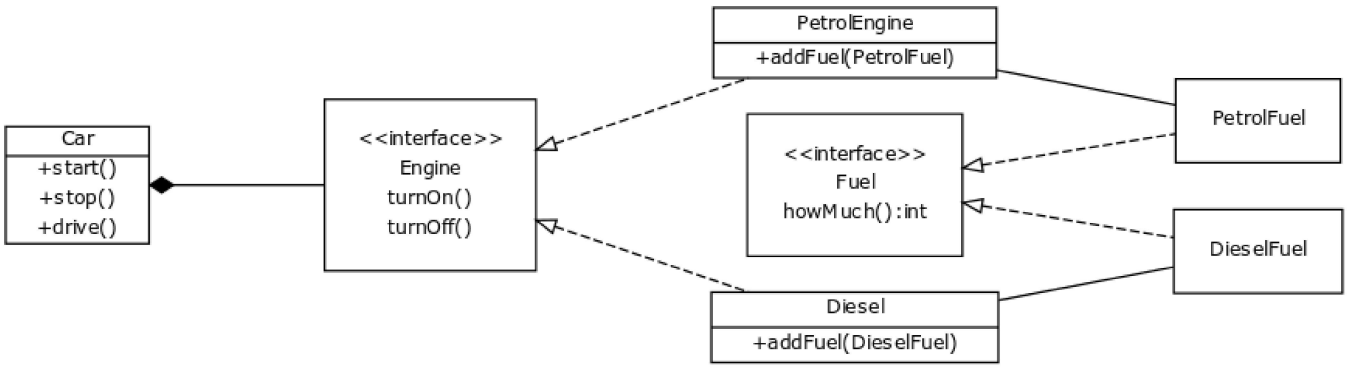
Zadanie 8.2 Nieudana wycieczka

Tym razem spróbujemy napisać program według przygotowanego wcześniej schematu, tzw. **diagramu klas** i zasymulujemy nie do końca udaną wycieczkę samochodem.

1. Zapoznaj się z informacjami o diagramach klas i zaimplementuj rozwiązanie zgodnie z poniższym schematem:



Jeśli któraś z napisanych metod jest pusta, dodaj w niej jedynie wywołanie `System.out.println` z stosowną informacją, tak by widzieć że została wywołana.

1. Dodaj obsługę błędów: gdy ktoś spróbuje wywołać metodę `drive()` na samochodzie, który nie został wcześniej uruchomiony (metodą `start()`) lub gdy w baku nie ma paliwa, samochód powinien rzucić wyjątkiem.
2. Zdefiniuj block `try..finally..catch` tak by niezależnie od tego, czy metoda `drive()` zakończyła się sukcesem czy nie, samochód na koniec gasił silnik (metoda `stop()`).
3. (*) Zastosuj interfejs `Autocloseable` oraz użyj mechanizmu try-with-resources zamiast bloku `try..finally..catch`.

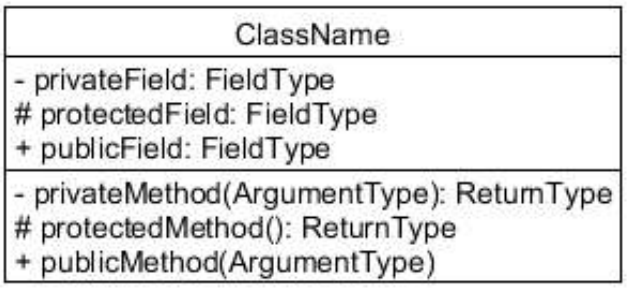
Przydatne informacje

Diagramy klas

Diagramy klas pomagają w zrozumieniu obiektowego kodu oraz w jego projektowaniu. Na diagramie można też zauważyć problemy w przyjętych rozwiązaniach (np. kod będzie się trudno rozwijać lub utrzymywać).

