*, a nie *interfejsem*.
W języku Python nie ma interfejsów sensu *stricte*, tak jak np. w języku Java (zob. The Java™ Tutorials – What Is an Interface?)!

"Zasady kciuka"[1] (zob. rule of thumb): **Klasa `Cls` zawiera metodę `m()`. Kiedy metoda ta powinna być metodą abstrakcyjną?**

Odpowiedz sobie na pytania zawarte w poniższym schemacie

Czy (macierzysta) klasa `Cls` faktycznie potrzebuje tej metody do realizacji swojej funkcjonalności?

- Nie. ⇒ To po co w ogóle została utworzona? Być może naruszasz zasadę YAGNI.
- Tak. ⇒ Czy jesteś w stanie podać rozsądną implementację metody `m()` w klasie `Cls` – taką, która będzie poprawna dla wszystkich klas potomnych klasy `Cls`?
  - Tak. ⇒ Metoda `m()` nie powinna być abstrakcyjna.
  - Nie. ⇒ Czy jesteś w stanie wydzielić jakąś część funkcjonalności metody `m()` wspólną dla wszystkich potencjalnych klas potomnych (wspólny schemat postępowania)?
    - Nie. ⇒ Metoda `m()` powinna być abstrakcyjna.
    - Tak. ⇒ Utwórz co najmniej dwie metody – jedną nieabstrakcyjną metodę `m_con()`, zawierającą logikę wspólną dla wszystkich implementacji, oraz abstrakcyjne metody pomocnicze dostarczające metodzie `m_con()` niezbędnych danych w zestandaryzowany sposób (tj. w tym samym formacie niezależnie od klasy potomnej).

Jakie powinny być specyfikatory metod umieszczonych w abstrakcji serwera?

Ogólna logika metody zwracającej listę produktów na podstawie zadanej liczby liter jest wspólna dla wszystkich serwerów:

- pobierz listę produktów spełniających kryterium
- sprawdź, czy lista jest zbyt długa: jeśli tak – rzuć wyjątek, jeśli nie – zwróć posortowaną listę

jednak szczegółowa logika fragmentu *"pobierz listę produktów spełniających kryterium"* będzie zależeć od sposobu reprezentacji danych w konkretnym typie serwera. Dlatego w publicznym API abstrakcji serwera będzie to zwykła metoda, natomiast metoda realizująca "wewnętrzne" pobieranie listy produktów będzie "prywatną" metodą abstrakcyjną (do zaimplementowania w każdej z klas reprezentujących konkretny typ serwera).

Do dobrych praktyk należy umieszczanie w ciele metody abstrakcyjnej w klasie macierzystej instrukcji `raise NotImplementedError`, a nie `pass` (żeby podkreślić celowy brak implementacji takiej metody)!

Gdzie i jak zdefiniować wyjątek rzucany przez serwery, aby zachować dobre praktyki programistyczne?

Zapoznaj się z rozdziałem *"Wyjątki zdefiniowane przez użytkownika"* w skrypcie do Pythona, zwróć uwagę na hierarchię klas w przypadku definiowania nowych wyjątków.

Zgodnie z dobrymi praktykami programistycznymi w Pythonie, nazw własnych klas wyjątków powinny kończyć się na "`Error`" (zob. Errors and Exceptions – User-defined Exceptions).

Ponieważ będziemy tworzyć hierarchię serwerów, które potencjalnie mogą rzucać też inne wyspecjalizowane wyjątki, do dobrej praktyki należy definiowanie jednej klasy macierzystej dla wszystkich typów wyjątków w ramach danej biblioteki – w tym przypadku np. `ServerError`. Dzięki temu później łatwo wychwycić (i obsłużyć) wszystkie wyjątki z takiej biblioteki.

Choć wyjątek informujący o zbyt długiej liście wyników wyszukiwania przynależy ściśle do domeny abstrakcji serwera, to ponieważ jest on rzucany przez publiczną metodę klasy `Server` – wyjątek ten może "wypłynąć" do innych modułów. Zgodnie z dobrymi praktykami importu/eksportu obiektów powinniśmy umożliwić korzystanie w innych modułach z takiej klasy wyjątku bez konieczności importu całej klasy `Server` (zob. Stack Overflow). W związku z tym obie zdefiniowane przez nas klasy wyjątków powinny być tzw. klasami zewnętrznymi (a nie klasami zagnieżdżonymi w klasie `Server`), a ich relacja z klasami w których rzucamy dany typ wyjątku – relacją typu *używa*.

Jaki wzorzec wyrażenia regularnego pozwoli zrealizować pożądane kryterium wyszukiwania nazw produktów?

