Report on

Design Patterns Project

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Sreejesh Saya	**PES1201800293**
Gaurika Poplai	**PES1201800374**

Under the guidance of

Prof. N.S. Kumar

PES University, Bengaluru

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Pipeline

Intent

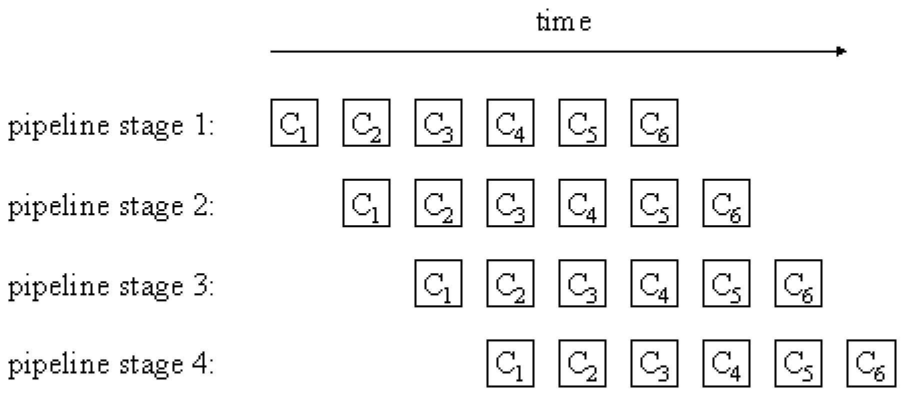
Allows processing of data in a series of independent stages by giving an initial input and passing the processed output to the stage in succession.

Motivation

The basic idea of this pattern is much like the idea of an assembly line: To perform a sequence of essentially identical calculations, each of which can be broken down into the same sequence of steps, we set up a "pipeline", one stage for each step, with all stages potentially executing concurrently. Each of the sequence of calculations is performed by having the first stage of the pipeline perform the first step, and then the second stage the second step, and so on. As each stage completes a step of a calculation, it passes the calculation-in-progress to the next stage and begins work on the next calculation.

This may be easiest to understand by thinking in terms of the assembly-line analogy: For example, suppose the goal is to manufacture a number of cars, where the manufacture of each car can be separated into a sequence of smaller operations (e.g., installing a windshield). Then we can set up an assembly line (pipeline), with each operation assigned to a different worker. As the car-to-be moves down the assembly line, it is built up by performing the sequence of operations; each worker, however, performs the same operation over and over on a succession of cars.

Returning to a more abstract view, if we call the calculations to be performed C_1, C_2, and so forth, then we can describe operation of a Pipeline processing program thus: Initially, the first stage of the pipeline is performing the first operation of C_1. When that completes, the second stage of the pipeline performs the second operation on C_1; simultaneously, the first stage of the pipeline performs the first stage of C_2. When both complete, the third stage of the pipeline performs the third operation on C_1, the second stage performs the second operation on C_2, and the first stage performs the first operation on C_3. The following figure illustrates how this works for a pipeline consisting of four stages.



time
pipeline stage 1:
C_1 C_2 C_3 C_4 C_5 C_6
pipeline stage 2:
C_1 C_2 C_3 C_4 C_5 C_6
pipeline stage 3:
C_1 C_2 C_3 C_4 C_5 C_6
pipeline stage 4:
C_1 C_2 C_3 C_4 C_5 C_6

This is also a huge win for testability and single responsibility — what this means is that each piece of code only does one or two things that it was designed to do, then passes the torch to the next class.

Applicability

The Pipeline pattern is useful when:

