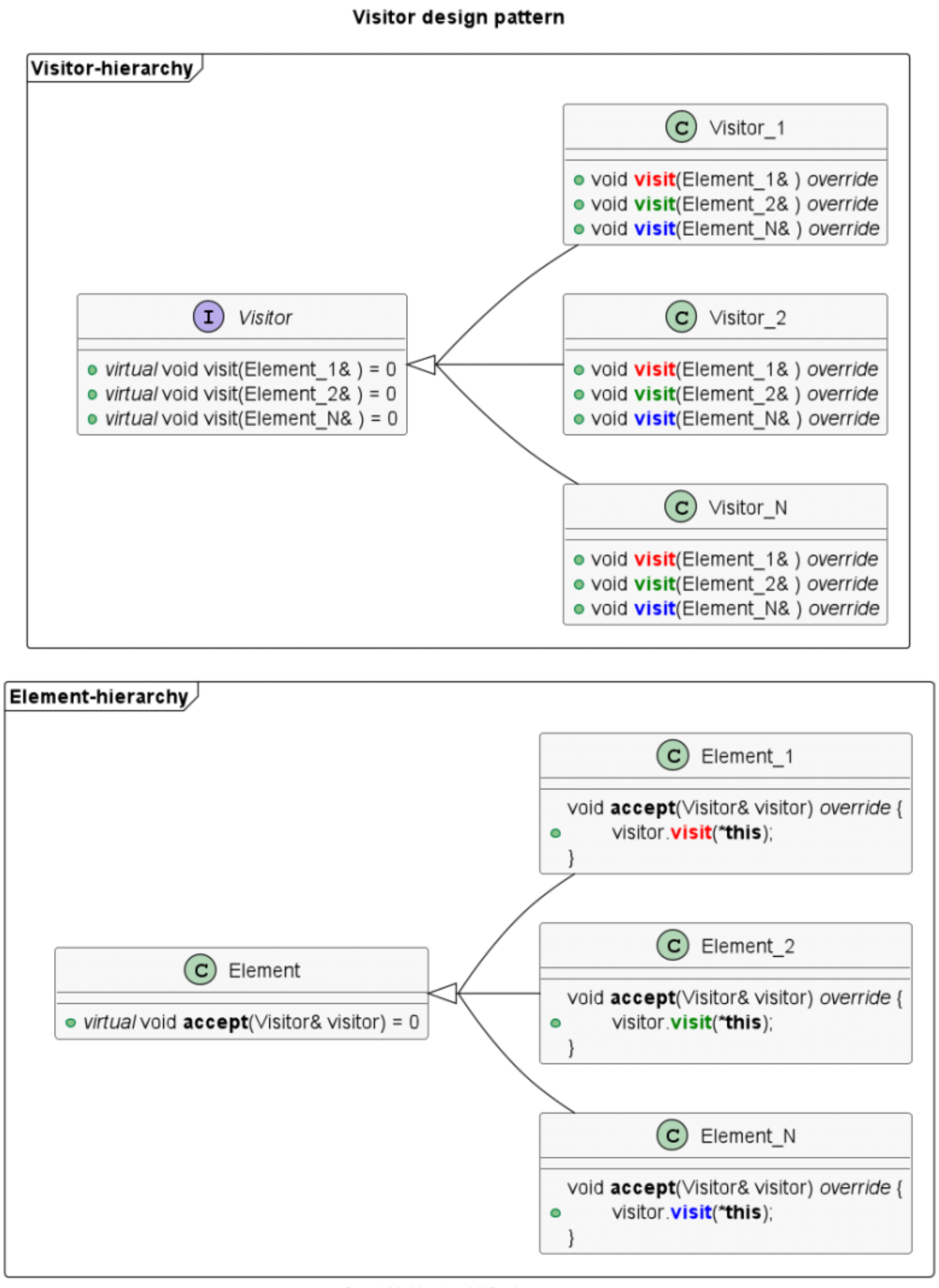
Visitor

Sonntag, 11. Juni 2023 16:45

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Intro

The classical, Erich Gamma and company implementation of the *Visitor design pattern* is kind of boring (sorry for being disrespectable), and can be illustrated with the following class diagram



But it is useful to explain the basic idea behind: **separation of the data structure from the algorithms - operations** that can be applied on the visitation, on each instance of Element-hierarchy.
This way, we can add a new operation (a *Visitor*) without affecting the Element-hierarchy.
This is also in line with *Open-Closed Principal*: we extending the operations set, without modifying the nodes in Element-hierarchy.

@note Other way around doesn't work: adding a new element into Element-hierarchy requires adding a new handling (a new *visit()* overloading) in Visitor-hierarchy and therefore refactoring all derived - concreate Visitors.
This is why the Visitor pattern is usually applied on a relatively stabled topologies - Element-hierarchies.

