Klasy i obiekty — część 3

1. Klasa abstrakcyjna

1. Utwórz plik 'deanerySystem/BasicTimetable.py' o następującej zawartości:

```
from abc import ABC, abstractmethod
from enum import Enum
from typing import List


class Action(Enum):
    pass
############################################################


class Lesson:
    pass
############################################################


class Term:
    pass
############################################################


class BasicTimetable(ABC):
    @abstractmethod
    def busy(self, term: Term) -> bool:
        pass
############################################################

    def get(self, term: Term) -> Lesson:
        print("Wywołano metodę 'get()' zdefiniowaną w klasie 'BasicTimetable'")
############################################################

    @abstractmethod
    def parse(self, actions: List[str]) -> List[Action]:
        pass
############################################################

    @abstractmethod
    def perform(self, actions: List[Action]):
        pass
############################################################

    @abstractmethod
    def put(self, lesson: Lesson) -> bool:
        pass
```

2. Wykonaj komendę `python3 -i deanerySystem/BasicTimetable.py`, a następnie (w konsoli Python) wprowadź poniższy kod:

```
class Timetable(BasicTimetable):
    def parse(self, actions):
        pass
    def perform(self, actions):
        pass
    def busy(self, term):
        pass
```

```
# Sprawdź, czy można tworzyć instancję klasy abstrakcyjnej – 'BasicTimetable'
timeTable = BasicTimetable()

# Sprawdź, czy można tworzyć instancję klasy pochodnej – 'Timetable'
timeTable = Timetable()

# Wywołujemy metodę, która NIE JEST zdefiniowana w klasie 'Timetable', a w klasie 'BasicTimetable'
timeTable.get(Term())
```

3. Korzystając z informacji przekazanych na wykładzie, doprowadź powyższy kod (definicję klasy *Timetable*) do stanu poprawności — ma być możliwe tworzenie instancji klasy *Timetable* oraz wywołanie metody `get()` utworzonego obiektu; inne metody też mogą być wywoływane ☺

2. Rozbudowa aplikacji

1 pkt za wykonanie zadań z tej sekcji

1. Utwórz plik 'deanerySystem/Break.py' zawierający definicję klasy *Break*, która:
   - posiada konstruktor `Break(Term term)`, którego parametr określaja termin przerwy (pomiędzy zajęciami) oraz czas jej trwania — UWAGA: zakładamy, że przerwy odbywają w danym czasie przez wszystkie dni tygodnia, czyli pole *day* nie jest używane
   - posiada metodę `__str__()` zwracającą ciąg trzech znaków "-"
   - posiada metodę `getTerm()` zwracającą informacje nt. terminu przerwy
2. Utwórz plik 'deanerySystem/TimetableWithBreaks.py' zawierający definicję klasy *TimetableWithBreaks*, która:
   - posiada te same metody co *TimetableWithoutBreaks*
   - posiada konstruktor `TimetableWithBreaks(breaks: List[Break] )`, gdzie *breaks* jest listą przerw (pomiędzy zajęciami)
   - posiada metodę `__str__()`, która działa podobnie jak analogiczna metoda klasy *TimetableWithoutBreaks*, ale dodatkowo wyświetla przerwy w postaci trójelementowego ciągu znaków "-"
   - umożliwia przenoszenie zajęć tak, jak to czyni klasa *TimetableWithoutBreaks*, czyli po warunkiem, że:
     1. przenoszone zajęcia nie kolidują z terminem pozostałych zajęć
     2. nie wychodzą one poza przyjęte ramy czasowe — patrz pkt 2 ćwiczenia nr 4
   - Dodatkowo, w przypadku klasy *TimetableWithBreaks* wprowadzamy jeszcze jeden warunek:
     3. zajęcia nie mogą kolidować z terminem przerw, o ile własność 'skipBreaks' ma wartość 'false' — opis tej własności znajdziesz poniżej
   - W przypadku próby dodania lekcji, której termin "nachodzi" na termin przerwy, dana lekcja ma nie być umieszczana w rozkładzie zajęć
3. Rozszerz listę własności wybranej klasy (zastanów się, która z istniejących jest najodpowiedniejsza) o własność statyczną (klasową) `skipBreaks: bool` — określa ona, czy przy przesuwaniu zajęć, można czy nie można "przenikać" przez przerwy.
   Przykładowo, jeżeli założymy, że przerwa jest w godzinach 9:30-9:40, a `skipBreaks` ma wartość 'true', to przesunięcie zajęć 8:00-9:30 o "90 minut" ma spowodować, że będą one się odbywać w godzinach 9:40-11:10, a więc przesuną się, de facto, o "100 minut" (czas trwania$_{\text{zajęcia}}$ + czas trwania$_{\text{przerwa}}$). Jeżeli `skipBreaks` ma wartość 'false', to przesunięcie tych zajęć nie powinno powieść się, gdyż nowy termin koliduje z terminem przerwy
4. Wprowadź inne, niezbędne modyfikacje do pozostałych klas, aby działała możliwość "przenikania" przez przerwy
5. Sprawdź czy implementacja jest poprawna — umieść przerwy w tych samych terminach co na naszym kierunku, tzn. 9:30-9:40, 11:10-11:20, itd.; dodaj kilka przedmiotów ze swojego, rzeczywistego rozkładu zajęć; wykonaj tę samą sekwencję działań (ciąg przesunieć zajęć) co w ćwiczeniu 4
6. Stwórz testy, które sprawdzają poprawność implementacji metod

