Structural Design Pattern

Hung Tran

Fpt software

October 31, 2021

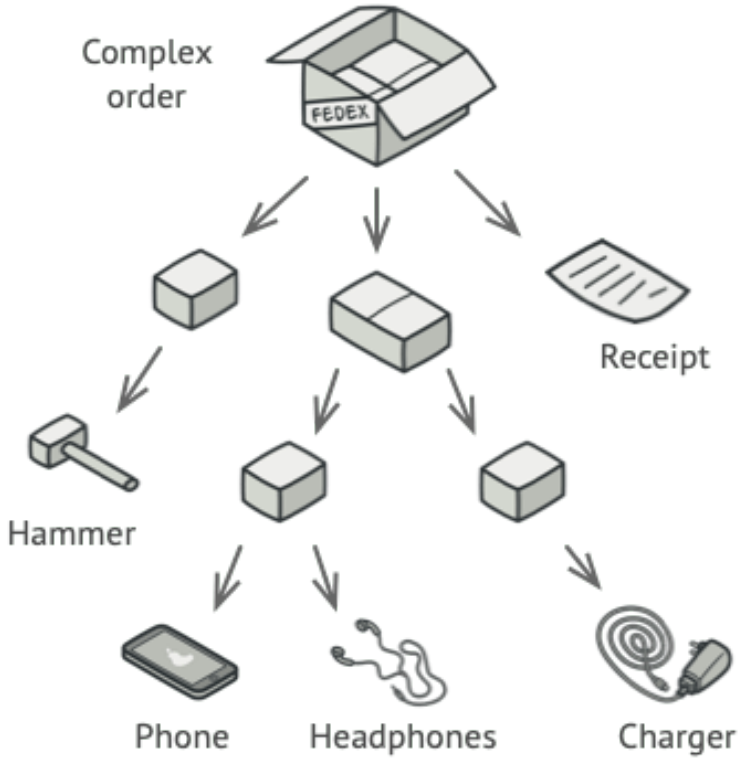
Outline

Structural Pattern Overview

How classes and objects are composed to form larger structure.

- **Adapter**: Convert the interface of a class into another interface.
- **Bridge**: Decouple an abstraction from its implementation.
- **Composite**: Compose objects into tree structure.
- **Decorator**: Attach additional responsibilities to an object dynamically.
- **Facade**: Provide a unified interface to a set of interfaces.
- **Flyweight**: Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy**: Provide a surrogate or placeholder for another object to control access to it.

Problem Statement



- Imagine that you have two types of objects: **Products** and **Boxes**
- A Box can contain several Products as well as a number of smaller Boxes.
- These little Boxes can also hold some Products or even smaller Boxes, and so on.

