Paradygmaty programowania obiektowego

Wykorzystanie paradygmatóww programowania obiektowego w kolekcjach

dr inż. Radosław Idzikowski

Przegląd paradygmatów programowania obiektowego

- Podejście do tworzenia programów, w którym obiekty stanowią podstawową jednostkę strukturalną.
- Główne cechy:
  - Enkapsulacja: Obiekty grupują dane i metody, które na nie działają, w jedną strukturę. Ukrywają wewnętrzne implementacje przed innymi obiektami.
  - Dziedziczenie: Pozwala na tworzenie hierarchii klas, w której klasy pochodne dziedziczą zachowanie i właściwości od klas nadrzędnych.
  - Polimorfizm: Pozwala na przesłanianie metod w klasach pochodnych oraz na korzystanie z różnych implementacji tych samych metod w zależności od kontekstu.
- Zalety:
  - Modularyzacja: Obiekty można łatwo ponownie używać i modyfikować, co prowadzi do bardziej modułowego kodu.
  - Łatwiejsze zarządzanie złożonością: Dzięki enkapsulacji i dziedziczeniu, złożone problemy można łatwiej rozbić na mniejsze, bardziej zrozumiałe części.
  - Wysoka abstrakcja: umożliwienie tworzenia bardziej abstrakcyjnych modeli, które odzwierciedlają rzeczywiste relacje między obiektami.

Zależności między paradygmatami a kolekcjami

- Paradygmat programowania obiektowego i kolekcje mają ściśle powiązane relacje, ponieważ kolekcje są często wykorzystywane w programowaniu obiektowym do przechowywania i zarządzania grupami obiektów.
- Kolekcje umożliwiają grupowanie obiektów o podobnych cechach i zachowaniu, co jest zgodne z ideą enkapsulacji i dziedziczenia.
- Kolekcje dostarczają mechanizmy dodawania, usuwania, wyszukiwania i iteracji po elementach, które odpowiadają operacjom, jakie można wykonywać na obiektach.
- Kolekcje mogą być również wykorzystywane do implementacji wzorców projektowych, takich jak Iterator, Obserwator czy Fabryka.
- Paradygmat programowania obiektowego dostarcza zasady projektowania, które mogą być stosowane przy tworzeniu i manipulacji kolekcjami, takie jak hermetyzacja, polimorfizm i abstrakcja.

Przypomnienie różnych typów kolekcji dostępnych w języku C++

- Język C++ dostarcza wiele wbudowanych typów kolekcji, które mogą być wykorzystane w programowaniu obiektowym, takich jak:
  - Tablice (`Array`): Jednowymiarowe i wielowymiarowe tablice, umożliwiające przechowywanie elementów tego samego typu w sekwencyjnym porządku.
  - Wektory (`Vector`): Dynamiczne tablice, które automatycznie dostosowują swoją pojemność do liczby przechowywanych elementów.
  - Listy (`List`): Dwukierunkowe listy, które umożliwiają dodawanie i usuwanie elementów w dowolnym miejscu listy.
  - Kolejki (`Queue`): Kolejki FIFO (First-In-First-Out), które obsługują operacje dodawania na końcu kolejki i usuwania z początku kolejki.
  - Stosy (`Stack`): Stosy LIFO (Last-In-First-Out), które obsługują operacje dodawania i usuwania na wierzchołku stosu.
  - Mapy (`Map`): Skojarzenia klucz-wartość, gdzie każdy klucz jest unikalny i przypisany do odpowiadającej mu wartości.
  - Zbiory (`Set`): Kolekcje unikalnych elementów, które nie mają określonego porządku.