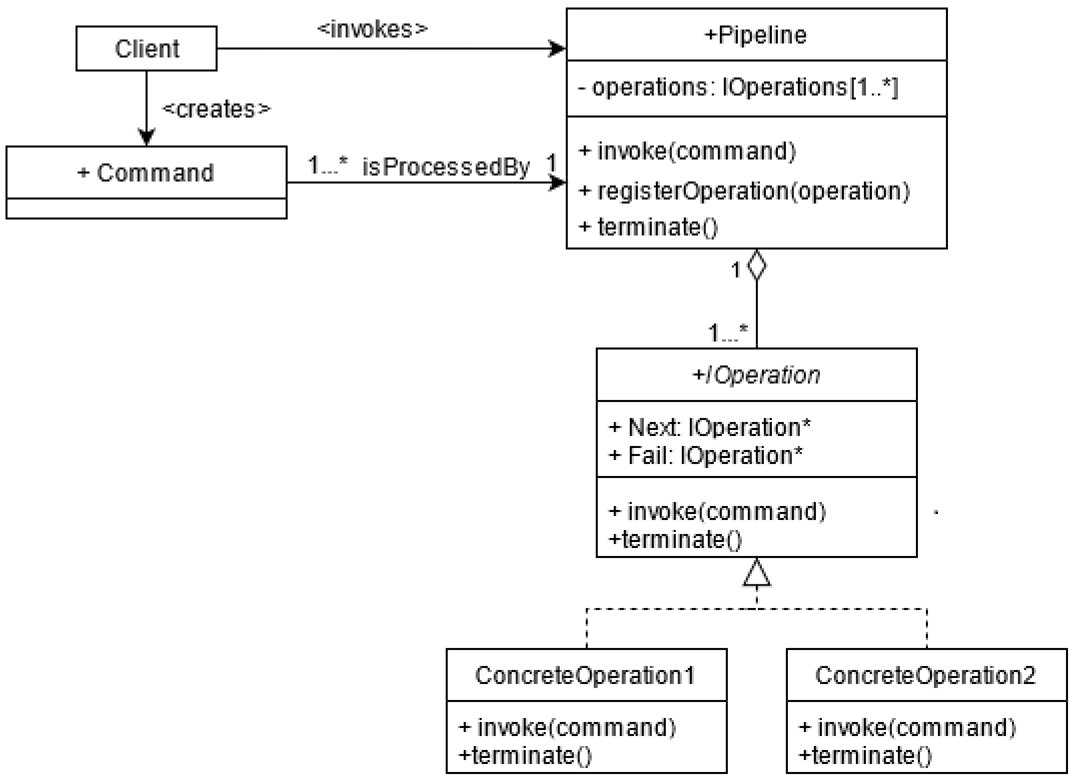
- We have individual stages that yield intermediate output that can be passed to the next stage.
- topology of application must be modified at runtime

This pattern is effective when:

1. The number of calculations (input) is large compared to the number of stages.
2. A processor can be dedicated to each stage of the pipeline.

Structure

Pipeline pattern is similar to CoR pattern, however instead of each *Concrete Operation* having a reference to the Successor in the chain, There is a IOperation manager that does this. The flexibility of the Pipeline pattern comes from the fact that at any time, a new IOperation can be injected into the pipeline through the Pipeline.



Participants

- **Command**: Object to be processed. An order in our example
- **Pipeline**: Central class of the pattern. Define the following attribute and method (Pipeline class)
    - *operations*: list (ordered) of **IOperations**
    - *invoke(**Command** object)*: execute the processing pipeline on the object.
- **IOperation**: an interface defining the *invoke(**Command** object)* method (Processor class)
- **ConcreteOperation:** defines the functionalities of the operations to be performed. Separate concrete operation for each operation.

Collaborations

- The pipeline is created by registering operations (ConcreteOperation).
- Next of an operation class points to the next registered operation. Incase of Fail, the command is handled by the circuit breaker.
- The pipeline processes the orders by calling only the first IOperation. The operation processes the command and feeds its output to the next ConcreteOperation in line.
- The client creates orders and invokes the pipeline after passing the orders.

Consequences

The pipeline pattern has benefits:

- Add readability to complex sequences of operations by providing a fluent interface.

- Improve code testability since stages will most likely perform one task, complying to the Single Responsibility Principle (SRP)

- Allows high flexibility, stages don’t depend on each other, they can be replaced, added, deleted. The entire topology of a pipeline can easily be modified to suit needs.

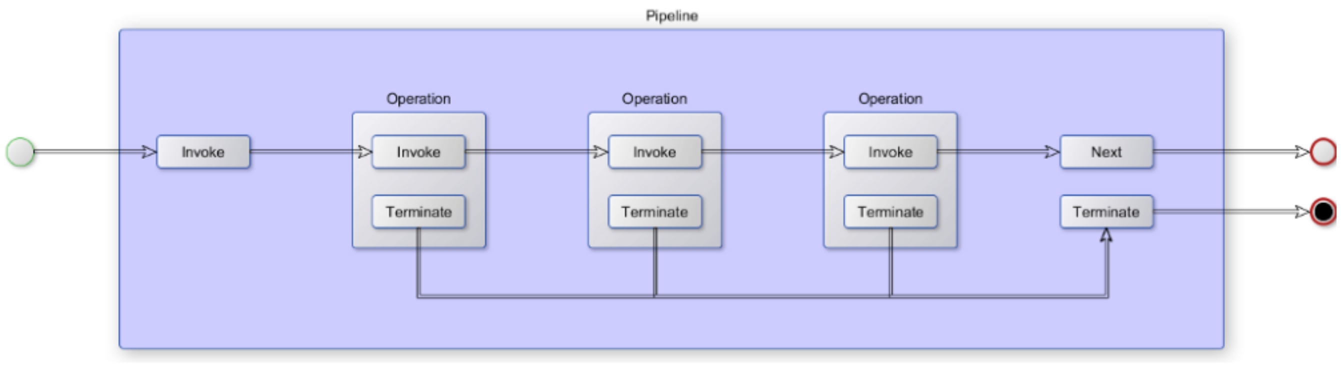
When using this pattern, we must remember :

- In a pipelined algorithm, the concurrency is low until all the stages are doing their task. The concurrency drops when a stage/stages are idle (due to low load). A pipeline algorithm is only effective when it is serving a load higher than a certain threshold. The load threshold is defined as the minimum load that fills up the entire pipeline i.e all the stages are processing data/ items.

