**Name:** State

**Category:** Behavioral

**Description:**

The idea behind the State behavioral programming pattern is to have an object whose behavior changes depending on its current state. The state can be described as a status that determines how an object performs an action that it’s being asked to do.

For example, consider the play/pause button on a music-playing application to be an object and one of its actions is to display an icon for the button. In this scenario, a play or pause button icon is displayed, depending on whether the music is playing or is paused, so it could have the status options: “musicPlaying” or “musicPaused”. If the status is currently “musicPlaying”, it’ll show the pause button, if the state is currently “musicPaused”, it’ll show a play button.

An implementation of the pattern would have a Context class, a State interface, and a number of ConcreteState classes implementing the State interface. The Context class acts as a way for a user to interact with the state object, the State interface acts as a template to standardize how to build the different ConcreteStates, and the ConcreteState classes are implementations of that template that define the specific behavior of actions for each state.

If we were to implement the play/pause button scenario, the Context could be a box containing the pause/play button that users click and define what state it needs to switch to when a user clicks it, the State could be a template that includes an action, “DisplayIcon()”, and the ConcreteStates could be “Playing” and “Paused”, where the “Playing” state’s “DisplayIcon()” implementation displays the pause icon, and the “Paused” state’s “DisplayIcon()” implementation displays the play icon.lic

**When to Use:**

When you have an object whose behavior needs to change dynamically depending on a state, especially when the alternative is a large collection of conditional statements that can be divided in a modular way

**Advantages:**

Overall:

* Since each state is implemented in its own subclass, it can be easier to read

If the state transition logic lives inside of each ConcreteState subclass:

* The logic becomes modular and it’s easier to add new states since the logic determining the transition patterns for a state is isolated to the subclass
* The transition logic also becomes easier to read since it’s partitioned out, where the alternative could be a large switch statement or something similar that becomes confusing very quickly

**Disadvantages:**

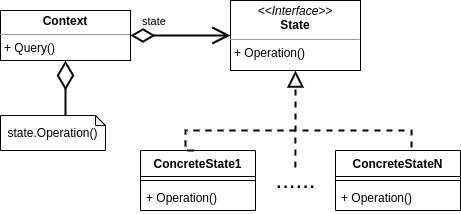
Overall:

* Since each state is implemented in its own subclass, the code is less compact and can be harder to keep track of

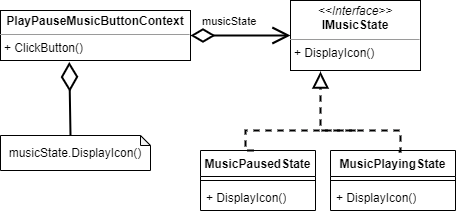
If the state transition logic lives inside of each ConcreteState subclass:

* The state subclasses will be dependent on each other since they’ll need rely on each other to determine the current state

**General UML:**



**Specific UML:**

****