# State Pattern

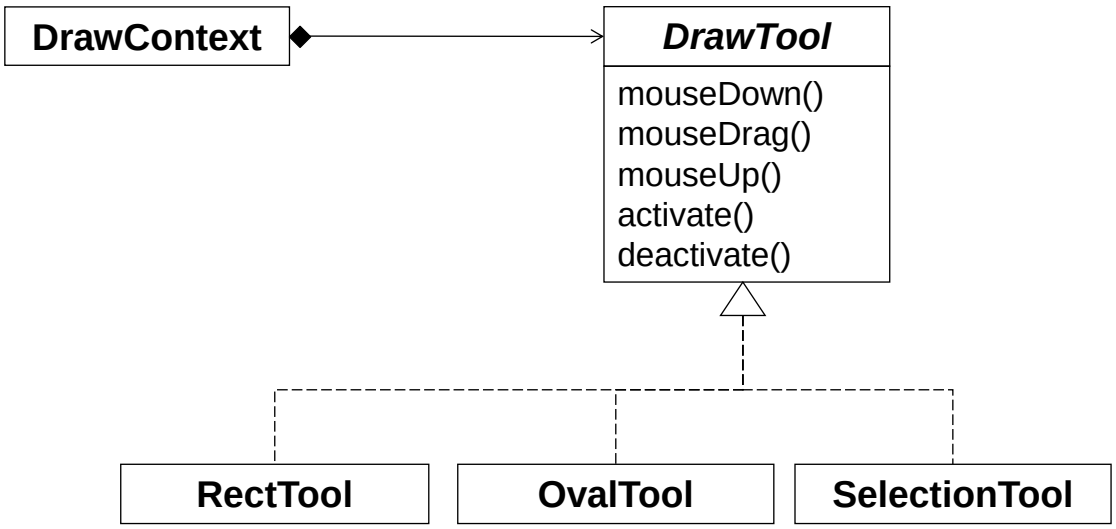
## Goal

Outsourcing of state specific behavior into separate classes. Allows to change the behavior of an object by exchanging its state object.

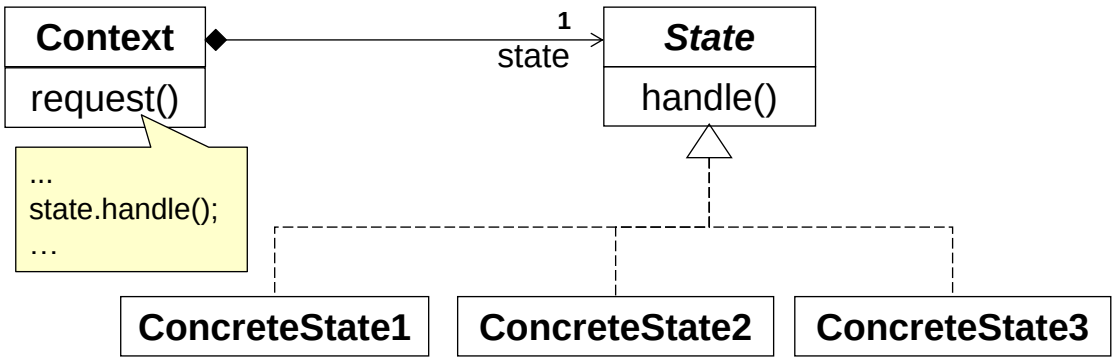
## Motivation

The view of our graphics editor has a reference to a tool instance implementing interface `DrawTool`. This reference defines

a) the current drawing mode, and
b) the behavior whenever the mouse is pressed.



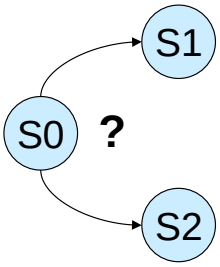

## Structure




| | |
|---|---|
| Context: | stores a reference to a concrete implementation of the `State` interface which represents the current state. |
| State: | Defines the interface to access and invoke the state dependent behavior. |
| Concrete State: | Implements the state dependent behavior. |

**Remark: How to set the (next) state?**

- The state object might itself initiate a state transition (e.g. S0 to S1 or S2)




  - S0 needs to know the other states (S1, S2)
  - S0 must have access to the context in order to set the new state.

- The state transition might be performed by the context (based on an indication from the current state instance, i.e. the value which triggered the state transition)

  The state machine (i.e. the mapping of state to concrete state objects) can then be defined in the context, e.g. as a table
  + configurable
  + separation of transition and state instances
  + change of the state machine (i.e. order of the states) is possible without changing the states

- State transition “from the outside”

DrawView:
```
setTool (DrawTool tool) {
   if (tool != null) {
      this.tool.deactivate();
      this.tool = tool;
      this.tool.activate();
   } else {
      // only if `null` is not a valid state
      throw new IllegalArgumentException();
  }
}
```

The test for the parameter being non-`null` asserts that after invoking method `setTool` a tool instance is set. Thus, code which is using the currently referenced tool does not have to check whether a tool is set or not (invariant: *this.tool != null*). Obviously, this invariant must be established in the constructor in order to be valid over the whole lifetime of the context.

## Applications

Application of the state pattern is advisable if your code contains at different places conditional statements which all depend on *one* attribute.

Example:
```
if (state == RECTANGLE) {
  // Rectangle related code here
else if (state == CIRCLE) {
  // Circle related code here
}
```