Structural Design Pattern

Hung Tran

Fpt software

November 3, 2021

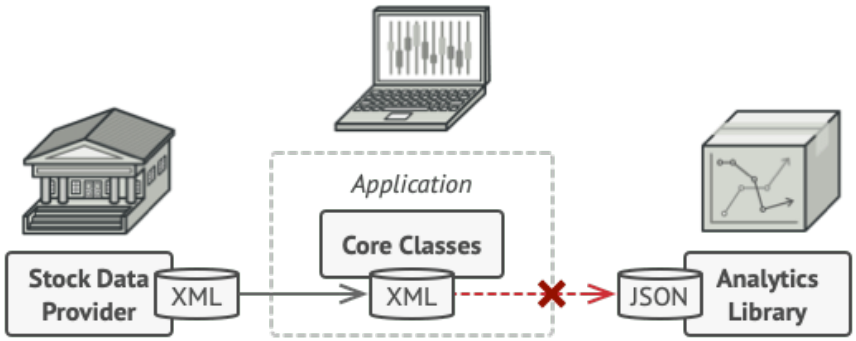
Outline

Structural Pattern Overview

How classes and objects are composed fo form larger structure.

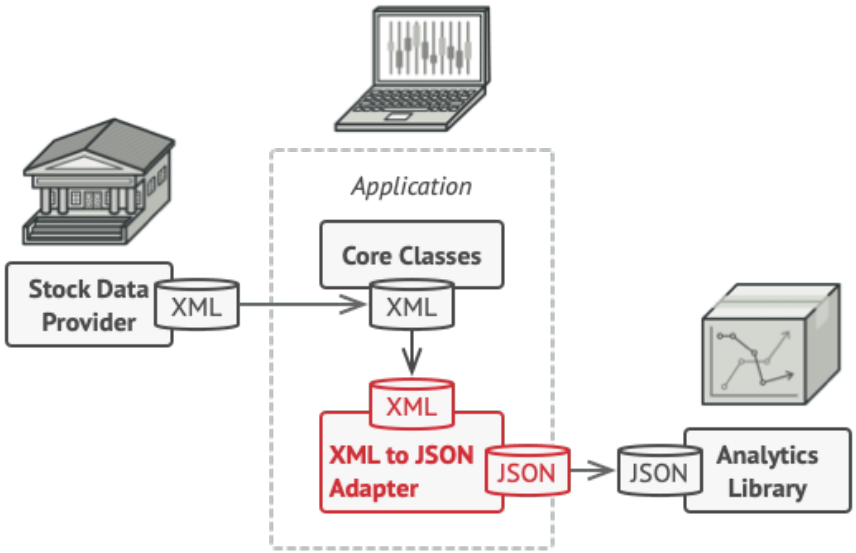
- **Adapter**: Convert the interface of a class into another interface.
- **Bridge**: Decouple an abstraction from its implementation.
- **Composite**: Compose objects into tree structure.
- **Decorator**: Attach additional responsibilities to an object dynamically.
- **Facçade**: Provide a unified interface 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're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.
- You could change the library to work with XML. However, this

