Behavioral Design Pattern

Instructor: Mehroze Khan

Behavioral Design Pattern

- Behavioral design patterns are concerned with algorithms and the **assignment of responsibilities** between objects.
- Behavioral patterns describe not just patterns of objects or classes but also the **patterns of communication** between them.

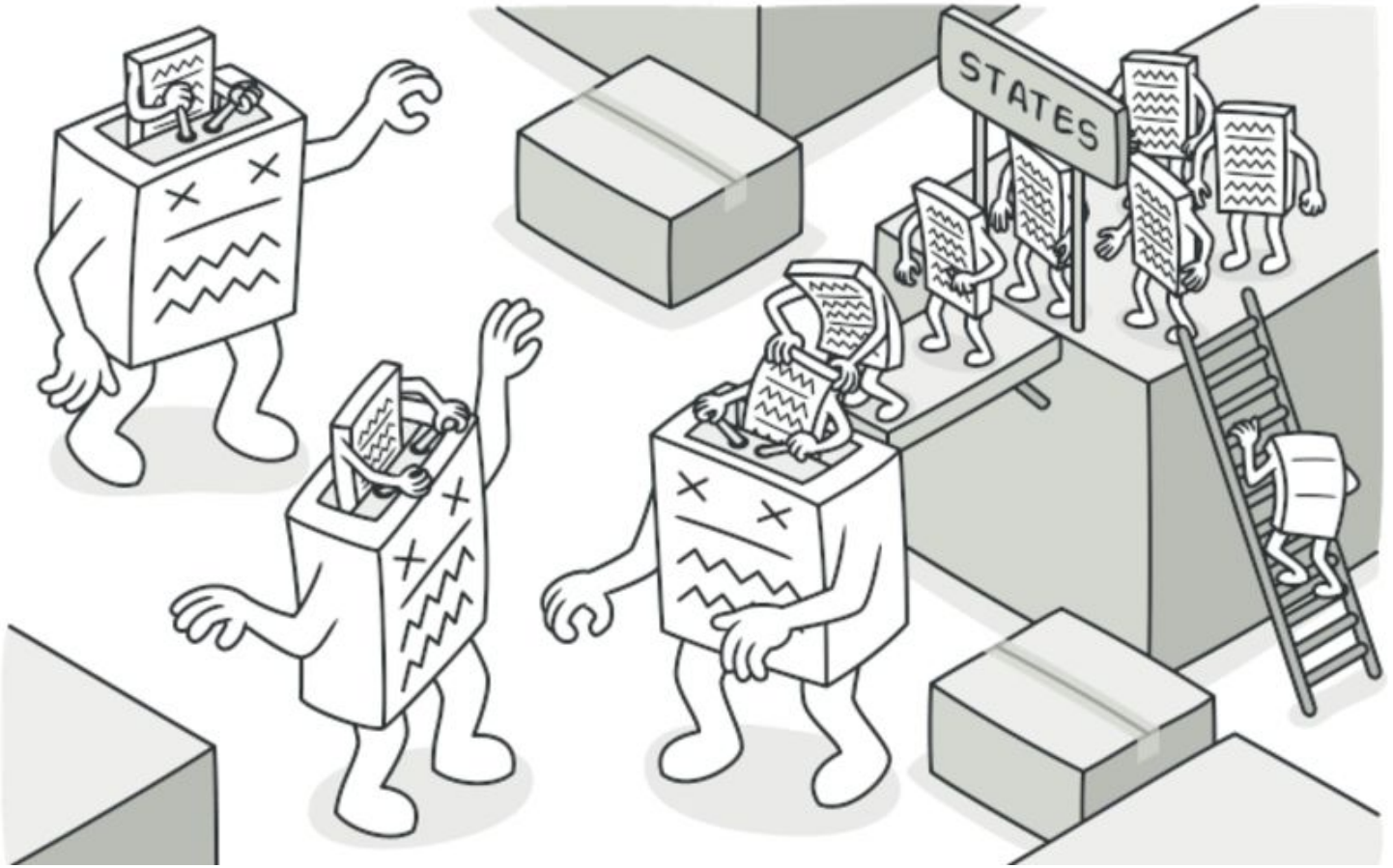
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