Wzorzec Iterator i jego zastosowania w C++

- Wzorzec Iterator jest używany do sekwencyjnego dostępu do elementów kolekcji bez ujawniania jej wewnętrznej struktury.
- W C++ wzorzec Iterator jest często implementowany jako klasa, która dostarcza interfejs do przechodzenia przez elementy kolekcji.
- Przykładowe zastosowania wzorca Iterator w C++ to:
  - Przechodzenie przez elementy kontenerów standardowej biblioteki C++, takich jak wektory, listy, mapy itp.
  - Implementacja niestandardowych kontenerów i dostarczanie interfejsu iteracyjnego.
  - Dostęp do elementów wewnętrznych struktur danych, takich jak drzewa, grafy czy listy.

```
#include <iostream>
#include <vector>
#include <list>
template <typename T>
class Iterator {
public:
    virtual T next() = 0;
    virtual bool hasNext() = 0;
};
```

```
template <typename T>
class VectorIterator : public Iterator<T> {
private:
    std::vector<T> collection;
    int currentIndex;

public:
    VectorIterator(const std::vector<T>& coll) :
    collection(coll), currentIndex(0) {}

    T next() override {
        return collection[currentIndex++];
    }

    bool hasNext() override {
        return currentIndex < collection.size();
    }
};
```

```
template <typename T>
class ListIterator : public Iterator<T> {
private:
    std::list<T> collection;
    typename std::list<T>::iterator currentIterator;

public:
    ListIterator(const std::list<T>& coll) :
        collection(coll), currentIterator(collection.begin()) {}

    T next() override {
        return *(currentIterator++);
    }

    bool hasNext() override {
        return currentIterator != collection.end();
    }
};
```

```
template <typename T>
void printCollection(Iterator<T>& iterator) {
    while (iterator.hasNext()) {
        std::cout << iterator.next() << "␣";
    }
    std::cout << std::endl;
}
```

Wzorzec Iterator i jego zastosowania w C++

Wywołanie Iteratora na różnych kolekcjach

```
int main() {
    std::vector<int> numbersVec = { 1, 2, 3, 4, 5 };
    VectorIterator<int> vecIterator(numbersVec);
    std::cout << "Vector collection: ";
    printCollection(vecIterator);

    std::list<std::string> namesList = { "Alice", "Bob", "Charlie", "Dave" };
    ListIterator<std::string> listIterator(namesList);
    std::cout << "List collection: ";
    printCollection(listIterator);

    return 0;
}
```

Optymalizacja wydajności operacji na kolekcjach w C++

Optymalizacja wydajności operacji na kolekcjach w języku C++ jest istotna, zwłaszcza gdy mamy do czynienia z dużymi zbiorami danych lub algorytmami, które wykonują częste operacje na kolekcjach. W tym obszarze istnieje wiele technik i praktyk, które można zastosować w celu zwiększenia wydajności operacji na kolekcjach.

- wybór odpowiedniej kolekcji,
- minimalizacja kopiowania danych,
- użycie iteratorów,
- wykorzystanie algorytmów biblioteki standardowej,
- rezerwowanie pamięci,
- używanie odpowiednich funkcji i operatorów,
- unikanie niepotrzebnych operacji i obliczeń.

Sortowanie obiektów

Klasa Animal

```
enum class AnimalType {
    Cat,
    Dog,
    Hamster
};

class Animal {
public:
    Animal(int id, int age, AnimalType type) : id(id), age(age), type(type) {}
    int getId() const { return id; }
    int getAge() const { return age; }
    AnimalType getType() const { return type; }
private:
    int id;
    int age;
    AnimalType type;
};
```

Sortowanie obiektów

Wywołanie

```
int main() {
    const int N = 100;
    std::vector<Animal> animals;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> ageDistribution(2, 15);
    std::uniform_int_distribution<int> typeDistribution(0, 2);

    for (int i = 0; i < N; ++i) {
        int age = ageDistribution(gen);
        AnimalType type = static_cast<AnimalType>(typeDistribution(gen));
        animals.push_back(Animal(i, age, type));
    }

    std::sort(animals.begin(), animals.end(), [](const Animal& a1, const Animal& a2)
        return a1.getAge() < a2.getAge();
        });

    return 0;
}
```

Podsumowanie

- Kolekcje są podstawowym narzędziem do przechowywania i manipulowania danymi w programowaniu.
- Wykorzystanie paradygmatu programowania obiektowego w kolekcjach umożliwia tworzenie bardziej czytelnego, elastycznego i skalowalnego kodu.
- Wzorce projektowe, takie jak Iterator czy Fabryka, mogą być używane w kontekście kolekcji, aby zwiększyć ich funkcjonalność i elastyczność.
- Paradygmaty programowania obiektowego wraz z odpowiednim wyborem i optymalizacją kolekcji stanowi potężne narzędzie w tworzeniu efektywnych i skalowalnych aplikacji.