POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

PJC Zadania 8

Rozwiązania należy przesłać w postaci odpowiednio podzielonych plików o rozszerzeniach `.hpp` i `.cpp`.

Zadanie 1

Mechanizm dziedziczenia może być wykorzystywany do realistycznego mapowania abstrakcji wziętych z życia codziennego na modele programistyczne (klasy) powiązane odpowiednimi relacjami. Rozważmy hierarchię dziedziczenia reprezentującą figury geometryczne, która się składa z:

- Klasy `shape`, będącą klasą bazową do reszty figur geometrycznych. Zdefiniuj w niej następujące metody **czysto wirtualne** (ang. *pure virtual*):
  - `get_area()` zwracającą pole danej figury;
  - `get_perimeter()` zwracającą obwód danej figury.

  Metody czysto wirtualne są odpowiednikiem **metod abstrakcyjnych** (oznaczanych modyfikatorem `abstract` w Javie) i deklaruje się je wpisując `= 0;` zamiast ich ciała (klamer).
- Klasy `rectangle` dziedziczącej po `shape`, która będzie miała dodatkowo **prywatne** pola `width` oraz `height`, których **inicjalizację** należy umożliwić za pomocą konstruktora.
- Klasy `circle` dziedziczącej po `shape`, która będzie miała dodatkowo **prywatne** pole `radius`, którego **inicjalizację** należy umożliwić za pomocą konstruktora.

Filip Kwiatkowski, fk@pjwstk.edu.pl

- Klasy `triangle` dziedziczącej po `shape`, która będzie miała dodatkowo **prywatne** pola `height` obraz `base`, których **inicjalizację** należy umożliwić za pomocą konstruktora.

Dla każdej z klas dziedziczących w tej hierarchii nadpisz odpowiednio metody odziedziczone z `shape`. Zwróć uwagę na to, żeby implementacje działały nawet wtedy, jeżeli ktoś zrobi niezmienny (`const`) obiekt jakiejś figury geometrycznej. W celu upewnienia się, że metody są poprawnie nadpisywane, skorzystaj w odpowiedni sposób z identyfikatora `override`.

Dodatkowo napisz funkcję `print_shape_info()`, która będzie mogła przyjąć dowolny z typów wymieniony wyżej. Za pomocą metod wirtualnych ma ona wyświetlić wszystkie dostępne informacje o dostarczonym obiekcie.

Zadanie 2

Dziedziczenie może zostać użyte nie tylko do reprezentowania relacji *jest / być* w modelowaniu abstrakcji z życia na kod – może też zostać użyte jako mechanizm odpowiedniego doboru implementacji w kontekście wzorca projektowego Dependency Injection.

Wzorzec ten pozwala nam uprościć zależności w kodzie (minimalnym kosztem przygotowania większego systemu na początku). Oddelegowuje pracę, której wymaga część naszego systemu do innych części, które oferują jakieś funkcjonalności. W Javie bardzo wygodnie się projektuje zależności za pomocą `interface`'ów, ale w C++ ich nie mamy – dlatego musimy skorzystać po prostu z klas.

Na podstawie podanej klasy `collector`:

```
struct collector {
    virtual void collect(const char to_collect) = 0;
};
```

Listing 1: Klasa collector

Stwórz hierarchię dziedziczenia, w której cztery nowe klasy: `string_collector`, `vector_collector`, `reverse_string_collector` oraz `reverse_vector_collector` będą implementowały w odpowiedni sposób (nawiązując do nazwy tej klasy) *zbieranie* przekazanych przez metodę `collect()` elementów.

Zbieranie elementów winno polegać na tym, że zostają one w odpowiedni sposób dodane do **wewnętrznej, prywatnej reprezentacji** danego kolektora, którego kopię można otrzymać za pomocą metody `representation()`.

Aby użyć kolektorów zaimplementuj funkcję `filter_hyphens_from()`, która przyjmie `std::vector<char>` i zwróci obiekt przechowujący wszystkie znaki z przekazanego wektora z wyjątkiem znaku myślnika (`'-'`), na którym to obiekcie będziemy mogli wywołać metodę `collect_using()`, która, dostając przez argument dostęp do dowolnego kolektora, zbierze wszystkie elementy z przefiltrowanego wektora i udostępni je kolektorowi.

