Structural Design Pattern

Hung Tran

Fpt software

October 30, 2021

Outline

1. Structural Pattern Overview

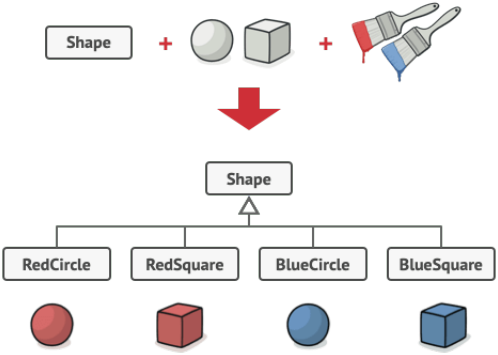
2. Bridge design pattern

Structural Pattern Overview

How classes and objects are composed to form larger structure.

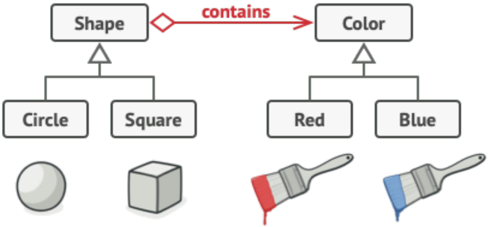
- **Adapter**: Convert the interface of a class into another interface.
- **Bridge**: Decouple an abstraction from its implementation.
- **Composite**: Compose objects into tree structure.
- **Decorator**: Attach additional responsibilities to an object dynamically.
- **Facade**: Provide a unified interface 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



- You have a geometric Shape class with a pair of subclasses: Circle and Square.
- You want to extend this class hierarchy to incorporate colors.
- Adding new shape types and colors to the hierarchy will grow it exponentially.
- The total classes by combination?

Problem Statement

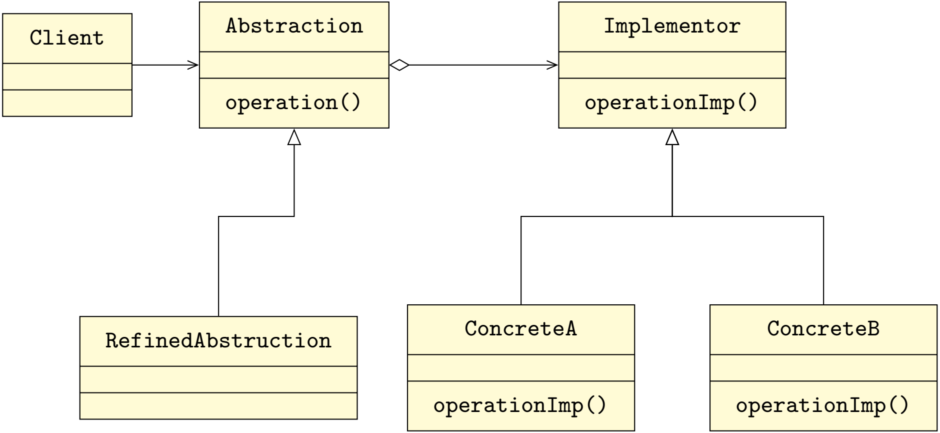


- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.
- Adding new colors won't require changing the shape hierarchy, and vice versa.

The Intent of Decorator Design Pattern

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

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

```cpp
#ifndef _PERSON_H_
#define _PERSON_H_

#include <string>
#include <memory>

struct Person {
  class PersonImpl;
  unique_ptr<PersonImpl> m_imp; // Bridge not necessaruly inner class, can vary
  string m_name;

  Person();
  ~Person();

  void greet();

private:
  // secret data members or methods are in ‘PersonImpl‘ not here
  // as we are going to expose this class to client
}
#endif // _PERSON_H_
```

person.cpp

```cpp
#include "person.h"

/* PIMPL implementation*/

struct Person::PersonImpl {
  void greet(Person* p) {
    std::cout << "Helllo" << p->name.c_str() << std::endl;
  }
};

Person::Person() : m_impl(new PersonImpl){
}

Person::~Person() {
  delete m_impl;
}

void Person::greet() {
  m_impl->greet(this);
}
```

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!