Visitor Design Pattern

Represent an operation to be performed on the elements of an object structure.

A diagram of a visitor pattern

Description automatically generated

**RTTI**

struct A{  
 virtual void op{…}  
 *type = A;*  
}

struct B:public A{  
 void op() override {…}  
 *type = B;*  
}

İtalik alanlar gizli alanlar.

std::vector<A\*> vec;  
for (pA:vec){  
 pA 🡪op(); ------> ÇALIŞIR  
   
 //if the type is A, then do:  
 if (dynamic\_cast<B\*>(pA)); -------> Eğer A’nın tipi B’yse B\* döndür, B değilse null döndür.  
 Burada italik gizli alanlar ile RTTI yapılır.  
}

std::any\_cast<> 🡪 Bu da RTTI kullanır.

int \*a;  
float \*b = (float\*)a; ---> BÖYLE CAST YAPMIYORUZ!

3 cast var:

* static\_cast
* dynamic\_cast 🡪 cast or return nullptr
* interpret\_cast 🡪 ilişki olmadığını bilerek cast yapmak

**std::variant<…>**

using TValue = std::variant<std::string, float, Foo>;

Avantajı 🡪 compile time’da yapılması.

struct Foo{  
 …  
};

struct diyince “public:” dememiz gerekmiyor, o yüzden kolay olsun diye struct yazıyoruz.

typedef yerine using kullanıyoruz.

Bu ikisi aynı:

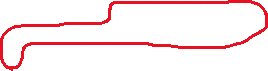
* typedef int MyInt;
* using MyInt = int;

using TValue = std::variant<Foo, Bar>;

struct Bar{  
 void process{};  
};

struct Foo{  
 void process{};  
};

void processItem(const TValue &value){  
 std::visit([]const auto &v){  
 v.process();  
 }, value);  
}



Kırmızı kısım lambda, value ise variant.

Variant üzerinde lambdayı çalıştır diyoruz. Foo’sa Foo’nun, Bar’sa Bar’ın processini çağırıyoruz.

Eskiden pointerlar ile yapardık.

std::vector<TValue> m\_vec;

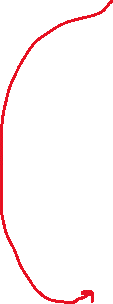
for(auto &item: m\_vec)  
 processItem(item);

std::variant olmasaydı:

struct Base{  
 virtual void process() = 0;  
};

struct Foo : public Base{  
 void process() override{  
 …  
 }  
};

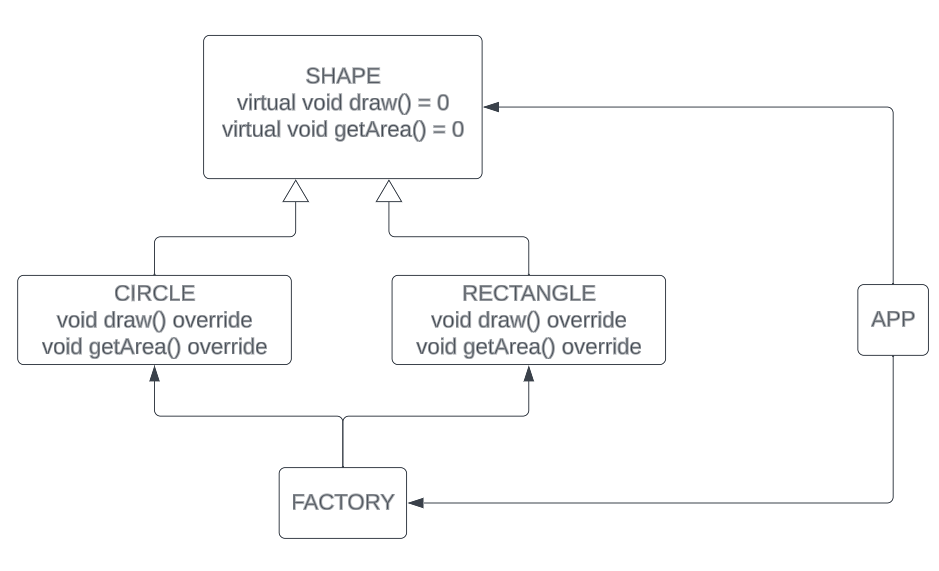
struct Bar : public Base{  
 void process() override{  
 …  
 }  
};



std::vector<Base\*> m\_vec;  
for (auto pBase:m\_vec)  
 pBase 🡪 process();