State Pattern

State Pattern

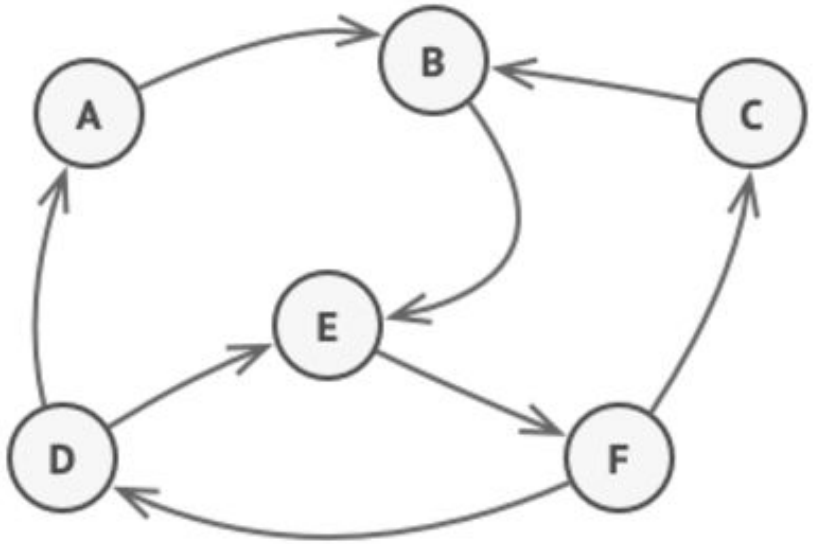
Intent:

- **State** is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



Problem

- The State pattern is closely related to the concept of a **Finite-State Machine.**
- At any given moment, there's a **finite number of states** which a program can be in.
- Within any unique state, the program **behaves differently**, and the program can be **switched** from one state to another instantaneously.
- Depending on a current state, **the program may or may not switch to certain other states**.
- These switching rules, called **transitions**, are also finite and predetermined.



Finite-State Machine.

Problem

- You can also apply this approach to objects. Imagine that we have a **Document** class.
- A document can be in one of three states: **Draft, Moderation and Published**.
- The **publish method** of the document works a little bit differently in each state:
  - In **Draft**, it moves the document to moderation.
  - In **Moderation**, it makes the document public, but only if the current user is an administrator.
  - In **Published**, it doesn't do anything at all.

Possible states and transitions of a document object.