Zgodnie z treścią zadania:

> Serwery udostępniają funkcję wyszukiwania produktów, których nazwa składa się z ciągu "n liter (dowolnej wielkości), a następnie (dokładnie) 2–3 cyfry" (…)

Poprawny wzorzec (z użyciem składni języka Perl, z której korzysta Python – zob. re — Regular expression operations) można schematycznie przedstawić jako: `^[a-zA-Z]{n}\d{2,3}$`
W powyższym zapisie za n należy podstawić n z treści polecenia, zatem np. dla $n=1$ "konkretny" wzorzec będzie wyglądał tak: `^[a-zA-Z]{1}\d{2,3}$`

Elementy składni wyrażeń regularnych, pozwalających na efektywną realizację polecenia:

- `^` – dopasowanie do początku łańcucha znaków (nie do znaku, tylko do pozycji)
- `[a-zA-Z]` – dopasowanie do dowolnej litery z zakresu a–z lub A–Z
  Użycie klasy znaków `\w` jest niepoprawne, gdyż obejmuje ona wszystkie znaki alfanumeryczne (a więc też cyfry)!
- `{m}` – (dokładnie) m-krotne dopasowanie dla RE poprzedzającego ten specyfikator krotności
- `\d` – dopasowanie do dowolnej cyfry
- `{m,n}` – RE poprzedzający ten specyfikator krotności musi zostać dopasowany co najmniej m-krotnie, ale nie więcej niż n-krotnie
- `$` – dopasowanie do końca łańcucha znaków (nie do znaku, tylko do pozycji)

Aby dla zmiennej `n_letters` odpowiadającej n otrzymać łańcuch znaków, który pozwoli na kompilację odpowiedniego obiektu wyrażenia regularnego, możesz użyć poniższego kodu:

```
'^[a-zA-Z]{{{n}}}\\d{{2,3}}$'.format(n=n_letters)
```

Aby po wywołaniu metody `str.format()` w wynikowym łańcuchu znaków otrzymać nawiasy klamrowe ({ i }), należy w łańcuchu formatującym umieścić odpowiednio {{ i }} (należy powtórzyć odpowiedni nawias klamrowy dwukrotnie). Z kolei najbardziej wewnętrzna para nawiasów klamrowych, otaczających `n_letters` odnosi się do pola, które ma zostać uzupełnione odpowiednią wartością w łańcuchu wynikowym (zob. pytanie How can I print literal curly-brace characters in python string and also use .format on it? oraz zaakceptowaną odpowiedź).

Ewentualnie (równoważnie) możesz użyć tzw. *surowych łańcuchów znaków*:

```
r'^[a-zA-Z]{' + str(n_letters) + r'}\d{2,3}$'
```

W przypadku, gdy wzorzec nie zostanie dopasowany do łańcucha znaków, metoda `re.match()` zwraca wartość None.

W tym zadaniu nie ma sensu korzystać z funkcji `re.compile()` dokonującej kompilacji wzorca w postaci tekstowej do specjalnego obiektu wyrażenia regularnego – ona przydaje się wówczas, gdy wielokrotnie dopasowujemy do tego samego wzorca (poprawia wydajność programu), jednak w tym zadaniu za każdym razem potrzebujemy innego wzorca (zależnego od n).

Klienci

Relacja z serwerem: kompozycja czy agregacja?

Przeczytaj ponownie informacje na stronie poświęconej diagramom UML w sekcji **Relacje miedzy klasami**.

Ponieważ w naszym systemie z jednego serwera może korzystać kilku klientów (klienci współdzielą serwery) – serwer nie stanowi nieodłącznej części klienta. W związku z tym poprawnym typem relacji jest agregacja. W przeciwnym razie usunięcie jednego klienta powodowałoby usunięcie serwera, co uniemożliwiłoby korzystanie z niego przez innych klientów!

Nie zaznaczamy krotności relacji jako `serwer 1:* klient`, gdyż serwer nie przechowuje informacji o klientach, którzy z niego korzystają (oznaczenie krotności stosujemy w przypadku kolekcji obiektów).

Jaka powinna być podpowiedź typu dla pola "serwer"?

W przypadku, gdy dana zmienna powinna umożliwiać przechowywanie referencji zarówno do klasy macierzystej, jak i klas potomnych, należy zdefiniować odpowiedni typ pomocniczy (zob. PEP484: Type variables with an upper bound):

```
HelperType = TypeVar('HelperType', bound=T)
```

Podanie argumentu słownikowego `bound=T` sprawia, że typ `HelperType` będzie oznaczał: “klasa T albo jej dowolna klasa potomna”.

UML

Czy trzeba na diagramie umieszczać klasę ABC?