3. Mechanizm dziedziczenia

1 pkt za wykonanie zadań z tej sekcji

Jak można zauważyć, duża część kodu klas *TimetableWithoutBreaks* oraz *TimetableWithBreaks* pokrywa się. Ponadto

klasa *Break*, w porównaniu do klasy *Lesson*, nie korzysta ze wszystkich pól klasy *Term*

1. Utwórz klasę *BasicTerm* — ma zawierać tylko te informacje o terminie (te pola), które są wspólne dla zajęć oraz przerw
2. Zmodyfikuj klasę *Term* — ma dziedziczyć z klasy *BasicTerm* i rozszerzać listę pól (z informacjami o terminie) o to, które jest potrzebne w przypadku zajęć, a zbędne w przypadku przerw

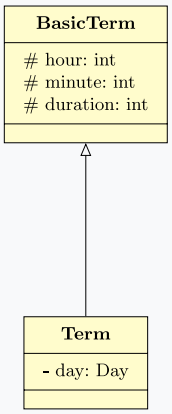

Diagram klas służących do reprezentacji zajęć oraz przerw.

3. Zmodyfikuj klasę *Break*: ma korzystać z klasy *BasicTerm*, a nie *Term*
4. Zmodyfikuj klasę abstrakcyjną *BasicTimetable*:
    1. Usuń linię `from enum import Enum`
    2. Usuń definicje klas *Action*, *Lesson* oraz *Term*
    3. Zaimportuj klasy *Action*, *Lesson* oraz *Term*, które stworzyłeś/aś na poprzednich zajęciach
    4. Skopiuj (do klasy *BasicTimetable*) wspólny kod (metody oraz własności) klas *TimetableWithoutBreaks* oraz *TimetableWithBreaks*; metody, które są charakterystyczne tylko dla jednej z nich (*TimetableWithoutBreaks* lub *TimetableWithBreaks*) pozostaw jako abstrakcyjne z pustym ciałem
5. Zmodyfikuj definicje klas *TimetableWithoutBreaks* oraz *TimetableWithBreaks* — obydwie mają dziedziczyć z klasy *BasicTimetable*
6. Usuń z każdej z nich kod, który skopiowałeś/aś do klasy *BasicTimetable*, a pozostaw ten, który jest charakterystyczny (indywidualny) tylko dla niej