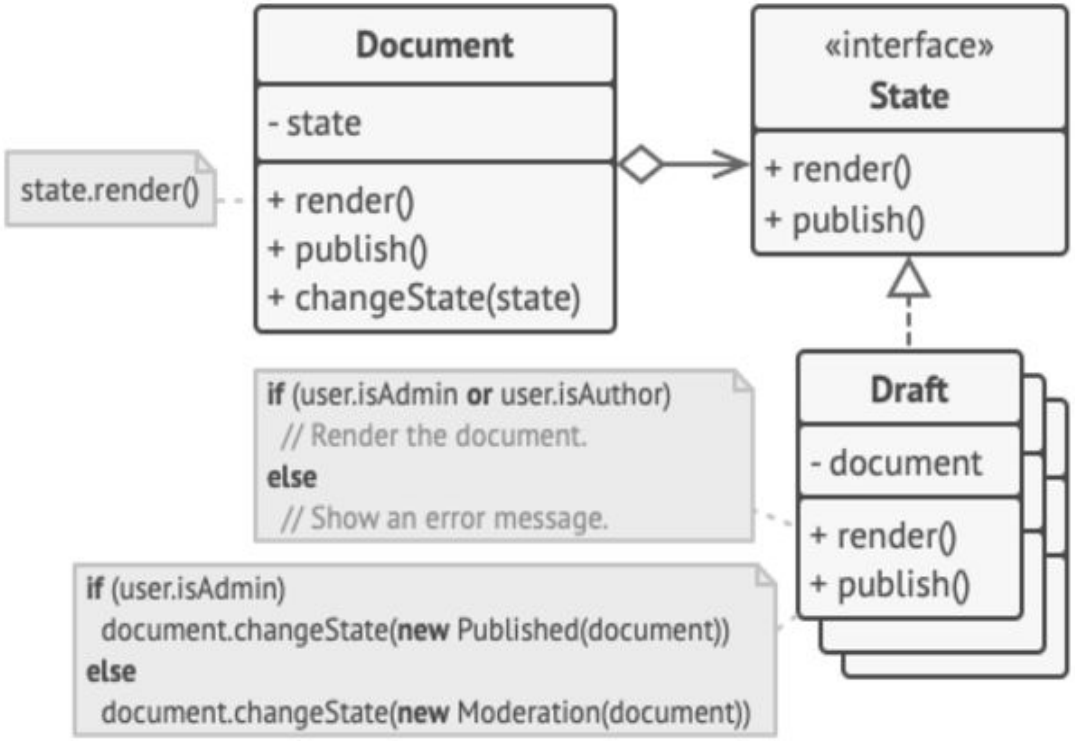
Problem

- State machines are usually implemented with lots of **conditional statements** (if or switch) that select the appropriate behavior depending on the current state of the object.
- Usually, this “state” is just a **set of values** of the **object’s fields**.
- Most methods will contain monstrous **conditionals** that pick the proper behavior of a method according to the current state.
- Code like this is very difficult to maintain because any **change to the transition logic** may require changing state conditionals in every method.

```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            "published":
                // Do nothing.
                break
    // ...
```

Solution

- The State pattern suggests that you **create new classes for all possible states** of an object and extract all state-specific behaviors into these classes.
- Instead of implementing all behaviors on its own, the original object, called **context**, stores a **reference to one of the state objects** that represents its current state, and delegates all the state-related work to that object.
- To transition the context into another state, replace the active state object with another object that represents that new state. This is possible only if all state classes follow the **same interface** and the context itself works with these objects through that interface.