Przykłady użycia:

```
int main() {
    auto vector = std::vector<char>{
            'a', '-', 'l', '-', '-', 'a', '-',
            ' ', '-',
            'm', 'a',
            '-', ' ', '-',
            '-', 'k', '-', '-', 'o', 't', '-', 'a', '-', '-'
    };

    auto str_collector = string_collector();

    filter_hyphens_from(vector).collect_using(str_collector);

    std::cout << '\n' << str_collector.representation() << '\n';
}
```

Jak widać, tworzymy wpierw wektor wypełniony znakami. Następnie przygotowujemy *strategię zbierania tych elementów*, a wybór pada na kolektor zbierający do stringa (`string_collector`). Następnie na obiekcie przefiltrowanych znaków wywołujemy *funkcję zbierającą* (`collect_using`), która zbierze wszystkie (przefiltrowane) elementy za pomocą przekazanego kolektora. W ten sposób **wstrzykujemy** zależność (implementację strategii) do naszego serwisu zbierającego.

Na koniec wyświetlamy reprezentację tego kolektora, która jest typu `std::string`.

Kod ten wyświetla: `ala ma kota`

Przykład 1

```
int main() {
    auto vector = std::vector<char>{
            'a', '-', 'l', '-', '-', 'a', '-',
            ' ', '-',
            'm', 'a',
            '-', ' ', '-',
            '-', 'k', '-', '-', 'o', 't', '-', 'a', '-', '-'
    };

    auto str_collector = reverse_string_collector(); // inny kolektor

    filter_hyphens_from(vector).collect_using(str_collector);

    std::cout << '\n' << str_collector.representation() << '\n';
}
```

Jak widać, jedyną różnicą między tym przykładem a poprzednim jest to, że używamy po prostu innego kolektora. Teraz wybór padł na strategię zbierania do stringa w odwrotnej kolejności – za pomocą `reverse_string_collector`.

Kod ten wyświetla: `atok am ala`

Przykład 2

```
int main() {
    auto vector = std::vector<char>{
            'a', '-', 'l', '-', '-', 'a', '-',
            ' ', '-',
            'm', 'a',
            '-', ' ', '-',
            '-', 'k', '-', '-', 'o', 't', '-', 'a', '-', '-'
    };

    auto vec_collector = vector_collector(); // inny kolektor

    filter_hyphens_from(vector).collect_using(vec_collector);

    for (const auto& item : vec_collector.representation()) {
        std::cout << item;
    }
}
```

Ten przykład jest praktycznie identyczny, choć wewnętrznie działa zupełnie inaczej – nasz kolektor tym razem zbiera elementy do wektora (stąd pętla – wektora nie wyświetlimy tak jak stringa). Proszę zwrócić uwagę, jak bardzo podobny jest kod i jak prosto zmieniać strategię implementacji, kiedy korzysta się z Dependency Injection.

Kod ten wyświetla: `ala ma kota`

Przykład 3

```
int main() {
    auto vector = std::vector<char>{
            'a', '-', 'l', '-', '-', 'a', '-',
            ' ', '-',
            'm', 'a',
            '-', ' ', '-',
            '-', 'k', '-', '-', 'o', 't', '-', 'a', '-', '-'
    };

    auto vec_collector = reverse_vector_collector(); // inny kolektor

    filter_hyphens_from(vector).collect_using(vec_collector);

    for (const auto& item : vec_collector.representation()) {
        std::cout << item;
    }
}
```

Kod ponownie jest praktycznie identyczny, ale tym razem wybraliśmy strategię zbierania do wektora w odwrotnej kolejności.

Kod ten wyświetla: `atok am ala`

Przykład 4

Wzorzec projektowy Dependency Injection, o ile wymaga więcej kodu służącego za przygotowanie interfejsów do współpracy z różnymi strategiami, cechuje się trywialnością korzystania z nowych strategii. Implementacja nowego kolektora jest prosta, a korzystanie z niego absolutnie trywialne – wystarczy przekazać go przez argument do już gotowego systemu.