1. 스테이트 패턴(State Pattern)

1) 스테이트 패턴

- 객체의 내부 상태가 바뀜에 따라서 객체의 행동을 바꿀 수 있으며, 마치 객체의 클래스가 바뀌는 것과 같은 결과를 얻을 수 있는 패턴
- 객체 지향 방식으로 상태 기계를 구현하는 행위 소프트웨어 디자인 패턴
- 상태 패턴 인터페이스의 파생 클래스로써 각각의 상태를 구현하고, 패턴의 슈퍼클래스에 의해 정의되는 메소드를 호출하여 상태 변화를 구현함으로써 상태 기계를 구현함
- 프로시저형 상태 기계를 쓸 때와는 달리 각 상태를 클래스를 이용하여 표현하게 됨
- 스테이트 패턴을 이용할 경우 디자인에 필요한 클래스의 개수가 늘어날 수 있음
- 스트래티지 패턴(Strategy Pattern)과 클래스 다이어그램이 동일하지만 그 용도가 다름

  스트래티지 패턴 : 알고리즘군을 정의하고 각각을 캡슐화하여 바꿔 쓸 수 있게 하는 패턴.

2) 상태 기계

- 유한 상태 기계(Finite-state machine, FSM), 유한 오토마톤(Finite automaton)
- 컴퓨터 프로그램과 전자 논리 회로를 설계하는 데에 쓰이는 수학적 모델
- 유한 상태 기계는 유한한 개수의 상태를 가질 수 있는 오토마타, 즉 추상 기계. 이러한 기계는 한 번에 하나의 상태만을 가지게 되고 어떠한 사건에 의해 한 상태에서 다른 상태로 변화할 수 있으며 이를 전이(Transition)이라고 함. 특정한 유한 오토마톤은 현재 상태로부터 가능한 전이 상태와, 이러한 전이를 유발하는 조건들의 집합으로써 정의됨
- 상태 : 전이를 시작하기 위해 대기하고 있는 시스템의 행동적 노드
- 전이 : 어떠한 조건이 만족되거나 이벤트가 발생하였을 때 수행되는 일련의 동작

3) 추상 기계

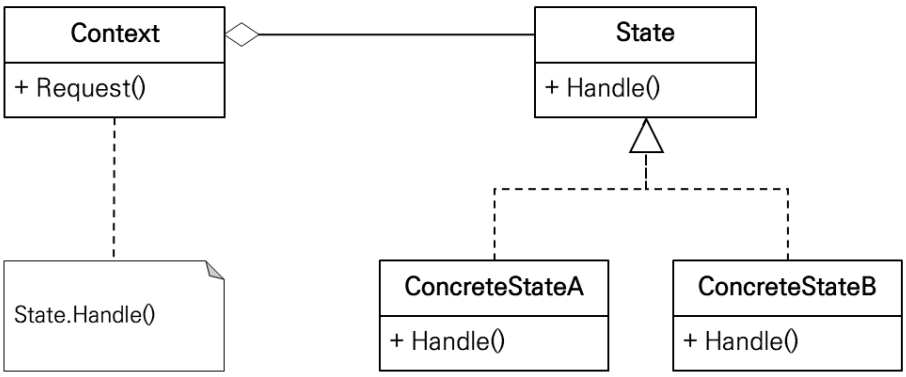
- 컴퓨터 하드웨어나 소프트웨어의 이상적인 모형으로, 추상 컴퓨터라고도 함
- 입력과 출력, 입력을 출력으로 변환시키는 명령어 등으로 구성되며 가장 유명한 예는 튜링 기계
- 추상 기계를 이용해서 컴퓨터 모형을 실제로 만들지 않더라도 실행 시간이나 메모리 같은 컴퓨터 자원이 얼마나 필요한지 예측 가능

4) 튜링 기계

- 긴 테이프에 쓰여 있는 여러 가지 기호들을 일정한 규칙에 따라 바꾸는 기계
- 적당한 규칙과 기호를 입력해 일반적인 컴퓨터 알고리즘을 수행할 수 있으며 컴퓨터 CPU의 기능을 설명하는데 상당히 유용함

5) 클래스 다이어그램

State Pattern_Class



6) 예제

```
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

class Strategy
{
public:
	virtual ~Strategy() {}
	virtual string DoAlgorithm(const vector<string> &data) const = 0;
};

class Context
{
private:
	Strategy *strategy_;

public:
	Context(Strategy *strategy = nullptr) : strategy_(strategy) { }
	~Context() { delete this->strategy_; }

	void set_strategy(Strategy *strategy)
	{
		delete this->strategy_;
		this->strategy_ = strategy;
	}

	void DoSomeBusinessLogic() const
	{
		cout << "Context: Sorting data using the strategy
			(not sure how it'll do it)\n";
		string result = this->strategy_
				->DoAlgorithm(vector<string>{"a", "e", "c", "b", "d"});
		cout << result << "\n";
	}
};

class ConcreteStrategyA : public Strategy
{
public:
	string DoAlgorithm(const vector<string> &data) const override
	{
		string result;
		for_each(begin(data), end(data), [&result](const string &letter)
		{ result += letter; }
		);
		sort(begin(result), end(result));

		return result;
	}
};
```

```
class ConcreteStrategyB : public Strategy
{
        string DoAlgorithm(const vector<string> &data) const override
        {
                string result;
                for_each(begin(data), end(data), [&result](const string &letter)
                { result += letter; }
                );

                sort(begin(result), end(result));
                for (int i = 0; i < result.size() / 2; i++)
                {
                        swap(result[i], result[result.size() - i - 1]);
                }

                return result;
        }
};
//========================================================================================

#include "Strategy.h"


void ClientCode()

{
        Context *context = new Context(new ConcreteStrategyA);

        cout << "Client: Strategy is set to normal sorting.\n";

        context->DoSomeBusinessLogic();

        cout << endl << "Client: Strategy is set to reverse sorting.\n";

        context->set_strategy(new ConcreteStrategyB);
        context->DoSomeBusinessLogic();

        delete context;
}

int main()
{
        ClientCode();
        return 0;
}
```