

Programowanie obiektowe - kolokwium 1, 12.05.2023

Po napisaniu kolokwium, należy spakować do formatu zip lub tar.gz katalog z projektem. Proszę wyłączyć niego katalog z wynikami kompilacji (standardowo nazywa się on target) oraz plików konfiguracyjnych środowiska (w IntelliJ Idea standardowo nazywa się .idea). Po zakończeniu kolokwium proszę nie wyłączać komputera.

Podczas kolokwium można korzystać ze wszystkich zasobów internetu, w tym samodzielnie przygotowanych. Podczas kolokwium zabroniona jest wzajemna komunikacja oraz publikowanie i dostęp do kodu stworzonego na potrzeby kolokwium. Podczas kolokwium wszystkie działania wykonywane na ekranie komputera są rejestrowane i mogą zostać obejrzone przez prowadzących. Zabronione jest korzystanie z innych urządzeń niż udostępniony komputer.

Przedstawione poniżej kroki stanowią propozycję kolejności rozwiązywania zadania. Po przeczytaniu całości można zdecydować o rozwiązywaniu w innej kolejności.

Krok 1.

Zdefiniuj rekord lub klasę Point posiadającą dwa publiczne, ostateczne pola - liczby zmiennoprzecinkowe. Liczby te powinny być ustawiane za pomocą konstruktora.

W dalszej części zadania, obiekty klasy Point będą skrótowo nazywane punktami.

Krok 2.

Zdefiniuj klasę Polygon, która będzie posiadała prywatną listę punktów. Lista punktów powinna być ustawiana przez konstruktor. Wielokąt zbudowany jest z krawędzi łączących kolejne punkty w liście oraz ostatni z pierwszym punktem z listy.

W dalszej części zadania, obiekty klasy Polygon będą skrótowo nazywane wielokątami.

Krok 3.

W klasie Polygon zdefiniuj publiczną metodę inside(Point), która zwraca wartość logiczną określającą, czy dany argumentem metody punkt znajduje się w wielokącie, na rzecz którego wywoływana jest metoda. Zaimplementuj w niej algorytm podany pseudokodem:

Dane są: wielokąt *poly* i sprawdzany punkt *point*.

counter = 0

Dla każdej pary (*pa*, *pb*) punktów tworzących krawędź wielokąta *poly*:

Jeżeli *pa.y* > *pb.y*:

 Zamień *pa* z *pb*

Jeżeli *pa.y* < *point.y* < *pb.y*:

d = *pb.x* - *pa.x*

 Jeżeli *d* == 0:

x = *pa.x*

 W przeciwnym razie:

a = (*pb.y* - *pa.y*) / *d*

b = *pa.y* - *a* * *pa.x*

x = (*point.y* - *b*) / *a*

 Jeżeli *x* < *point.x*:

counter++

Zwróć prawdę, jeżeli *counter* jest nieparzysty, a fałsz w przeciwnym przypadku.

Krok 4.

Dodaj do projektu bibliotekę JUnit,. Napisz trzy testy. W każdym stwórz wielokąt oraz punkt - tak aby w kolejnych testach:

- punkt leżał w wielokącie,
- punkt leżał pod wielokątem,
- punkt leżał na prawo od wielokąta.

Uwaga! W przypadku tego oraz pozostałych kroków wymagających testowania, w razie nieumiejętności wykorzystania biblioteki JUnit, część punktów można zdobyć pisząc metody klasy Main, wykonujące działania, które mają zostać przetestowane.

Wyobraźmy sobie dwuwymiarową mapę, na której położenie jest opisane za pomocą współrzędnych x i y , gdzie punkt $(0, 0)$ znajduje się w lewej górnej krawędzi mapy. Mapa jest uproszczonym modelem, nie uwzględniającym krzywizny Ziemi, który może być wykorzystany np. na potrzeby gry komputerowej. Na mapie znajdują się lądy opisane za pomocą wielokątów.

Krok 5.

Napisz klasę Land dziedziczącą po klasie Polygon. Napisz w niej taki sam konstruktor jak w Polygon.

Na mapie znajdują się także miasta. Każde miasto otoczone jest murami w kształcie kwadratu, o ścianach wychodzących na cztery strony świata (parami równoległe do osi układu współrzędnych). Środek miasta leży na przecięciu przekątnych kwadratu.

Krok 6.

Napisz klasę City dziedziczącą po Polygon. W klasie zdefiniuj publiczną ostateczną zmienną center - punkt będący środkiem miasta oraz prywatną nazwę miasta. Napisz publiczny konstruktor, takie dwa argumenty oraz liczbę ziemioprzecinkową - długość ściany muru. Konstruktor powinien zapisać w polu klasy środek miasta oraz nazwę, a w liście punktów odziedziczonej z klasy Polygon, wierzchołki kwadratu stanowiącego mury.

Uwaga! Ten krok da się rozwiązać bez zmiany modyfikatora dostępu do listy wierzchołków wielokąta na chroniony. Można dokonać takiej zmiany, ale takie rozwiązanie nie będzie w pełni punktowane.

Miasta mogą znajdować się na lądzie - nie na wodzie.

Krok 7.

W klasie Land stwórz prywatną listę miast. Napisz metodę addCity(City), która doda miasto do tej listy. Miasto może zostać dodane wyłącznie jeżeli jego środek znajduje się na lądzie. W przeciwnym razie należy rzucić wyjątek RuntimeException, którego metoda getMessage() powinna wyświetlić nazwę miasta.

Krok 8.