Yalnızca virtuallar üzerinden yapabilirim, overhead fazla.

Daha sonra yeni fonksiyon ekleyeceksek visitor dp kullanmak uygun olabilir.





Red part 🡪 Circle.so

Blue part 🡪 Rectangle.so

App itself is App.so

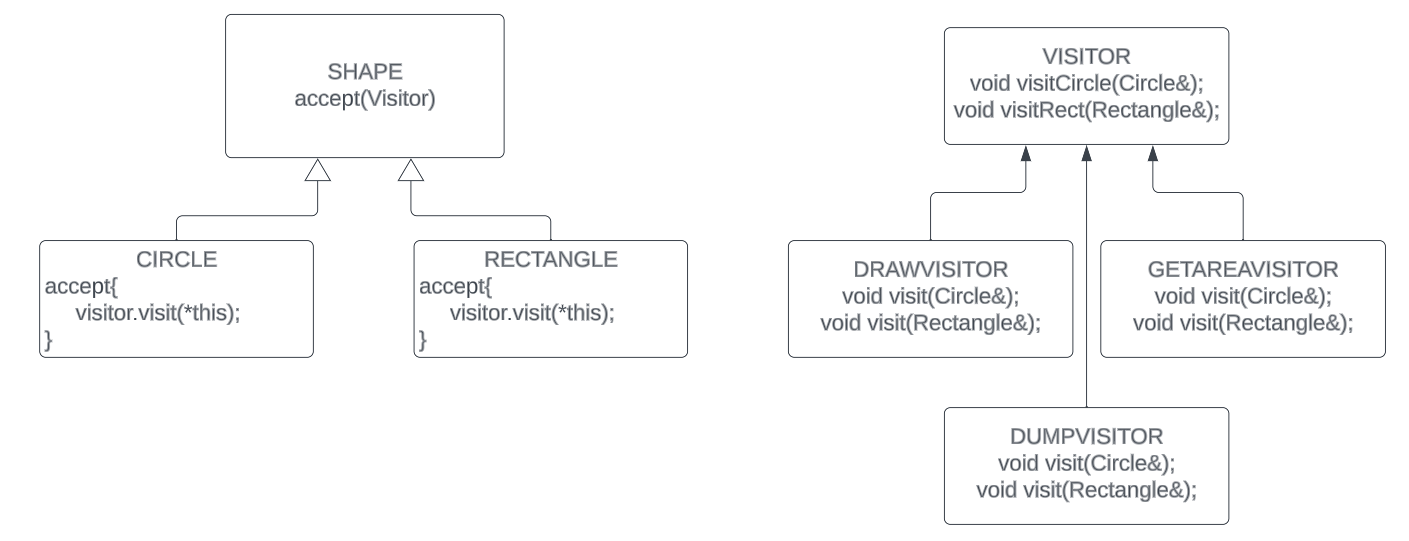
Factory itself is Factory.so

Client gets:

* App.so
* Factory.so
* Circle.so
* Rectangle.so
* ---
* Bunları verdim, link oldular ve çalıştılar.

Yeni tip eklemek için uygun

Yeni fonksiyon eklemek için uygun değil (Örneğin Shape’e dump() fonksiyonu eklemek.)



Bu sefer (visitor):

* Yeni tip eklemek için uygun değil.
* Yeni fonksiyon eklemek için uygun.