Dziedziczenie po klasie ABC ze standardowego modułu `abc` ma na celu wyłącznie umożliwienie emulacji zachowania klasy abstrakcyjnej (m.in. jej użycie jest wymagane, aby dekorator `@abstractmethod` działał poprawnie) – do oznaczania klas jako abstrakcyjnych w PlantUML służy słowo kluczowe `abstract`, którym należy poprzedzić nazwę danej klasy.

Testy jednostkowe

W jaki sposób sprawdzić, czy rzucony został wyjątek?

W tym celu należy użyć konstrukcji z `assertRaises`:

```
with self.assertRaises(ExT):
    ...
```

która oczekuje, że któraś z instrukcji w bloku with rzuci wyjątek typu ExT. Jeśli taki wyjątek nie zostanie rzucony – test nie przechodzi.

Ważne zasady inżynierskie

DRY

(zob. **Zasada DRY**)

W przypadku tego zadania duplikacja mogła pojawiać się m.in. w sytuacji, gdy abstrakcja serwera zawierała wyłącznie metodę abstrakcyjną służącą do wyszukiwania produktów, a całość jej implementacji znajdowała się w odpowiednich klasach konkretnych reprezentujących poszczególne typy serwerów.

Dokonywanie ciągu operacji *"kopiuj → wklej → (nieco) zmień"* to najlepsza wskazówka, że coś robimy "nie tak" – że naruszamy zasadę DRY.

W takiej sytuacji należy wydzielić największą wspólną część funkcjonalności "poziom wyżej", w tym konkretnym przypadku – do abstrakcji serwera, do klasy macierzystej. Klasy potomne powinny być odpowiedzialne wyłącznie za realizację tych operacji, których klasa macierzysta zwyczajnie nie jest w stanie wykonać (bo brak jej odpowiednich danych) – w tym przypadku chodzi o zwracanie kolekcji wszystkich produktów w ustandaryzowany sposób, gdyż serwery stosują różne kontenery do ich przechowywania.

YAGNI

(zob. **Zasada YAGNI**)

Nie ma sensu tworzyć nadmiarowej funkcjonalności, nie wymaganej nigdzie w specyfikacji (np. metoda do dodawania produktów w hierarchii klas serwerów, pole "nazwa" w klasie reprezentującej klienta itp.). Po co w tym przypadku poświęcać nerwy i czas (= pieniądze!) na coś, co nie będzie nikomu potrzebne?

"Zasady kciuka"[2] (zob. rule of thumb): **Czy dodawać daną funkcjonalność do tworzonego systemu już w tym momencie?** (albo **Czy naruszam zasadę YAGNI?**)

Odpowiedz sobie na pytania zawarte w poniższym schemacie

Czy dana funkcjonalność jest wprost wymagana w specyfikacji?

- Tak. ⇒ Wszystko jest OK.
- Nie. ⇒ Czy dana funkcjonalność jest niezbędna do realizacji wymagań zawartych w specyfikacji?
  - Tak. ⇒ Wszystko jest OK.
  - Nie. ⇒ Czy jest bardzo prawdopodobne, że dana funkcjonalność będzie potrzebna?
    - Nie. ⇒ Naruszasz zasadę YAGNI.
    - Tak. ⇒ Czy łatwo będzie wprowadzić taką funkcjonalność (bez wprowadzania istotnych zmian w istniejącym kodzie)?
      - Tak. ⇒ Naruszasz zasadę YAGNI.

- Nie. ⇒ Ok, być może warto ją wprowadzić już teraz, ale lepiej skonsultuj to ze swoim przełożonym…

Dobre praktyki

Podpowiedź typu, gdy dopuszczamy dwa typy wartości: `T` i `None`

Sposobem zgodnym z dobrymi praktykami będzie użycie podpowiedzi `Optional[T]`, która jest równoważna podpowiedzi `Union[T, None]` – “typ T albo typ None” (zob. PEP484 – Union types).

Jak zwiększyć czytelność diagramów UML?

Czasem nie ma sensu pokazywać wszystkich (pustych) rubryk klas – w przypadku PlantUML można je ukryć dodając w kodzie polecenie `hide empty members` (zob. Diagramy klas UML.

[1)] [2)]
Zasady o szerokim zastosowaniu, które nie mają być zawsze poprawne, ale mają w przystępny sposób tłumaczyć pewne ogólne prawidła.

From:
http://home.agh.edu.pl/~mdig/dokuwiki/ - **MVG Group**

Permanent link:
http://home.agh.edu.pl/~mdig/dokuwiki/doku.php?id=teaching:programming:soft-dev:solutions:servers_hints

Last update: **2020/09/30 08:54**