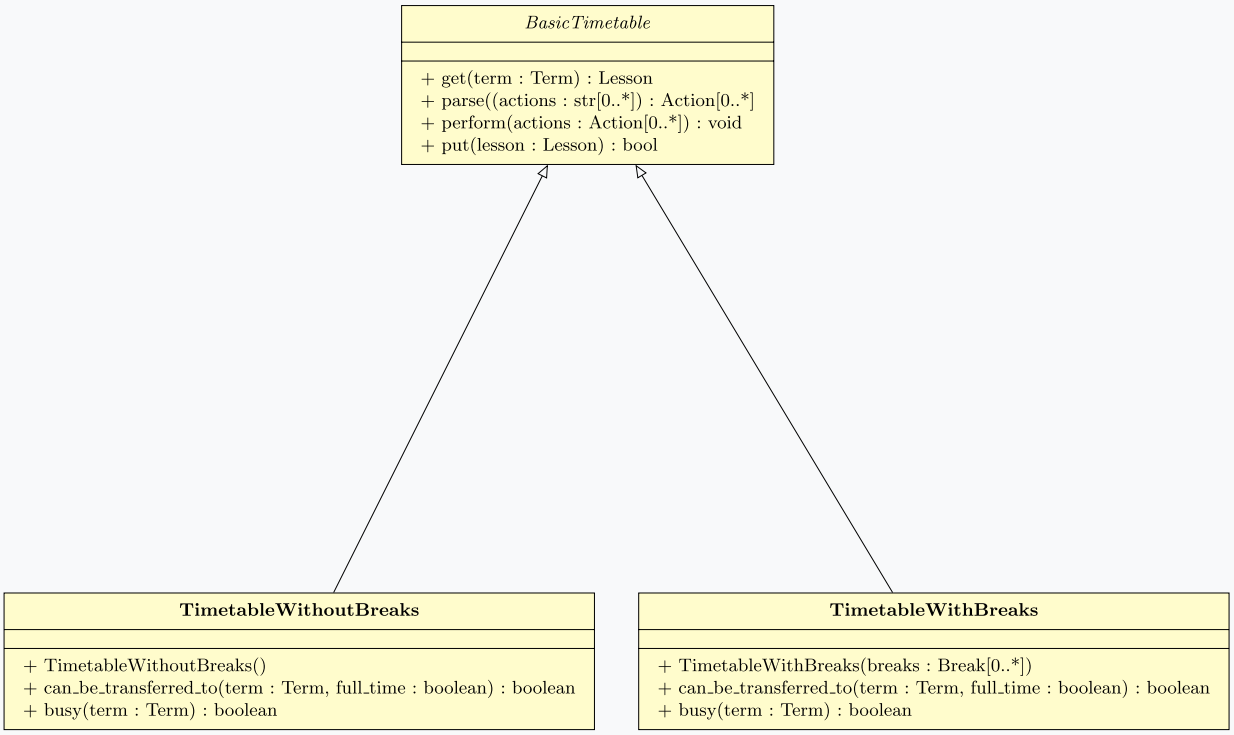

Diagram klas służących do reprezentacji rozkładu zajęć.

7. Użyj poprzednich testów (jednostkowe oraz integracyjne) — sprawdź poprawność implementacji metod

Kompozycja czy dziedziczenie

Po odkryciu polimorfizmu łatwo jest pomyśleć, iż ponieważ jest on tak sprytnym narzędziem, więc należy stosować dziedziczenie wszędzie. Może to mieć jednak negatywny wpływ na nasze projekty. W istocie wybieranie dziedziczenia jako pierwszego sposobu w sytuacji, gdy wykorzystujemy istniejącą klasę dla stworzenia nowej, powoduje, że sprawy niepotrzebnie się komplikują.
Lepszym wyjściem jest, gdy wybór nie jest oczywisty, wybieranie jako pierwszego przybliżenia kompozycji. Kompozycja nie wymusza budowania projektu jako hierarchii dziedziczenia. Jest także bardziej elastyczna, ponieważ możliwa jest dynamiczna zmiana typów (a przez to zachowania) składowych klasy, podczas gdy w przypadku dziedziczenia dokładny typ musi być znany na etapie kompilacji. Ilustruje to poniższy przykład:

```
from abc import ABC, abstractmethod

# Dynamiczna zmiana zachowania obiektu poprzez użycie kompozycji.
class Actor(ABC) :
    @abstractmethod
    def act(self) :
        pass

class HappyActor(Actor) :
    def act(self) :
        print("HappyActor")

class SadActor(Actor) :
    def act(self) :
        print("SadActor")

class Stage :
    def __init__(self):
        self.__actor = HappyActor()

    def change(self) :
        self.__actor = SadActor()

    def performPlay(self) :
        self.__actor.act()

if __name__=="__main__":
    stage = Stage()
    stage.performPlay() # Wypisuje "HappyActor"
    stage.change()
    stage.performPlay() # Wypisuje "SadActor"
```