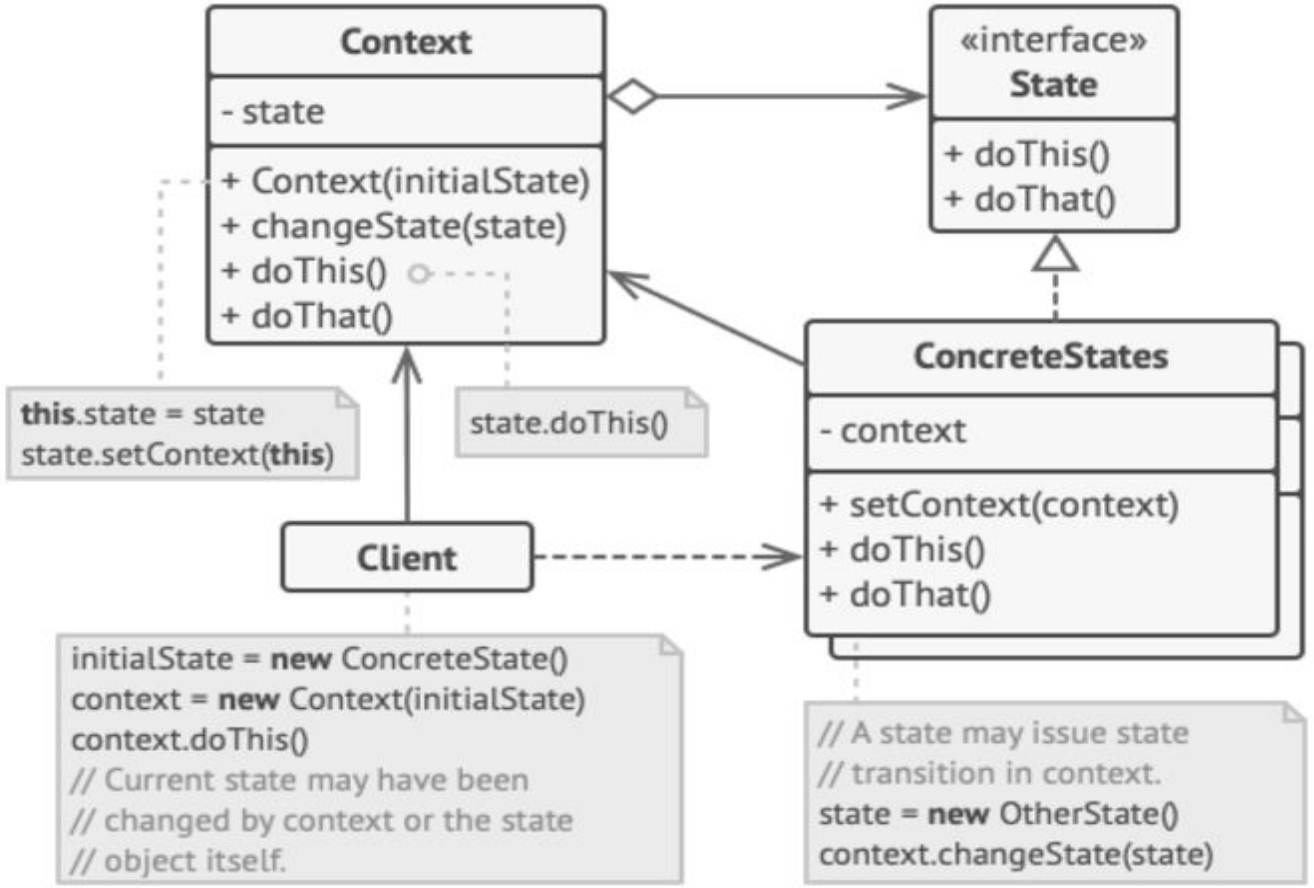


Document delegates the work to a state object.

Structure

1 **Context** stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

2 The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will never be called.



3 **Concrete States** provide their own implementations for the state-specific methods. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

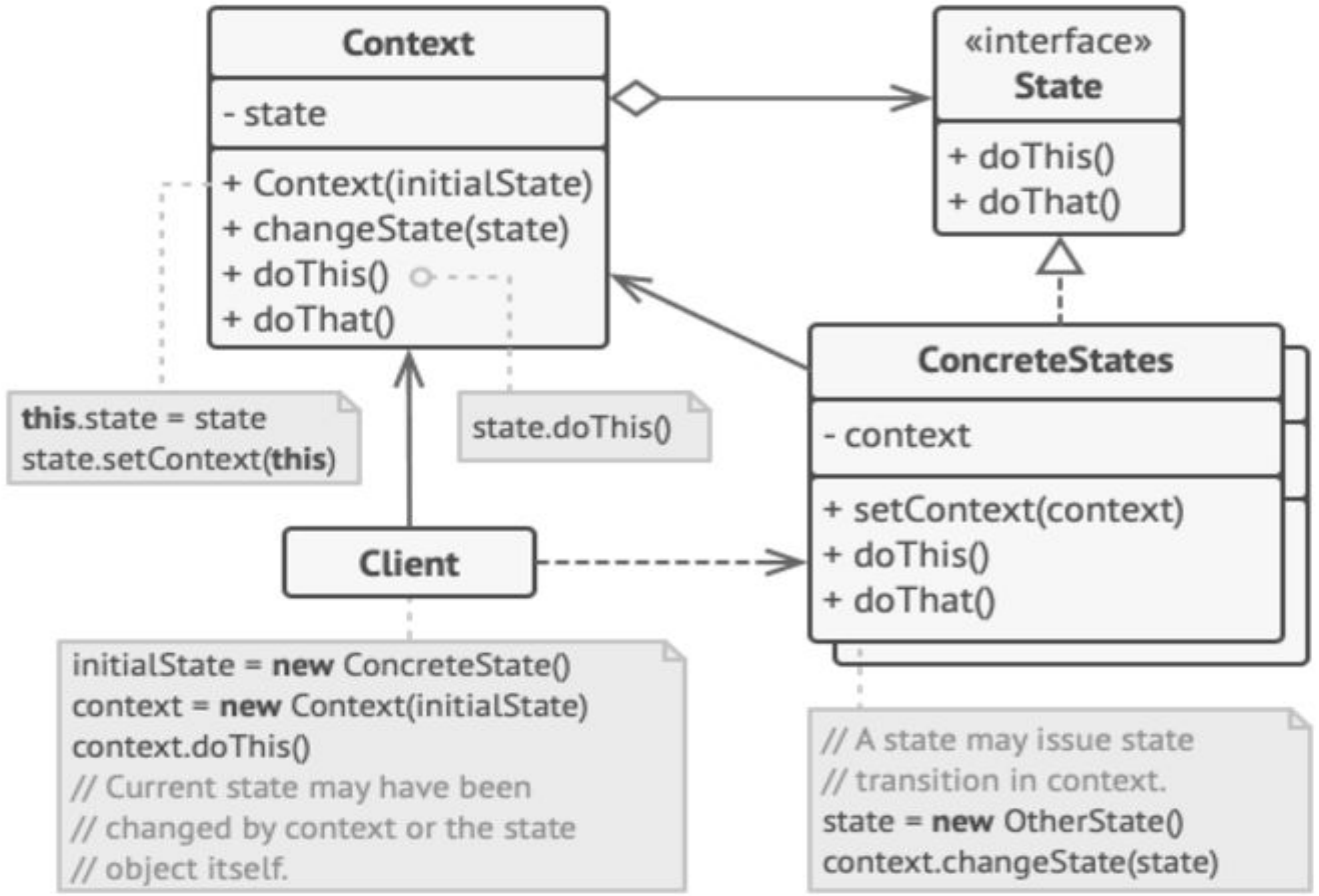
State objects may store a backreference to the context object. Through this reference, the state can fetch any required info from the context object, as well as initiate state transitions.