Solution

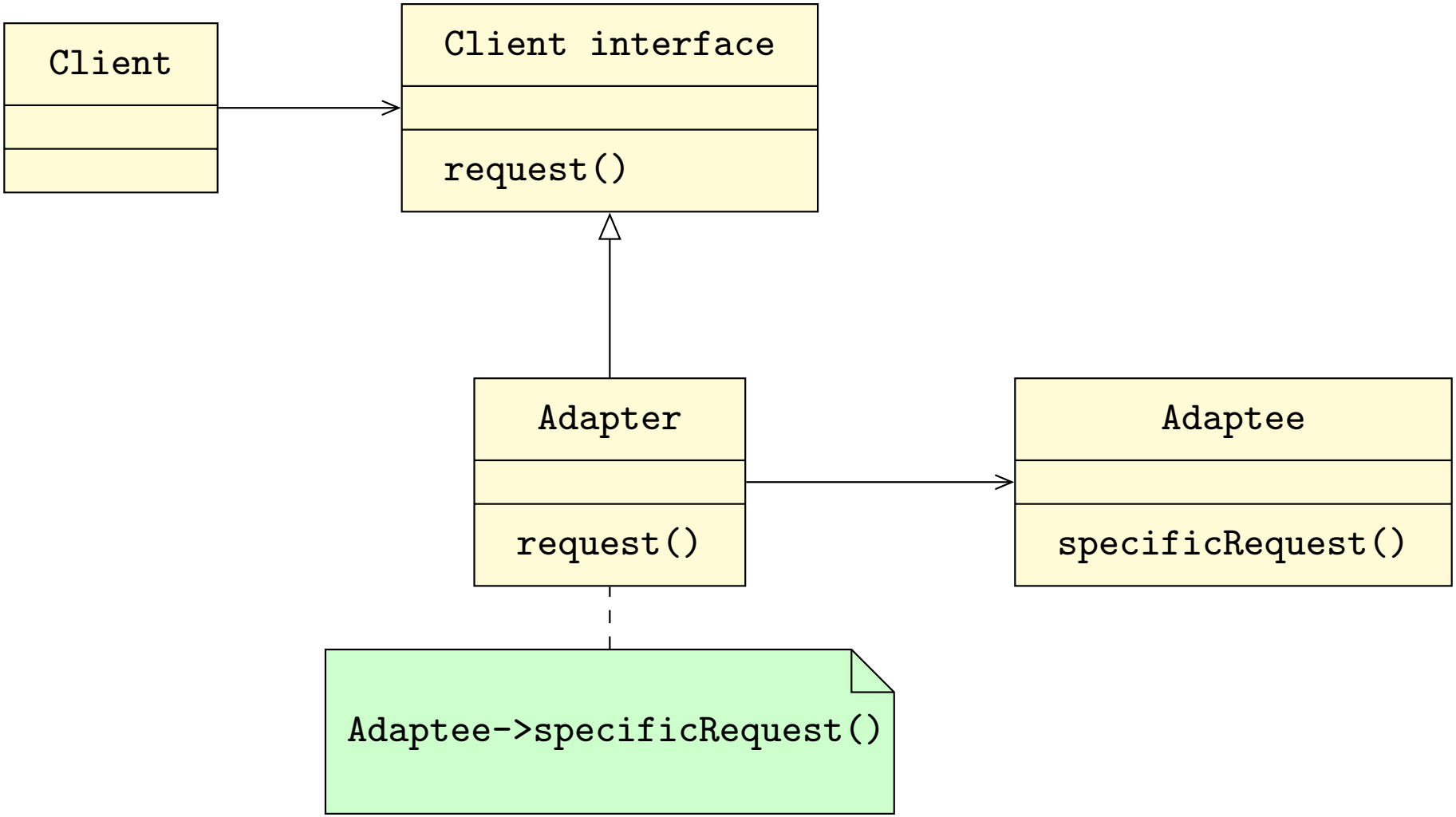


- You can create an adapter. This is a special object that converts the interface of one object so that another object can understand it.
- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter.
- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:

The Intent of Adapter Design Pattern

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Structure of Adapter Pattern: Object adapter



Basic implementation: Rectangle class

rectangle.h

```
1  #ifndef _RECTANGLE_H_
2  #define _RECTANGLE_H_
3  class Rectangle {
4    int a;
5    int b;
6  public:
7    Rectangle();
8    Rectangle(int a, int b);
9    virtual int width() const;
10   virtual int height() const;
11   virtual int area() const;
12 };
13
14 #endif // _RECTANGLE_H_
```

rectangle.cpp

```
1  #include "rectangle.h"
2
3  Rectangle::Rectangle() : a{0}, b{0} {
4  }
5
6  Rectangle::Rectangle(int a, int b) : a{a
       }, b{b} {
7  }
8
9  int Rectangle::width() const {
10   return a;
11 }
12
13 int Rectangle::height() const {
14   return b;
15 }
16
17 int Rectangle::area() const {
18   return a*b;
19 }
```

Basic implementation: Square class

square.h

```
1  #ifndef _SQUARE_H_
2  #define _SQUARE_H_
3  class Square {
4    int a;
5  public:
6    Square();
7    Square(int a);
8    int getEdge() const;
9    int area() const;
10 };
11 #endif // SQUARE_H_
```

square.cpp

```
1  #include "square.h"
2
3  Square::Square() : a{0} {
4  }
5
6  Square::Square(int a) : a{a} {
7  }
8
9  int Square::getEdge() const {
10   return a;
11 }
12
13 int Square::area() const{
14   return a*a;
15 }
```

Basic implementation: Adapter class

adapter.h

```
1  #ifndef _ADAPTER_H_
2  #define _ADAPTER_H_
3  #include "rectangle.h"
4  #include "square.h"
5  class Adapter : public Rectangle{
6    Square& s;
7  public:
8    Adapter(Square& s);
9    int width() const override;
10   int height() const override;
11   int area() const override;
12 };
13 #endif // _ADAPTER_H_
```

adapter.cpp

```
1  #include "adapter.h"
2
3  Adapter::Adapter(Square& s) : s{s}{
4  }
5
6  int Adapter::width() const {
7    return s.getEdge();
8  }
9
10 int Adapter::height() const {
11   return s.getEdge();
12 }
13
14 int Adapter::area() const {
15   return s.area();
16 }
```

