Structural Design Pattern

Hung Tran

Fpt software

November 16, 2021

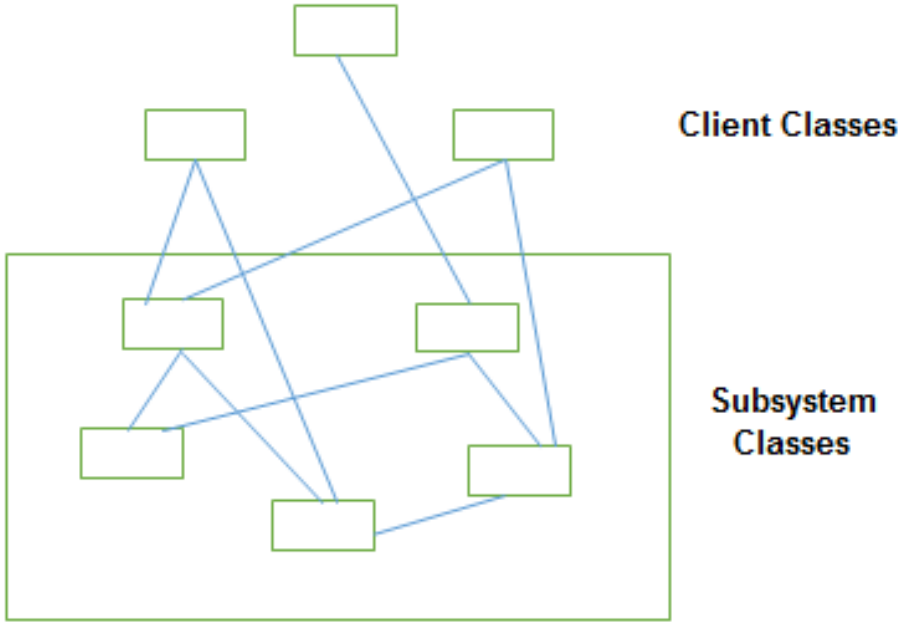
Outline

Structural Pattern Overview

How classes and objects are composed fo form larger structure.

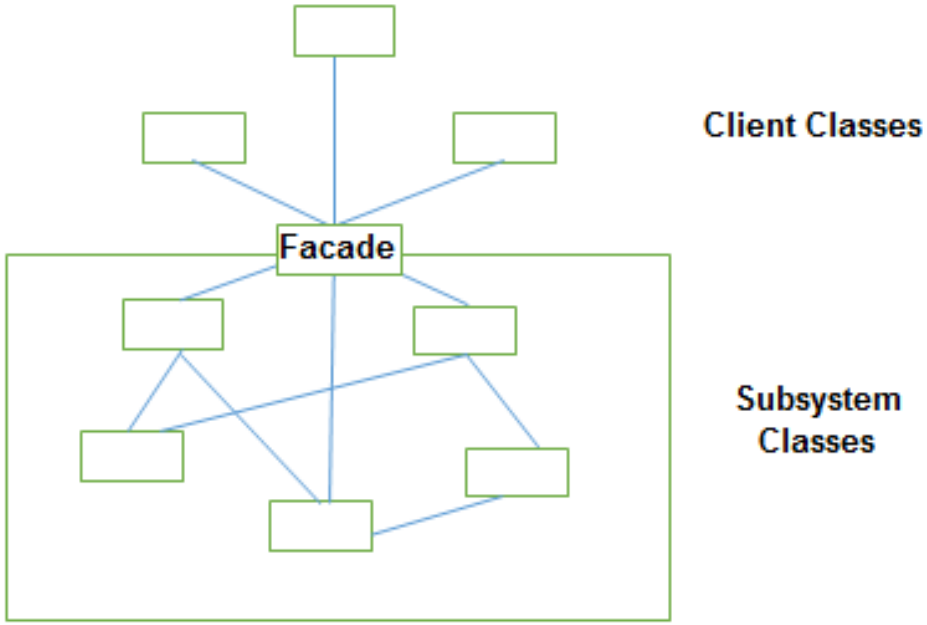
- **Adapter**: Convert the interface of a class into another interface.
- **Bridge**: Decouple an abstraction from its implementation.
- **Composite**: Compose objects into tree structure.
- **Decorator**: Attach additional responsibilities to an object dynamically.
- **Façade**: Provide a unified interface 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 must make your code work with a broad set of objects that belong to a sophisticated library or framework.
- You'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.
- As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to **comprehend and maintain**.

