Structural Design Pattern

Hung Tran

Fpt software

November 2, 2021

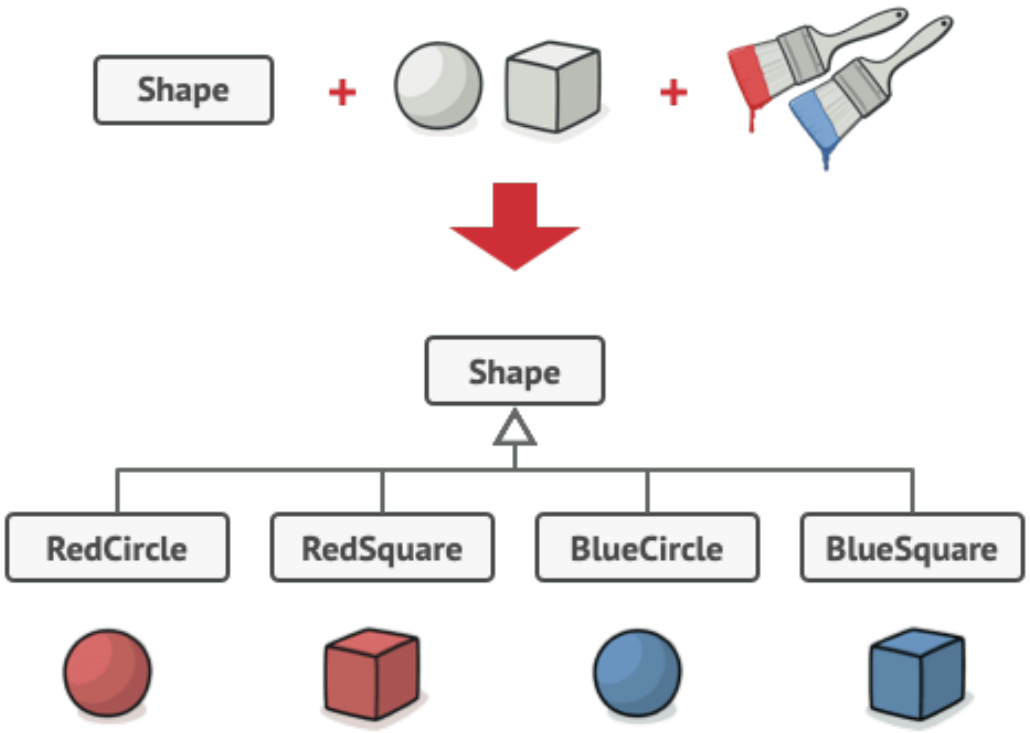
Outline

Structural Pattern Overview

How classes and objects are composed to form larger structure.

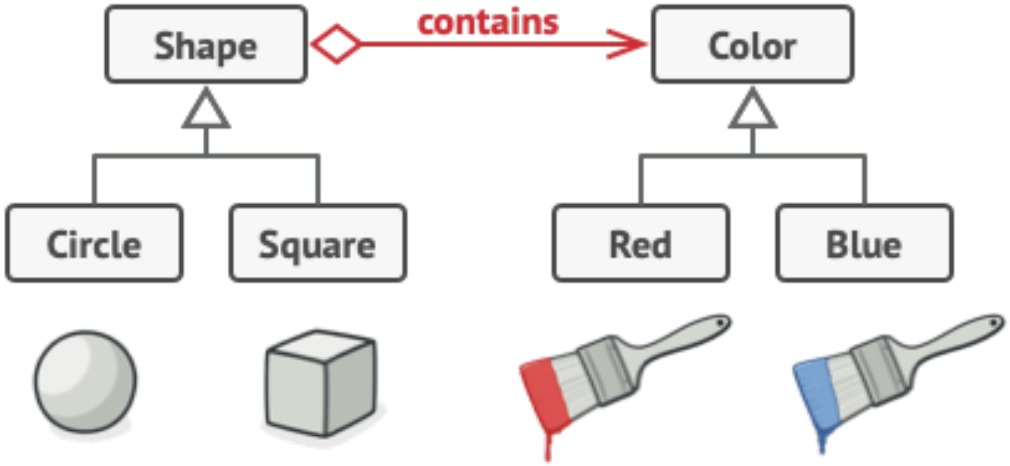
- **Adapter**: Convert the interface of a class into another interface.
- **Bridge**: Decouple an abstraction from its implementation.
- **Composite**: Compose objects into tree structure.
- **Decorator**: Attach additional responsibilities to an object dynamically.
- **Facade**: Provide a unified interface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



