DESIGN PATTERN
STATE

Group 6:
Lê Duy Bách
Đào Thanh Danh
Nguyễn Khắc Tuấn

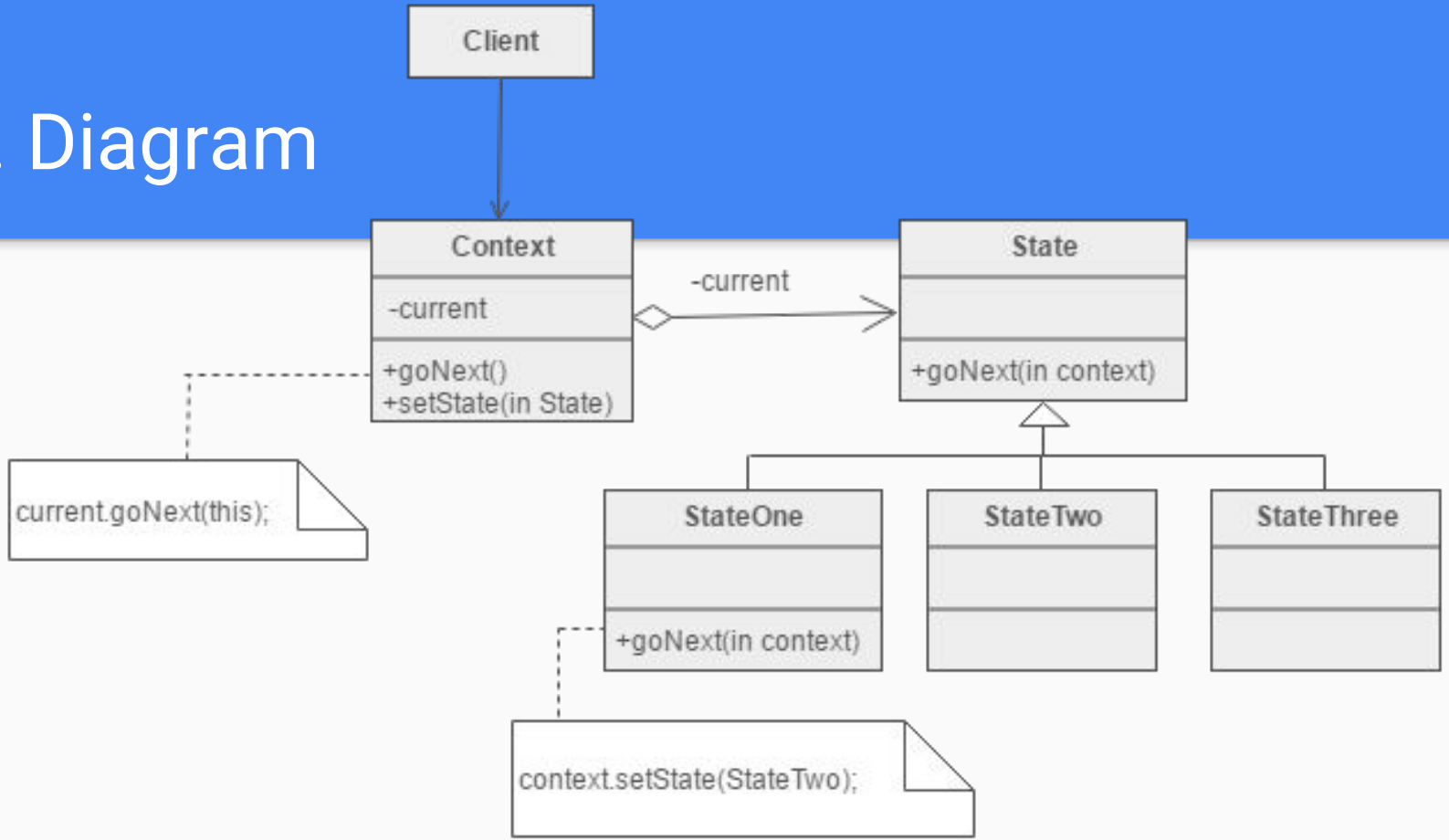
I. introduction

- State is one of behavior patterns.
- In State pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern.
- In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.
- Designers use this programming construct to break complex problems into manageable states and state transitions

II. Problems

A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

III. Diagram


Client
Context
-current
+goNext()
+setState(in State)
-current
State
+goNext(in context)
current.goNext(this);
StateOne
+goNext(in context)
StateTwo
StateThree
context.setState(StateTwo);

IV. Example

The machine has 2 states on or off.

Machine changes state depends on ON state and OFF state is trigger or not.

```
class Machine{ //Context
    class State *current;
    public:
        Machine();
        void setCurrent(State *s){
            current = s;
        }
        void on(); //Request
        void off(); //Request
};

class State{ //State
    public:
    virtual void on(Machine *m){};
    virtual void off(Machine *m){};
};
class ON: public State{ //Concrete State
    public:
        ON(){
            cout << "ON-ctor ";
        };
    void off(Machine *m);
};

class OFF: public State{ //Concrete State
    public:
        OFF(){
            cout << "OFF-ctor ";
        };
    void on(Machine *m);
};
```

```
Machine::Machine(){
    current = new OFF();
}
void Machine::on(){
    current->on(this);
}
void Machine::off(){
    current->off(this);
}
void ON::off(Machine *m){
    cout << "going from ON to OFF";
    m->setCurrent(new OFF());
    delete this;
}
void OFF::on(Machine *m){
    cout << "going from OFF to ON";
    m->setCurrent(new ON());
    delete this;
}
```

V. Summary

- Identify an existing class, or create a new class, that will serve as the "state machine" from the client's perspective. That class is the "wrapper" class.
- Create a State base class that replicates the methods of the state machine interface. Each method takes one additional parameter: an instance of the wrapper class.
- Create a State derived class for each domain state. These derived classes only override the methods they need to override.
- The wrapper class maintains a "current" State object.
- All client requests to the wrapper class are simply delegated to the current State object, and the wrapper object's this pointer is passed.
- The State methods change the "current" state in the wrapper object as appropriate.

VI. References

https://sourcemaking.com/design_patterns/state

http://www.journaldev.com/1751/state-design-pattern-java

https://sourcemaking.com/design_patterns/state/cpp/1

https://ideone.com/IM6CDa