Solution: Flyweight Pattern

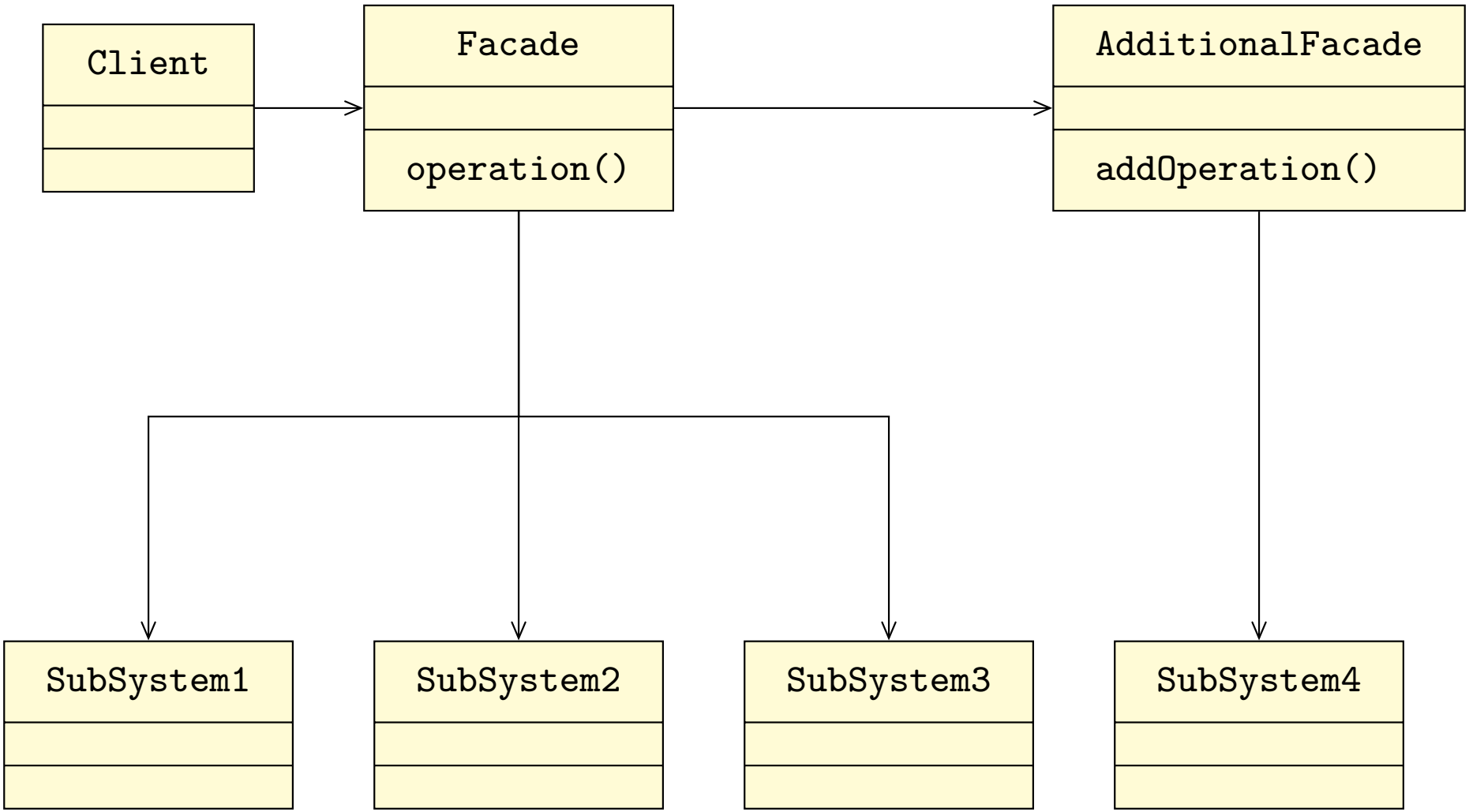


- A facade is a class that provides a simple interface to a complex subsystem.
- Provide limited functionality in comparison to working with the subsystem directly
- Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