To be able to apply different operations on the same element, the Element interface has a single gateway method *accept()* - as entry point of visitation.
This is also known as **double-dispatch mechanism**, since we vary on two different hierarchies:

- Element-hierarchy: on which derived reference is *accept* called, and therefore passed to the Visitor (red/green/blue *visit* overloading)
- Visitor-hierarchy: which Visitor - concreate operation is applied (Visitor_1, Visitor2,..., Visitor_N) on a given element

To emphasize this, we can even write a helper free-function [1)]

```
void dispatch(Element& element, Visitor& visitor) {
    element.accept(visitor);
}
```

Implementation

For the classical approach, you could represent your topology as

```
class ListOfElements
{
    public:
        ListOfElements(std::initialized_list<std::unique_ptr<Element>> elements) noexcept:
            elements_(elements)
        {}

        // Visiting all elements
        void accept(Visitor& visitor)
        {
            for(auto& element :elements_)
            {
                element->accept(visitor);
            }
        }
    private:
        std::vector<std::unique_ptr<Element>> elements_;
};
```

Starting with C++17, there is more convenient way of modeling the Visitor pattern, using discriminated unions: **std::variant**.
Now we can represent our Element-hierarchy quite compound

```
using element_type = std::variant<Element_1, Element_2,..., Element_N>;
using element_types = std::vector<element_type>;
```

There is no need more for the single entry-point: *accept* call, nor double-dispatching.
We use it combined with another std library algorithm: **std::visit - which also work with concreate Visitors (we use value semantic here).**

```
element_types elements;
std::visit(Vistor_1{}, elements);
```

We can even be generic, and provide the compile-time version, for Visitor-hierarchy (the one that usually expends)

```
 // Generic: compile-time visitor

template <typename...Es>
struct Visitor;// declaration, without definitions

// Specializations

template <typename E, typename...Es>
struct Visitor<E,Es...> : Visitor<Es>...
{
    using Visitor<Es>::visit...; // this will unroll all the base-classes visitors matching visit() call implementations
    virtual void visit(E& ) = 0; // interface definition
};

template <typename E>
struct Visitor<E>
{
    virtual void visit(E& ) = 0; // interface definition
};
```

Our topology must be now upfront defined, in terms of the elements (as with finite state-machine)

```
// Our visitation structure
class ListOfElements
```

```
{
    public:
        using element_type = std::variant<A<int>, A<std::uint8_t> /*, A<bool>, A<double>*/>;
        using element_types = std::vector<element_type>;

        constexpr ListOfElements(std::initializer_list<element_type> elements) noexcept: elements_{elements} {}

        template <typename Visitor>
        constexpr void accept(Visitor&&visitor) noexcept
        {
            for(auto& el :elements_) {
                std::visit([v =std::forward<Visitor>(visitor)](auto& e) mutable { v.visit(e); }, el);
            }
        }

   private:
       element_types elements_;
};
```

The other drawback using std::visit() is that the concreate Visitor must include all overloaded versions of *visit*() (for each element in Element-hierarchy)

```
// Concreate Visitor
struct IntegralVisitor : details::Visitor<A<int>, A<std::uint8_t>, /*, A<bool>, A<double>*/>
{
    void visit(A<int>&a) override {std::cout <<a;}
    void visit(A<std::uint8_t>&a) override {std::cout <<a;}
    // void visit(A<bool>&a) override {std::cout <<a;}
    // void visit(A<double>&a) override {std::cout <<a;}
};
```

The entire code: https://godbolt.org/z/4e1ToqoYY

There is even more convenient way to accomplish the same, using the well-known utility helper [2)] for having **in-place Visitor** that can be configured with lambda expressions

```
namespace details
{
    /**
     * For defining overload resolution that will be used to
     * construct in-place visitor, that will be applied on a given
     * variant set in visitor pattern implemented with std::visit() library call
     *
     * @tparam Fs The set of callable objects that define in-place visitor
     */
    template <typename...Fs>
    struct overload: Fs...
    {
        /**
         * C-tor
         *
         * @param ts Constructing each callable object Fs from the ts by copying or moving it
         */
         template <typename...Ts>
         overload(Ts&&...ts): Fs{std::forward<Ts>(ts)}...
         {}

         // Import all Fs call operators
         using Fs::operator()...;
    };

    // CTAD rule - needs to be manually defined, since the c-tor is templated
    template <typename...Ts>
    overload(Ts&&...) -> overload<std::remove_reference_t<Ts>...>;

}// namespace details
```

The syntax becomes cleaner (more readable)

```
class ListOfElements
{
    public:
        using element_type = std::variant<A<int>, A<std::uint8_t> /*, A<bool>, A<double>*/>;
        using element_types = std::vector<element_type>;

        constexpr ListOfElements(std::initializer_list<element_type> elements) noexcept: elements_{elements} {}

        template <typename Visitor>
        constexpr void accept(Visitor&& visitor) noexcept
        {
            for(auto& el :elements_) {
               std::visit(std::forward<Visitor>(visitor), el);
            }
```

```
        }

    private:
        element_types elements_;
};
```

This has also limitations - callable type (Fs) must be inheritable.
That is to say, it doesn't work with function pointers, or the final (sealed) callable types.

The seconds problem is that is not any more in line with Open-Closed principal, since Visitor-hierarchy is vanished, and therefore the inheritance as mechanism for extending the Visitor-hierarchy without direct modification of existing code

```
ListOfElements elements = {A<int>(328),A<std::uint8_t>(0xA2)};
elements.accept(details::overload(// Define (but also modify) visitor on the fly
                       [](A<int>&a) {std::cout <<"A<int> - "<<a;},
                       [](A<std::uint8_t>&a) {std::cout <<"A<std::uint8_t - "<<a;}));
```

The entire code: https://godbolt.org/z/cjf4zK1Wz

Links

1) **Fedor G. Pikus**, "*Hands-on design patterns with C++*": https://www.packtpub.com/product/hands-on-design-patterns-with-c
2) **Arne Mertz**, "*Build a variant visitor on the fly*": https://arne-mertz.de/2018/05/overload-build-a-variant-visitor-on-the-fly