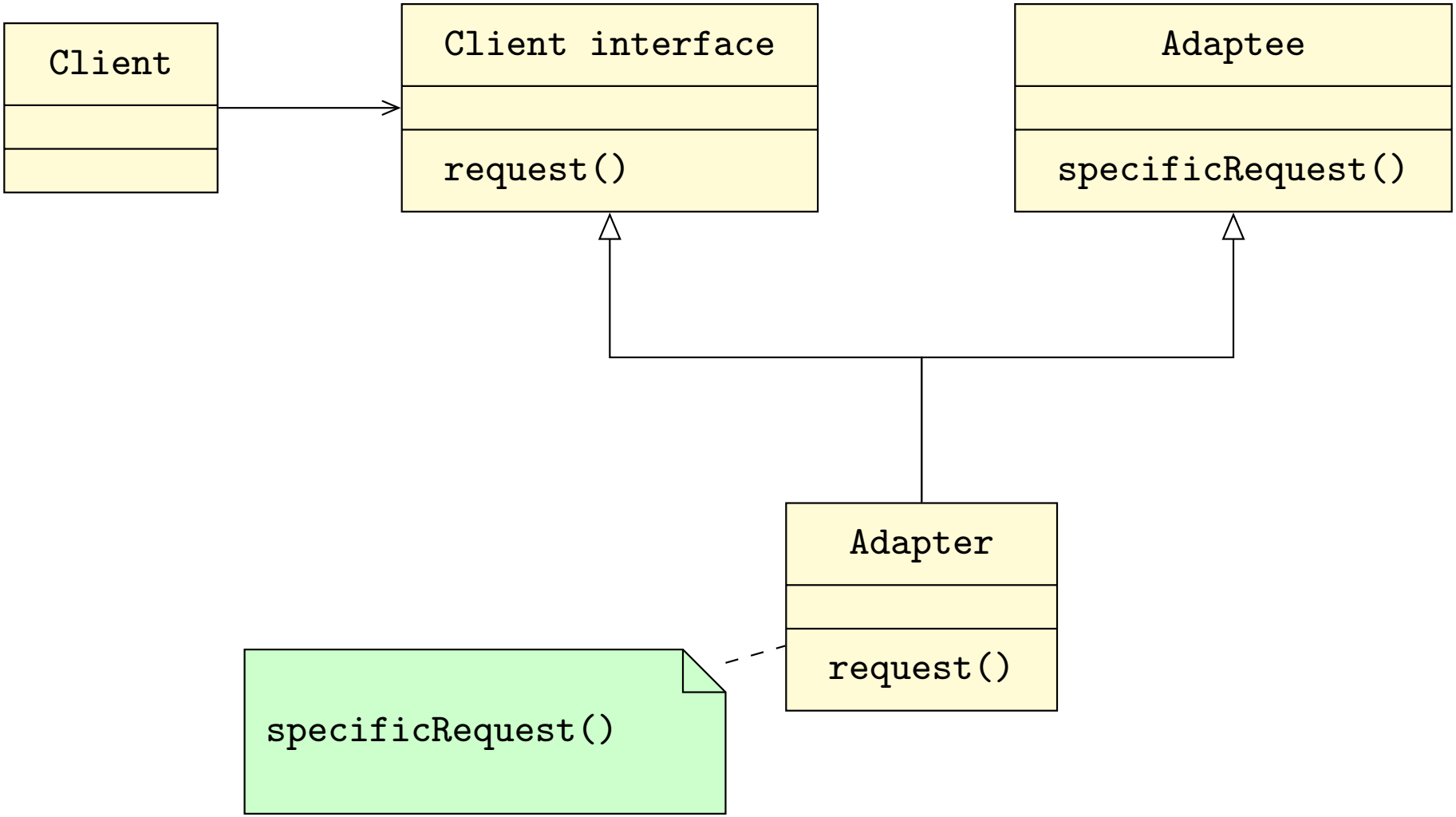
Basic implementation: client code

main.cpp

```
1  #include "adapter.h"
2  #include <iostream>
3
4  // client code
5  void doSomething(Rectangle& r) {
6    int w = r.width();
7    int h = r.height();
8
9    std::cout << "width: " << w << "\
       nheight: " << h << std::endl;
10 }
11
12 int main() {
13   Square s(2);
14   Rectangle r(2,3);
15   doSomething(r);
16   //doSomething(s);
17   Adapter a(s);
18   doSomething(a);
19   return 0;
20 }
```