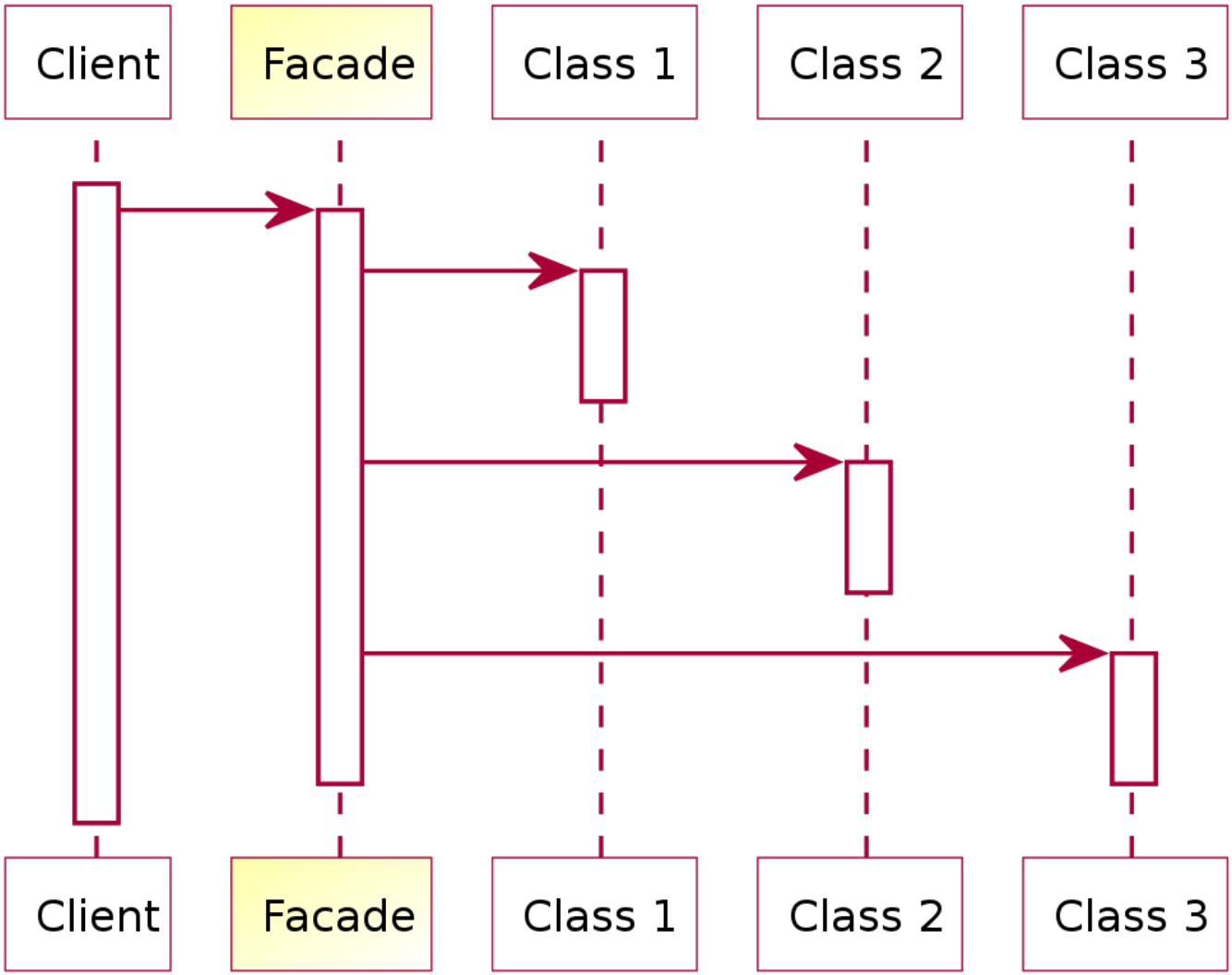
The Intent of Facade Design Pattern

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Structure of Facade Pattern: Object adapter