4 Both context and concrete states can set the next state of the context and perform the actual state transition by replacing the state object linked to the context.

Structure

1 **Context** stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

2 The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will never be called.



3 **Concrete States** provide their own implementations for the state-specific methods. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

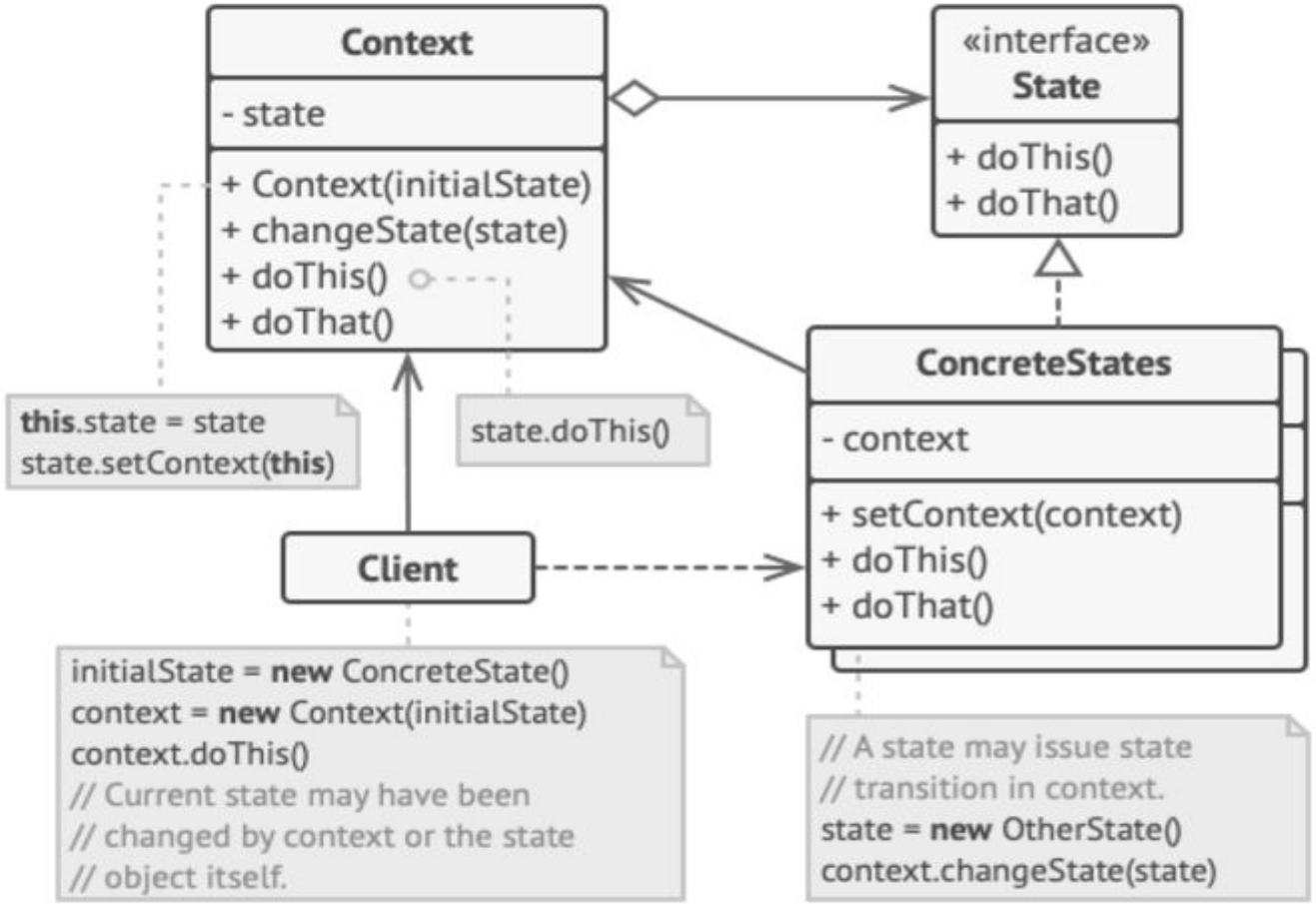
State objects may store a backreference to the context object. Through this reference, the state can fetch any required info from the context object, as well as initiate state transitions.