- Object adapter

Structure of Adapter Pattern: Class adapter



Basic implementation: Rectangle class

rectangle.h

```
1  #ifndef _RECTANGLE_H_
2  #define _RECTANGLE_H_
3  class Rectangle {
4    int a;
5    int b;
6  public:
7    Rectangle();
8    Rectangle(int a, int b);
9    virtual int width() const;
10   virtual int height() const;
11   virtual int area() const;
12 };
13
14 #endif // _RECTANGLE_H_
```

rectangle.cpp

```
1  #include "rectangle.h"
2
3  Rectangle::Rectangle() : a{0}, b{0} {
4  }
5
6  Rectangle::Rectangle(int a, int b) : a{a
       }, b{b} {
7  }
8
9  int Rectangle::width() const {
10   return a;
11 }
12
13 int Rectangle::height() const {
14   return b;
15 }
16
17 int Rectangle::area() const {
18   return a*b;
19 }
```

Basic implementation: Square class

square.h

```
1  #ifndef _SQUARE_H_
2  #define _SQUARE_H_
3  class Square {
4    int a;
5  public:
6    Square();
7    Square(int a);
8    int getEdge() const;
9    int area() const;
10 };
11 #endif // SQUARE_H_
```

square.cpp

```
1  #include "square.h"
2
3  Square::Square() : a{0} {
4  }
5
6  Square::Square(int a) : a{a} {
7  }
8
9  int Square::getEdge() const {
10   return a;
11 }
12
13 int Square::area() const{
14   return a*a;
15 }
```

Basic implementation: Adapter class

adapter.h

```
1  #ifndef _ADAPTER_H_
2  #define _ADAPTER_H_
3  #include "rectangle.h"
4  #include "square.h"
5  class Adapter : public Rectangle,
       private Square{
6  public:
7    Adapter(Square& s);
8    int width() const override;
9    int height() const override;
10   int area() const override;
11 };
12 #endif // _ADAPTER_H_
```

adapter.cpp

```
1  #include "adapter.h"
2
3  Adapter::Adapter(Square& s) : Square(s)
       {
4  }
5
6  int Adapter::width() const {
7    return this->getEdge();
8  }
9
10 int Adapter::height() const {
11   return this->getEdge();
12 }
13
14 int Adapter::area() const {
15   return this->area();
16 }
```

Basic implementation: client code

main.cpp

```
#include "adapter.h"
#include <iostream>

// client code
void doSomething(Rectangle* r) {
  int w = r->width();
  int h = r->height();

  std::cout << "width: " << w << "\
      nheight: " << h << std::endl;
}

int main() {
  Square s(2);
  Rectangle* r = new Rectangle(2,3);
  doSomething(r);
  //doSomething(s);
  Rectangle* a = new Adapter(s);
  doSomething(a);
  return 0;
}
```

- This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time.
- The Class Adapter doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

Applicability

- Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.
- The Adapter pattern lets you create a middle-layer class that serves as a translator between your code and a legacy class, a 3rd-party class or any other class with a weird interface.
- Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.
- You could extend each subclass and put the missing functionality into new child classes. However, you'll need to duplicate the code across all of these new classes, which smells really bad.

How to Implement

- Make sure that you have at least two classes with incompatible interfaces:
  - A useful service class, which you can't change (often 3rd-party, legacy or with lots of existing dependencies).
  - One or several client classes that would benefit from using the service class.
- Declare the client interface and describe how clients communicate with the service.
- Create the adapter class and make it follow the client interface. Leave all the methods empty for now.

How to Implement

- Add a field to the adapter class to store a reference to the service object. The common practice is to initialize this field via the constructor, but sometimes it's more convenient to pass it to the adapter when calling its methods.
- One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object, handling only the interface or data format conversion.
- Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.

Pros and Cons

- Single Responsibility Principle. You can separate the interface or data conversion code from the primary business logic of the program.
- pen/Closed Principle. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.
- The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

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.
- Adapter changes the interface of an existing object, while Decorator enhances an object without changing its interface. In addition, Decorator supports recursive composition, which isn't possible when you use Adapter.
- Adapter provides a different interface to the wrapped object, Proxy provides it with the same interface, and Decorator provides it with an enhanced interface.

Relations with Other Patterns

- Facade defines a new interface for existing objects, whereas Adapter tries to make the existing interface usable. Adapter usually wraps just one object, while Facade works with an entire subsystem of objects.
- 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.

Thank You!