Obiekt typu *Stage* zawiera referencję do typu *Actor*, która jest inicjalizowana na obiekt klasy *HappyActor* (linia 19). Oznacza to, iż metoda `performPlay()` działa w pożądany sposób. Ponieważ jednak referencja może w czasie wykonywania zostać związana z innym obiektem, możemy więc podstawić do *__actor* referencję do obiektu *SadActor*,

zmieniając dzięki temu zachowanie metody `performPlay()`. Zyskujemy zatem dynamiczną elastyczność czasu wykonania. Dla kontrastu: nie możemy zdecydować się na inny sposób dziedziczenia w czasie wykonywania programu — musi to być jednoznacznie określone w chwili kompilacji (do kodu pośredniego).
Generalną wskazówką jest: "używaj dziedziczenia dla wyrażania różnic w zachowaniu, zaś pól dla wyrażania zmian stanu". W powyższym przykładzie wykorzystuje się obie techniki: stworzone zostały dwie różne klasy potomne dla wyrażenia różnic w zachowaniu się metody `act()`, natomiast obiekt *Stage* wykorzystuje kompozycję dla umożliwienia zmiany swego stanu. W tym przypadku zmiana stanu powoduje zmianę w zachowaniu.

— Treść niniejszego akapitu bazuje na książce *Bruce Eckel, "Thinking in Java - edycja polska", Helion, 2001, str. 255-256*

Porównanie ww. mechanizmów można również znaleźć na stronie blog.helion.pl lub Sarven Dev

4. Wyjątki i ich obsługa, modyfikacja sposobu przechowywania elementów

1 pkt za wykonanie zadań z tej sekcji

1. Zmodyfikuj treść metody `parse()` tak, aby wyrzucała wyjątek *ValueError*, jeżeli lista argumentów programu (z opisem translacji zajęć) zawiera nieprawidłowy parametr, tzn. różny od "d-", "d+", "t-" oraz "t+". Treścią wyświetlanego komunikatu ma być napis postaci: `"Translation " + argument + " is incorrect"`
2. We wszystkich metodach dodających elementy do rozkładu zajęć, sprawdź czy dany termin jest zajęty czy też nie; w przypadku gdy termin jest zajęty, wyrzuć wyjątek *ValueError* z adekwatnym komunikatem błędu
3. Obsłuż wyjątki w obrębie bloku głównego. Obsługa powinna polegać na wyświetleniu komunikatu wyjątku oraz zakończeniu działania programu,
4. Przetestuj swoją implementację, wprowadzając nieprawidłową translację oraz dodając dwa zajęcia w jednakowym terminie
5. Zmodyfikuj inne metody, które Twoim zdaniem powinny wyrzucać wyjątki — użyj tych podklas klasy BaseException , które najbardziej pasują do sytuacji błędu. W przypadku braku adekwatnych klas, utwórz własną dziedziczącą z *BaseException*
6. Implementacja metody `busy()` nie jest wydajna, ponieważ za każdym razem wymaga przejścia przez wszystkie elementy znajdujące się na liście. Można ją poprawić zamieniając listę na słownik — zastąp implementację na liście, implementacją na słowniku — kluczem ma być termin zajęć. W tym przypadku jeżeli istnieje określony klucz w słowniku, to jest to informacja, że dany termin jest zajęty
   - Poprawne działanie słownika uzależnione jest od implementacji metod `__eq__()` oraz `__hash__()` w klasie, która stanowi klucze słownika (w ćwiczeniu dotyczy to klasy *Term*)
   - Wynik działania metody `__hash__()` musi być zgodny z wynikiem działania metody `__eq__()`, tzn. jeśli dwa obiekty są równe według `__eq__()`, to ich `__hash__()` musi być równy
   - Przeczytaj opis modułu dataclasses i zastanów się, czy będzie on przydatny
7. Zmodyfikuj testy — mają sprawdzać poprawność bieżącej implementacji

5. Zadanie

(**2 pkt.**) Rozbuduj kod (z poprzednich ćwiczeń) o nowe klasy oraz metody — zostaną one wyspecyfikowane na początku zajęć