4 Both context and concrete states can set the next state of the context and perform the actual state transition by replacing the state object linked to the context.

Structure

1 **Context** stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

2 The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will never be called.



3 **Concrete States** provide their own implementations for the state-specific methods. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

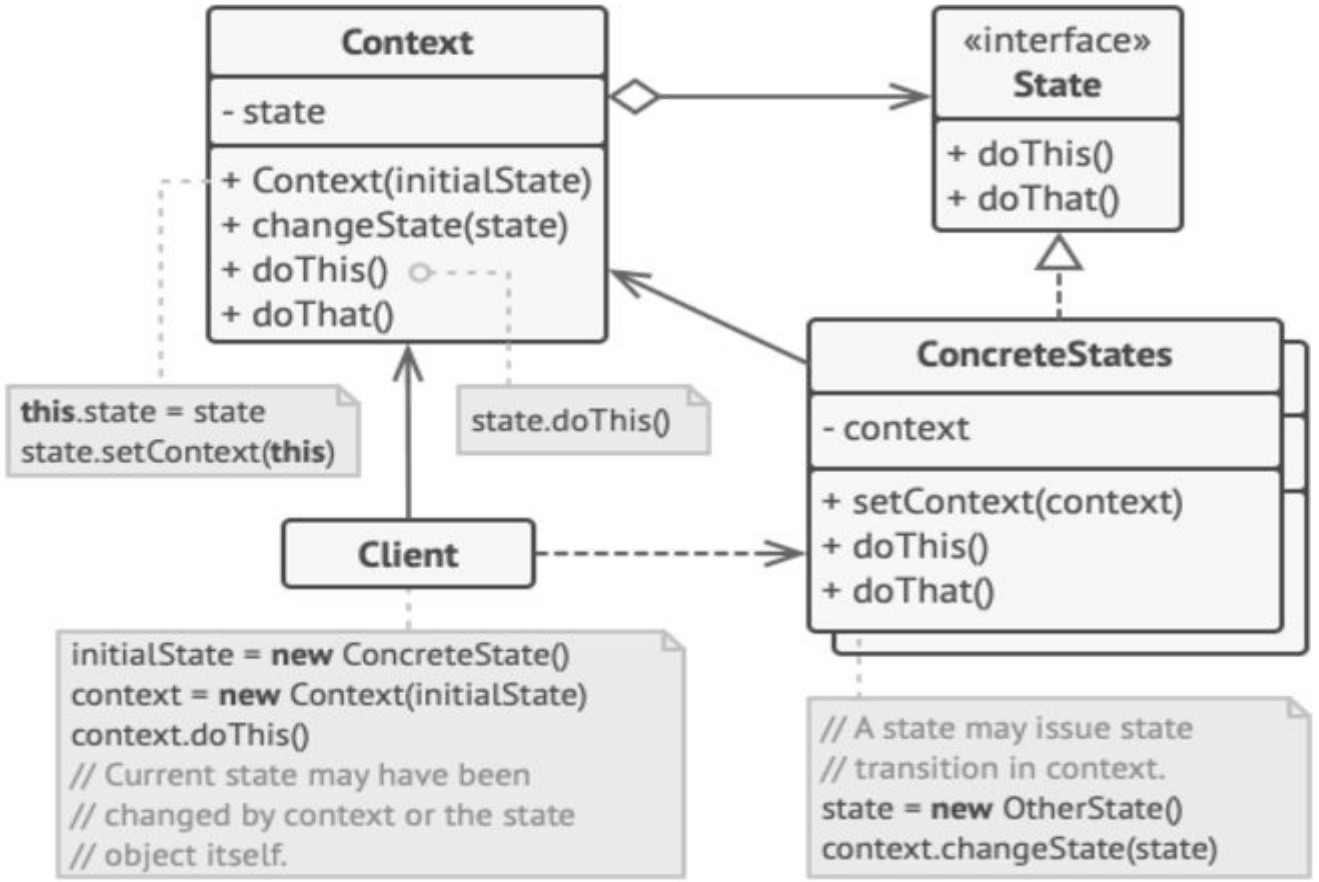
State objects may store a backreference to the context object. Through this reference, the state can fetch any required info from the context object, as well as initiate state transitions.

