РЕАЛІЗАЦІЯ ПАТТЕРНУ VISITOR У МОВІ C++ ЗА ДОПОМОГОЮ STD::VARIANT

Р. Попов, Н. Карпенко
Дніпровський національний університет імені Олеся Гончара
popov_ro@ffeks.dnu.edu.ua

Під час розробки компіляторів та інтерпретаторів виникає проблема організації та моделювання вузлів дерева абстрактного синтаксису (ДАС). Ці програми можуть виконувати різноманітні дії над ДАС. Наприклад, компілятор може проводити оптимізацію програми (результатом такої дії буде нове дерево), інтерпретатор може виконати програму, наприклад, функція може повернути значення, але може й викликати побічні ефекти, наприклад, зміну стану об’єкту, виведення інформації в консоль або лог-файл тощо. Отже, постає проблема кодування дерева таким чином, щоб можна було легко додавати функції для обробки цього дерева. Ця проблема у зарубіжній літературі має назву “expression problem” [1].

Зазвичай у мовах об’єктно-орієнтованого програмування, таких як C++ чи Java, ДАС має вигляд ієрархії класів. Методи роботи над цим деревом перенесені на спеціальні класи, які реалізують інтерфейс Visitor. Visitor — це патерн, за допомогою якого можна додавати нові функції до класів не змінюючи їх. Виходить, що дані ДАС та код, що працює над ним, розділені на два класи: вузол ДАС та відвідувач [2].

Для реалізації патерну Visitor у мові C++ спочатку створюють абстрактний клас Visitor, в якому описують віртуальні методи роботи над кожним вузлом ДАС (visitInteger, visitBinOp тощо). Після цього певний функціонал обертається у клас, який реалізує цей інтерфейс. У кожного вузла ДАС має бути віртуальний метод accept для прийому посилання на екземпляр класу Visitor та виклику в ньому методу visit, який відповідає цьому вузлу ДАС. Таким чином увесь функціонал роботи над ДАС розподілений на конкретні класи, отже не виникає потреби змінювати визначення вузлів і можна легко додавати нові методи.

Недоліком такого підходу є те, що з додаванням нового вузла ДАС потрібно кожен раз виконати цілий комплекс дій в наступному порядку:

1) створити віртуальний метод visit, який відповідає новому вузлу;
2) у вузлі реалізувати метод accept;
3) перевірити коректність використання типів перед їх реалізацією;
4) додати новий функціонал.

Виникає питання: чи можна відтворити цей патерн легше? Відповідь на це дає стандарт мови C++, який описує сучасну типово-безпечну реалізацію об’єднання (Std::variant). При цьому доступ до змінних компонентів здійснюється через виклик функції std::visit, яка приймає Callable об’єкт, в якому мають бути функції для роботи з відповідним типом даних, який збережений в std::variant [3].

Запропонований спосіб реалізації патерну Visitor у мові C++ полягає в тому, щоб представити абстрактні вузли ДАС (такі як Declaration, Statement, Expression) за допомогою std::variant. Певний функціонал над вузлами ДАС реалізується як окремий клас, що містить функції, які приймають аргументи усіх вузлів ДАС.

Перевагами цього способу є використання вбудованих можливостей мови C++, при цьому немає потреби в створенні окремого інтерфейсу Visitor, не треба змінювати вузли ДАС і створювати метод accept. Створення нового вузла полягає в створені

окремого класу та додаванні його в std::variant. Оскільки виклик необхідних функцій для роботи зі збереженими даними реалізований автоматично за допомогою системи типів, то програміст убережений від орфографічних помилок, які можуть виникнути при реалізації методів visit.

Але найбільшою перевагою запропонованого підходу є те, що методи, які працюють над ДАС, можуть повертати будь-який тип даних. За класичною реалізацією патерна Visitor, усі функції visit є віртуальними, а отже над ними не може бути створений шаблон. Оптимізатор не може повернути нове дерево, інтерпретатор не може повернути результат обчислень, тому вони вимушені зберігати усі результати у полі свого класу.

У цього методу є декілька недоліків, на які слід звернути увагу перед використанням:

- ✓ std::variant доступний лише зі стандарту C++17;
- ✓ std::variant реалізований за допомогою шаблонного метапрограмування, тому компілятор, у разі яких-небудь помилок у коді, може видавати дивні та складні повідомлення про помилки;
- ✓ типи Statement або Expression перестають бути абстрактними поліморфними класами, натомість вони стають об’єднаннями, тому програма може використовувати більше пам’яті ніж потрібно (але насправді компілятор Rust та CPython реалізують ДАС саме таким чином);
- ✓ треба бути акуратним при створенні вузлів ДАС, їх не можна використовувати всередині вузла std::variant, до якого він належить, оскільки виникне класична проблема використання структури усередині самої себе.

Таким чином, у даній публікації був розглянутий класичний спосіб відтворення ДАС у мові C++ за допомогою патерна Visitor та запропоновано новий спосіб кодування цього патерну, який є швидшим у написанні та надає більше можливостей.

[1] Expression problem [Електронний ресурс]. Режим доступу: https://wiki.c2.com/?ExpressionProblem.
[2] Visitor in C++ / Design Patterns [Електронний ресурс]. Режим доступу: https://refactoring.guru/design-patterns/visitor/cpp/example.
[3] std::variant - cppreference.com [Електронний ресурс]. Режим доступу: https://en.cppreference.com/w/cpp/utility/variant.

IMPLEMENTATION OF VISITOR PATTERN IN C++ BY USING STD::VARIANT

R. Popov, N. Karpenko
Oles Honchar Dnipro National University
popov_ro@ffeks.dnu.edu.ua

In this paper we presented a new way to implement the Visitor pattern in C++ by using the standard container std::variant.

We have shown how this method solves the expression problem and simplifies the creation of compilers and interpreters.

We compared the classical approach with ours and highlighted all the pros and cons of the new solution.