- 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

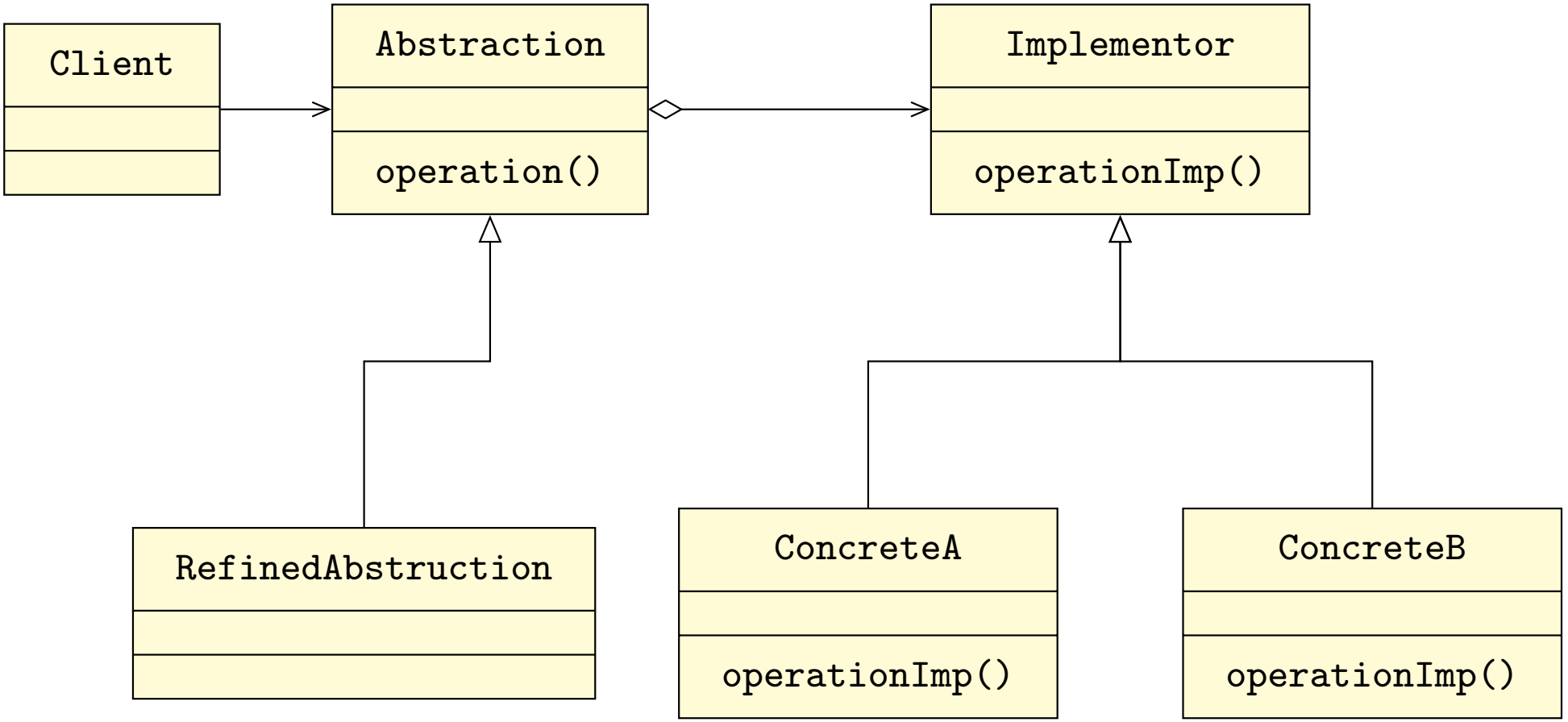


- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.
- Adding new colors won't require changing the shape hierarchy, and vice versa.

The Intent of Bridge Design Pattern

Decouple an abstraction from its implementation so that the two can vary independently.

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

Applicability

- Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers).
- The bigger a class becomes, the harder it is to figure out how it works, and the longer it takes to make a change. The changes made to one of the variations of functionality may require making changes across the whole class, which often results in making errors or not addressing some critical side effects.
- The Bridge pattern lets you split the monolithic class into several class hierarchies. After this, you can change the classes in each hierarchy independently of the classes in the others. This approach simplifies code maintenance and minimizes the risk of breaking existing code.

Applicability

- Use the pattern when you need to extend a class in several orthogonal (independent) dimensions.
- The Bridge suggests that you extract a separate class hierarchy for each of the dimensions. The original class delegates the related work to the objects belonging to those hierarchies instead of doing everything on its own.
- Use the Bridge if you need to be able to switch implementations at runtime.
- Although it's optional, the Bridge pattern lets you replace the implementation object inside the abstraction. It's as easy as assigning a new value to a field.

How to Implement

- Identify the orthogonal dimensions in your classes. These independent concepts could be: abstraction/platform, domain/infrastructure, front-end/back-end, or interface/implementation.
- See what operations the client needs and define them in the base abstraction class.
- Determine the operations available on all platforms. Declare the ones that the abstraction needs in the general implementation interface.
- For all platforms in your domain create concrete implementation classes, but make sure they all follow the implementation interface.

How to Implement

- Inside the abstraction class, add a reference field for the implementation type. The abstraction delegates most of the work to the implementation object that's referenced in that field.
- If you have several variants of high-level logic, create refined abstractions for each variant by extending the base abstraction class.
- The client code should pass an implementation object to the abstraction's constructor to associate one with the other. After that, the client can forget about the implementation and work only with the abstraction object.

Pros and Cons

- You can create platform-independent classes and apps.
- The client code works with high-level abstractions. It isn't exposed to the platform details.
- Open/Closed Principle. You can introduce new abstractions and implementations independently from each other.
- Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.
- You might make the code more complicated by applying the pattern to a highly cohesive class.

Relations with Other Patterns

- Bridge is usually designed up-front, letting you develop parts of an application independently of each other. On the other hand, Adapter is commonly used with an existing app to make some otherwise-incompatible classes work together nicely.
- Bridge, State, Strategy (and to some degree Adapter) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.
- You can use Abstract Factory along with Bridge. This pairing is useful when some abstractions defined by Bridge can only work with specific implementations. In this case, Abstract Factory can encapsulate these relations and hide the complexity from the client code.
- You can combine Builder with Bridge: the director class plays the role of the abstraction, while different builders act as implementations.

Thank You!