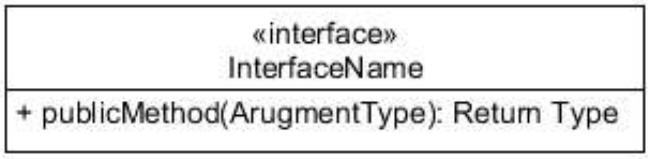
Klasy

Głównym elementem diagramu klas są… klasy. Reprezentuje się je w postaci prostokątów:



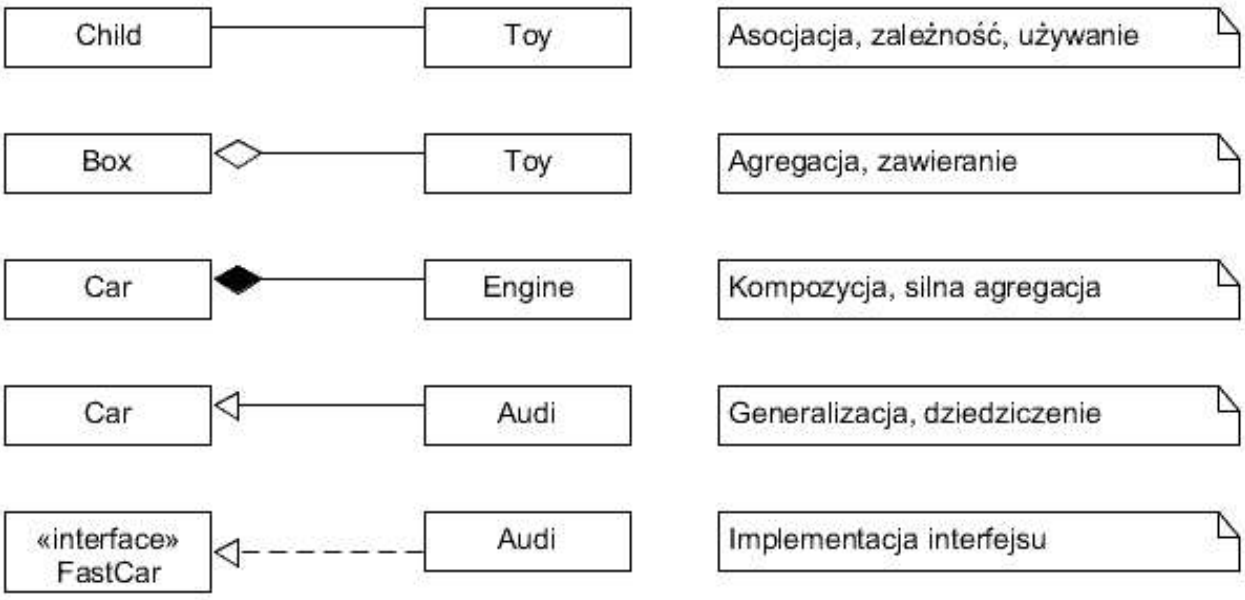
Interfejsy

Interfejs wygląda bardzo podobnie do klasy. Jedyna różnica to taka, że nad nazwą interfejsu powinien znaleźć się stereotyp `<<interface>>` :



Zależności pomiędzy klasami

Żaden system stworzony w modelu obiektowym nie składa się z jednego elementu. Występuje w nim wiele klas, na podstawie których tworzone są obiekty, które korzstając nawzajem ze swoich funkcjonalności dostarczają w pełni działający system. To, co jest najcenniejsze w diagramie klas to właśnie możliwość zobrazowania zależności występujących między klasami w obiektowym systemie. Poniżej zamieszczono notację podstawowych zależności między klasami, które występują w obiektowo zaimplementowanym systemie:



Blok *finally*

Czasem podczas obługi błędów zależy nam by jakieś instrukcje wykonały się bez względu na to, czy błąd poleciał czy nie. Klasycznym przykladem jest tu obsługa otwartego pliku:

```
File file = new File("data.txt");
Scanner scanner = new Scanner(file);

// analiza danych z scannera, tu leci blad

scanner.close();
```

Jeśli w trakcie analizy danych wczytanych z pliku poleciałby bląd i przerwalibyśmy przez to pracę z plikiem, plik pozostałby otwarty i zablokowany dla innych aplikacji. Niezależnie od sytuacji chcielibyśmy **zawsze** wywołać metodę `close()`. W tym celu pomoże nam blok `finally`:

```
File file = new File("data.txt");
Scanner scanner = new Scanner(file);
try {
  // analiza danych z scannera, tu leci blad
} finally {
   scanner.close();
}
```

Blok `finally` występuje w kombinacji z `try` lub nawet `try..finally..catch` i daje nam gwarancję, że kod, który w nim zawrzemy wykona się zawsze. Oczywiście nie uchroni nas przed naszymi błędami - w trakcie wykonywania `finally` również możemy doprowadzić do błędu...