Sequence Diagram



Sample sequence diagram

Basic implementation: cpu class

cpu.h

```
1  #ifndef _CPU_H_
2  #define _CPU_H_
3  #include <string>
4
5  class CPU {
6  public:
7    void freeze();
8    void jump(long position);
9    void execute();
10 };
11 #endif // _CPU_H_
```

cpu.cpp

```
1  #include "cpu.h"
2  #include <iostream>
3
4  void CPU::freeze() {
5    std::cout << "call freeze function\n";
6  }
7
8  void CPU::jump(long position) {
9    std::cout << "call jump function\n";
10 }
11
12 void CPU::execute() {
13   std::cout << "call execute function\n"
         ;
14 }
```

Basic implementation: memory class

memory.h

```
1  #ifndef _MEMORY_H_
2  #define _MEMORY_H_
3  #include <string>
4
5  class Memory {
6  public:
7    void load(long position, std::string
       data);
8  };
9  #endif // _MEMORY_H_
```

memory.cpp

```
1  #include "memory.h"
2  #include <iostream>
3
4  void Memory::load(long position, std::
       string data) {
5    std::cout << "load method called\n";
6  }
```

Basic implementation: hard driver

hardDriver.h

```
1  #ifndef _HARD_DRIVER_H_
2  #define _HARD_DRIVER_H_
3
4  class HardDriver {
5  public:
6    void read(long lba, int size);
7  };
8  #endif // _HARD_DRIVER_H_
```

hardDriver.cpp

```
1  #include "hardDriver.h"
2  #include <iostream>
3
4  void HardDriver::read(long lba, int size
       ) {
5    std::cout << "read method called\n";
6  }
```

Basic implementation: computer facade

computerFacade.h

```
#ifndef _COMPUTER_FACADE_H_
#define _COMPUTER_FACADE_H_
#include "cpu.h"
#include "memory.h"
#include "hardDriver.h"
class ComputerFacade {
public:
 ComputerFacade();
 ~ComputerFacade();
 void start();
private:
  CPU* _cpu;
  Memory* _memory;
  HardDriver* _hd;
};
#endif // _COMPUTER_FACADE_H_
```

computerFacade.cpp

```
#include "computerFacade.h"

ComputerFacade::ComputerFacade() {
  _cpu = new CPU();
  _hd = new HardDriver();
  _memory = new Memory();
}

ComputerFacade::~ComputerFacade() {
  delete _cpu;
  delete _hd;
  delete _memory;
}

void ComputerFacade::start() {
  _cpu->freeze();
  _memory->load(1, "string");
  _cpu->execute();
}
```

Applicability

- Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.
- Use the Facade when you want to structure a subsystem into layers.

How to Implement

- Check whether it's possible to provide a simpler interface than what an existing subsystem already provides.
- Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem. The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.
- To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade. Now the client code is protected from any changes in the subsystem code.
- If the facade becomes too big, consider extracting part of its behavior to a new, refined facade class.

Pros and Cons

- You can isolate your code from the complexity of a subsystem.
- A facade can become a god object coupled to all classes of an app.

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.
- Abstract Factory can serve as an alternative to Facade when you only want to hide the way the subsystem objects are created from the client code.
- Flyweight shows how to make lots of little objects, whereas Facade shows how to make a single object that represents an entire subsystem.
- Facade and Mediator have similar jobs: they try to organize collaboration between lots of tightly coupled classes.
- A Facade class can often be transformed into a Singleton since a single facade object is sufficient in most cases.

Thank You!