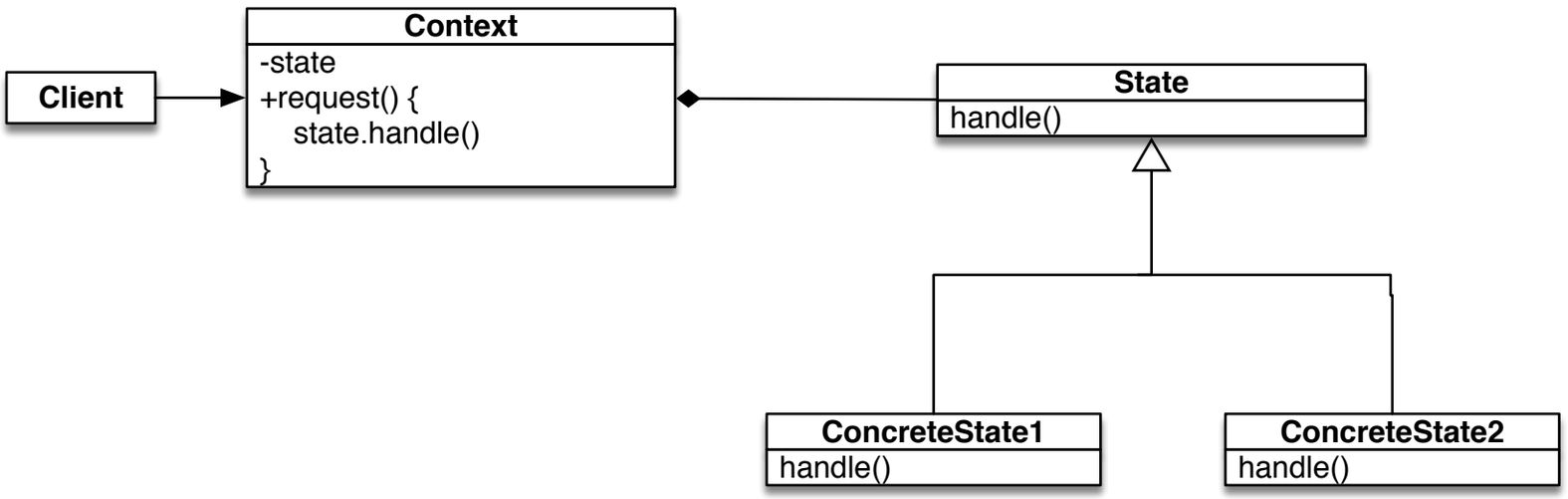
State



Intent: to model an object whose behavior of the same operation depends on the runtime state it's in.

E.g. A TCPConnection class can be in Closed, Listen or Established states at runtime, and in each different state, the call open(), close() or acknowledge() results in different actions.
Instead of a big switch case statement in the corresponding calls, we can add a polymorphic TCPState member in TCPConnection whose implementation can be TCPClosed, TCPListen or TCPEstablished, and open(), close() or acknowledge() call forwards the implementation to calls with the same name in State. When transitioning between states, State variable gets instantiated with a different implementation.

In addition to getting rid of giant if-else statements, State pattern makes state transitions explicit. (And happens via rebinding one variable atomically instead of several.)
If ConcreteStates don't have states internally, they can be shared by multiple contexts, essentially like a **FlyWeight** without intrinsic states.

In practice, state transitions can be triggered inside Context object or each ConcreteState.
States can be instantiated upfront and never freed (Context keeps reference to each and decides when to switch to particular ones; in this sense State objects are often **Singletons**), or instantiated dynamically and destroyed each time a state is entered / left from. Pros and cons.

(Just judging by its looks, this looks quite like Strategy pattern as well: method of Context can have multiple concrete algorithms implementing it, and Context calls the implementation through a Strategy interface, with call site often times ignorant of which concrete algorithm is used. Strategy has less emphasis on state transitions and runtime behavior changing based on a changing State.)