- A problem occurs if the stages in a pipeline vary in their computational effort. The slowest stage becomes a bottleneck for the algorithm's aggregated output. If this slow stage were present in the middle of the pipeline, it can cause data items to pile up in the input queue, potentially resulting in buffer overflow problems.

Implementation

1. The most basic implementation of a pipeline would be a simple sequence of operations. The interface of an operation can be invoked to process data.
2. The pipeline processes each operation one by one.
3. The operation can be written in a dedicated class. Or use a wrapper to automatically create an operation from a lambda.
4. The pipeline operations should be registered before the pipeline is invoked.
5. The first feature you want to add to your pipeline is to add a circuit breaker. Each operation will return the result : fail or success.If an operation fails, the pipeline execution for that command/item should stop.
6. Another requirement could be to have a pipeline that can deal with asynchronous operations. Every operation will now have to call the next operation in the pipeline, after they are finished processing the data.
7. One can use a pipeline to optimize the processing of a huge amount of data by running each operation of the pipeline in a dedicated thread.
8. Each thread will consume and produce data from a concurrent queue that will act as a buffer.
9. For our project, the idea is to have different threads to process incoming orders. When an order is finished processing, we check the status of the order.
10. Each order processor is isolated in a dedicated thread, so you can optimize how you store the data and have direct memory access without using a lock.
11. The payment order processor is the only thread that can access user balances. It can get or update any balance without concern about concurrency issues.The pipeline is processing the sequence of operations as fast as possible.



Sample Code

IOperation: an interface defining the *invoke* method (implemented in Processor class). Next is used to link the registered operations and terminate is used to implement the circuit breaker.

```
template <typename T>
class IOperation {
public:
   IOperation<T>* Next;
   IOperation<T>* Terminate;
   virtual void invoke(T&) = 0;
};
```

Processor: <abstract class> defines how to run the pipeline. if the operation (concrete operation class) returns true after being processed, it goes to the next stage, else circuit breaker is called and it goes
to the last stage directly.

```
template<typename T>
class Processor : public IOperation<T> {
private:
        std::thread _t;
        Buffer<T> queue;

        void Run() {
          while(true) {
        T data = queue.pop();
        auto operation = Process(data) ? Next : Terminate;
        if (operation != nullptr) {
                operation->invoke(data);
        }
          }
        }

        virtual bool Process(T& data) = 0;

public:
        Processor():
        _t(&Processor::Run, this), stopLoop(false)
        {}

        void invoke(T& data) { queue.push(data); }
};
```

Example of a Processor --- This processor prints the data onto stdout.

```
class ExampleProcessor : public Processor<int> {
protected:
        bool Process(int& data) {
            std::cout << "Hello World: " << data << std::endl;
            return true;
        }
};
```

<u>Pipeline:</u> First registers the operations in the order specified and is then invoked.

```
template <typename T>
class Pipeline : public IOperation<T> {
private:
   std::vector<IOperation<T>*> operations;

public:
   Pipeline() {};
   void registerOperation(IOperation<T>*);
   void invoke(T&);
   void terminate();

};

template <typename T>
void Pipeline<T>::registerOperation(IOperation<T>* operation) {
   if (!operations.empty()) {
         operations.back()->Next = operation;
   }
   operations.push_back(operation);
}

template <typename T>
void Pipeline<T>::invoke(T& data) {
   IOperation<Order>* op = (!operations.empty()) ? operations.front() : nullptr;
   if (op == nullptr)
         std::cout << "PIPELINE EMPTY!" << std::endl;
   op->invoke(data);
}
```

Known Uses

1) Maven Build Lifecycle (Software Project management tool)
2) UNIX Shell Pipes
3) Machine Learning models (Tensorflow)

Related Patterns

<u>**Chain of Responsibility**</u> **-** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
The pipeline pattern though similar to the Chain of Responsibility pattern in the sequential linked list of handlers, it differs in the principle that in Chain of Responsibility, only **one handler** can process the data i.e either one or none. If a handler is unable to process the data, it passes the data to the handler next in line. Whereas in pipeline, the data is **processed by all stages** in some form or the other.