4 Both context and concrete states can set the next state of the context and perform the actual state transition by replacing the state object linked to the context.

Structure

1 **Context** stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

2 The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will never be called.



3 **Concrete States** provide their own implementations for the state-specific methods. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

State objects may store a backreference to the context object. Through this reference, the state can fetch any required info from the context object, as well as initiate state transitions.

4 Both context and concrete states can set the next state of the context and perform the actual state transition by replacing the state object linked to the context.

Example

```
// Context Class
class Document {
private:
    unique_ptr<State> currentState; // Current state of the document

public:
    Document(State* initialState) {
        currentState = initialState;
        currentState->setContext(this);
    }

    void setState(State* newState) {
        currentState.reset(newState);
        currentState->setContext(this);
    }

    void publish() {
        currentState->publish(); // Delegate directly to current state
    }

    void edit() {
        currentState->edit(); // Delegate directly to current state
    }
};
```

Example

```
// State Interface
class State {
protected:
    Document* context; // Back-reference to the context

public:

    void setContext(Document* ctx) {
        context = ctx;
    }

    virtual void publish() = 0; // Behavior for publishing
    virtual void edit() = 0;    // Behavior for editing
};
```

Example

```
// Concrete State: Draft
class DraftState : public State {
public:
    void publish() override {
        cout << "Document is now under moderation.\n";
        context->setState(new ModerationState());
    }

    void edit() override {
        cout << "Editing the draft document.\n";
    }
};
```

```
// Concrete State: Moderation
class ModerationState : public State {
public:
    void publish() override {
        cout << "Document is now published.\n";
        context->setState(new PublishedState());
    }

    void edit() override {
        cout << "Cannot edit the document. It's under moderation.\n";
    }
};
```

```
// Concrete State: Published
class PublishedState : public State {
public:
    void publish() override {
        cout << "Document is already published. No action taken.\n";
    }

    void edit() override {
        cout << "Cannot edit the document. It's already published.\n";
    }
};
```

```
// Main Function
int main() {
    // Start with the Draft state
    Document doc(new DraftState());

    // Perform actions
    doc.edit();
    doc.publish();
    doc.edit();
    doc.publish();
    doc.publish();

    return 0;
}
```

References

- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Pearson, 1995.

Helping Links:

- https://refactoring.guru/design-patterns/