Napisz test sprawdzający rzucanie i poprawność nazwy wyświetlanej przez getMessage().

Miasta portowe mają dostęp do wody.

Krok 9.

W klasie City dodaj prywatne pole logiczne port określające, czy miasto jest portowe. Miasto jest miastem portowym, jeżeli co najmniej jeden z wierzchołków kwadratu stanowiącego mury miasta znajduje się poza lądem.

Pole to powinna ustawiać metoda znająca zarówno obiekt City, jak i obiekt Land, na którym istnieje lub do którego dodawane jest miasto. Samodzielnie wybierz odpowiednie położenie definicji i wywołania oraz symbol tej metody.

Lądy i morza są pełne zasobów.

Krok 10.

Zdefiniuj klasę Resource. W klasie Resource zdefiniuj publiczny typ wyliczeniowy Type {Coal, Wood, Fish}. Klasa Resource powinna posiadać dwa publiczne, ostateczne pola: punkt (Point) określający rozmieszczenie pozycję zasobu na mapie oraz typ (Type) zasobu. Napisz konstruktor ustawiający te pola.

Krok 11.

Założmy bez sprawdzenia, że węgiel i drewno znajdują się na lądzie, a ryby w wodzie.

W klasie City utwórz zbiór (Set) obiektów Resource.Type o nazwie resources i dostępie pakietowym. W tej samej klasie napisz metodę addResourcesInRange, która przyjmie listę obiektów Resource oraz liczbę zmiennoprzecinkową range i umieści w zbiorze typy tych zasobów, które znajdują się w odległości nie większej niż range od środka miasta. Ryby powinny być uwzględniane wyłącznie w miastach portowych.

Krok 12.

Napisz klasę CityTest zawierającą sparymetryzowany test zawierający:

- poprawne dodanie węgla,
- nieudaną próbę dodania drewna spoza zasięgu miasta,
- poprawne dodanie ryb do miasta portowego,
- nieudaną próbę dodania ryb do miasta nieportowego.

Test będzie wymagał definicji miasta śródlądowego i morskiego, co można wykonać na wiele sposobów, np. przez ustawienie ich jako pola klasy.

Test powinien przyjmować argumenty: miasto, zasób, oczekiwana wartość logiczna.

Sposób parametryzacji testu jest dowolny. Test, przy użyciu pakietowego dostępu do pola City.resources powinien sprawdzać, czy testowany zasób został dodany do zbioru.

Dany jest plik map.svg zawierający przykładową mapę. Mapa zawiera niebieski prostokąt reprezentujący wodę, który można pominąć w dalszych rozważaniach oraz za pomocą znaczników:

- *polygon o kolorze zielonym - ląd,*
- *rect o kolorze czerwonym - miasto,*
- *circle o kolorze czarnym, brązowym, jasnoniebieskim - zasoby, odpowiednio węgiel, drewno, ryby.*
- *text - nazwy miast.*

Symbole zasobów nie będą rozważane w dalszej części zadania.

Krok 13.

Zapoznaj się z dołączoną klasą MapParser służącą do parsowania tego pliku. Klasa wykorzystuje bibliotekę Jackson Dataformat XML. Dołącz ją do projektu. W klasie MapParser znajduje się lista obiektów typu Label zawierającego reprezentację znaczników <text>. Metody parse i parseText prezentują proces konwersji zawartości znacznika na obiekt.

W klasie MapParser stwórz analogiczne do istniejącej listy nazw, nowe listy lądów i miast i zapełnij je wzorując się na istniejącym kodzie. Napisz publiczny akcesor do listy lądów.

Uwaga! Podczas wczytywania miasta jego nazwa nie jest znana. Można ją tymczasowo ustawić jako null. Wartość x i y w znaczniku rect definiuje lewy, górny wierzchołek prostokąta. Pamiętaj o obliczeniu pozycji środka miasta.


Krok 14.

W klasie MapParser zaimplementuj metodę addCitiesToLands. Metoda powinna pracować na liście lądów i miast w obiekcie klasy MapParser. Dla każdego lądu, dodać do jego prywatnej listy miast, miasta, których środki znajdują się na tym lądzie.


Krok 15.

W klasie MapParser zaimplementuj metodę matchLabelsToTowns. Metoda powinna dla każdego miasta w liście miast przyporządkować najbliższy obiekt Label. Konieczny będzie publiczny mutator nazwy miasta w klasie City.

Krok 16.

W klasie Main utwórz obiekt MapParser i wywołaj jego metodę parse dla dołączonej w pliku svg mapy. W klasie City napisz metodę toString, która wyświetli nazwę miasta oraz symbol , jeżeli jest to miasto portowe. W klasie Land napisz metodę toString, która wywoła metody toString wszystkich miast na tym lądzie i zwróci ich wynik w postaci napisu połączonych przecinkami.

Wynik działania takiej funkcji, dla wszystkich lądów z mapy będzie wyglądał następująco:

Kryształowiec, Złoty Horyzont 

Słoneczna Przystań , Srebrzysko, Kamienna Brama, Księżycowe Miasto, Prześwitnia, Pustelnia 

Mglisty Port , Złote Wrzosowiska, Morska Ostoja , Diamentowa Wieża, Różany Las

Martwe Ogrody, Czarne Mokradła, Smoczy Kraniec 

Burzowe Wybrzeże , Żelazna Oaza, Granitowe Wzgórza

Centra miast Wodny Kamień i Zatoka Nadziei zostały umieszczone na wodzie i dlatego nie zostały dodane do żadnego z lądów.