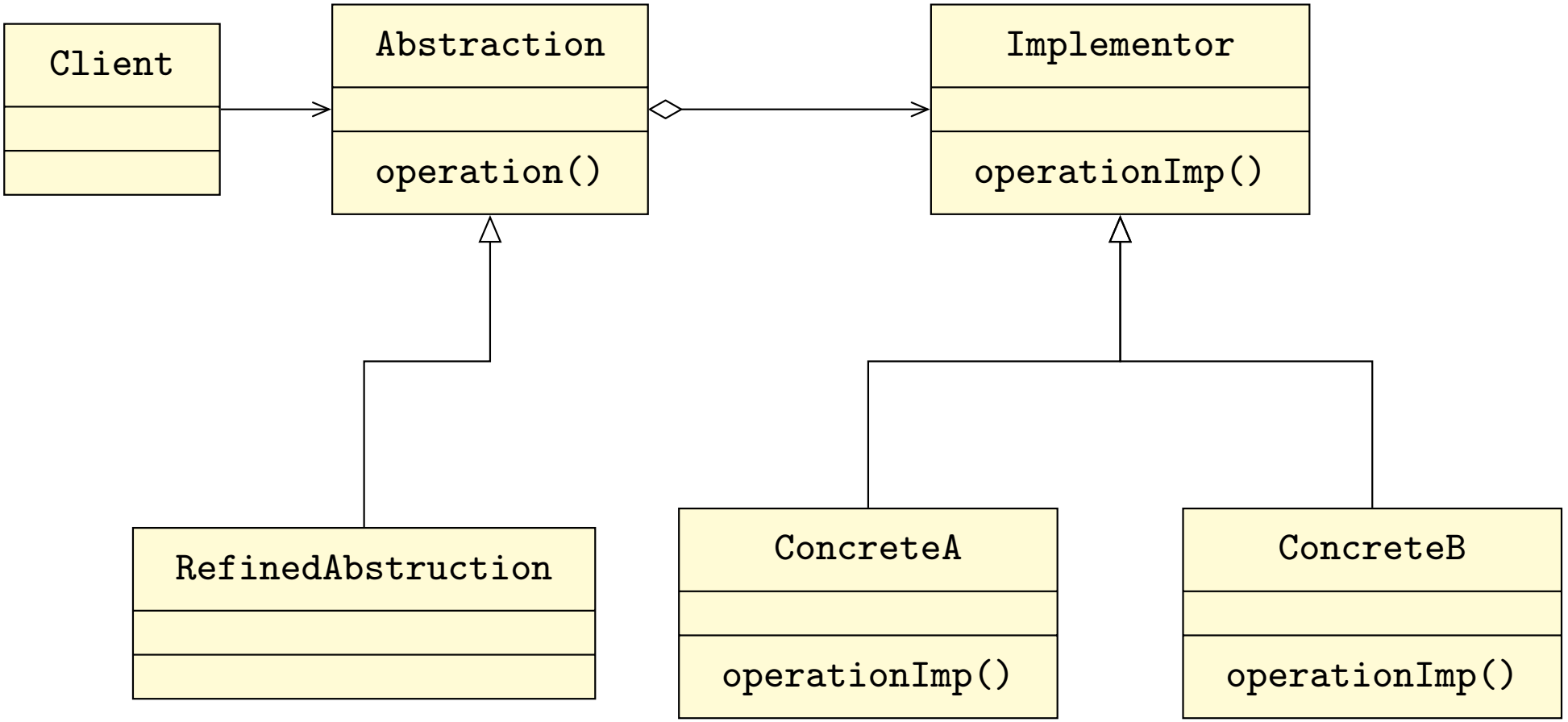
Problem Statement

- Create an ordering system that uses these classes
- Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes.
- How would you determine the total price of such an order?
- You could try the direct approach: unwrap all the boxes, go over all the products and then calculate the total.
- That would be doable in the real world; but in a program, it's not as simple as running a loop.
- You have to know the classes of Products and Boxes you're going through, the nesting level of the boxes and other nasty details beforehand.
- All of this makes the direct approach either too awkward or even impossible.

The Intent of Composite Design Pattern

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structure of Bridge Pattern: Object adapter



Pointer to Implementation (PIMPL)

- PIMPLE is the manifestation of the bridge design pattern albeit a slightly different one.
- PIMPL idiom is all about hiding the implementation details of a particular class by sticking it into separate implementation pointed by pointer just as the name suggests.

PIMPL implementation

person.h

```
1  #ifndef _PERSON_H_
2  #define _PERSON_H_
3
4  #include <string>
5  #include <memory>
6
7  struct Person {
8    class PersonImpl;
9    unique_ptr<PersonImpl> m_imp; //
       Bridge not necessaruly inner class,
        can vary
10   string m_name;
11
12   Person();
13   ~Person();
14
15   void greet();
16
17 private:
18   // secret data members or methods are
       in 'PersonImpl' not here
19   // as we are going to expose this
       class to client
20 }
21 #endif // _PERSON_H_
```

person.cpp

```
1  #include "person.h"
2
3  /* PIMPL implementation*/
4
5  struct Person::PersonImpl {
6    void greet(Person* p) {
7      std::cout << "Helllo" << p->name.
         c_str() << std::endl;
8    }
9  };
10
11 Person::Person() : m_impl(new PersonImpl
       ){
12 }
13
14 Person::~Person() {
15   delete m_impl;
16 }
17
18 void Person::greet() {
19   m_impl->greet(this);
20 }
```

Why would you want to do this PIMPL?

- Security purpose: a data member which contains critical information.
- Compilation time

Disadvantages of PIMPL?

- Run-time overhead as we have to dereference the pointer every time for access.
- Construction & destruction overhead of unique_ptrbecause it creates a memory in a heap
- We also have to bear some indirection if we want to access the data member of Person in PersonImpl like passing this pointer or so

Advantages

- Bridge Design Pattern provides flexibility to develop abstraction(i.e. interface) and the implementation independently. And the client/API-user code can access only the abstraction part without being concerned about the Implementation part.
- It preserves the Open-Closed Principle, in other words, improves extensibility as client/API-user code relies on abstraction only so implementation can modify or augmented any time.
- By using the Bridge Design Pattern in the form of PIMPL. We can hide the implementation details from the client as we did in PIMPL idiom example above.
- The Bridge Design Pattern is an application of the old advice, "prefer composition over inheritance" but in a smarter way. It comes handy when you must subclass different times in ways that are orthogonal with one another(say 2×2 problem discuss earlier).
- A compile-time binding between an abstraction and its

Thank You!