# State

GoF object behavioral

Operation pattern

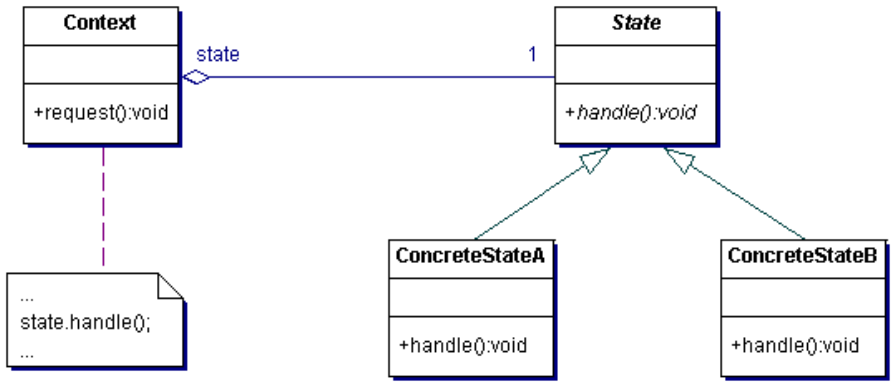
# State

- **Intent**
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
  - Typically to implement behavior changing according to a state transition diagram
- **Applicability**
  - A big 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 implement flow of control based on the state of the application. Often the control code is duplicated in many methods. State pattern will move each branch of the conditional logic in a separate class.

# State

- **How it works**
  - The State pattern is a solution to the problem of how to make behavior depend on state.
  - Define a "context" class to present a single interface to the outside world.
  - Define a State abstract base class.
  - Represent the different "states" of the state machine as derived classes of the State base class.
  - Define state-specific behavior in the appropriate State derived classes.
  - Maintain a pointer to the current "state" in the "context" class.
  - To change the state of the state machine, change the current "state" pointer.

# State - Structure



# State - Consequencies

- Localize state-specific behavior and partitions for different states
  - The state pattern puts all behavior associated with a state into one object.
  - New states can be added easily by defining new subclasses. Compare this with the alternative of changing the conditional logic of all methods in a single class.
  - As the number of states grow, the number of classes may become large. This is however good design, the alternative would be unmanageable.
- It makes state transitions explicit, localize state transition code if implemented in the context
  - The alternative would be to define the state in terms of internal data values. When the state changes would not be seen easily in the code, but scattered in the complicated conditional logic.
- State objects can be shared if they do not require instance variables
  - In this case state objects are essentially an instance of the Flyweight pattern.

# State - discussion

- The State pattern does not specify where the state transitions will be defined. The choices are two
  1) In the "context" object (applicable if the criteria for state transitions are fixed)
  2) In each individual State derived class
  - The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses. [GOF, p308]

- Creating and Destroying State Objects – two options
  - Create state object when needed
  - Create states once as singletons

# State – related patterns

- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, *Advanced C++*, p58]. They differ in intent - that is, they solve different problems.
- The implementation of the State pattern builds on the Strategy pattern. The difference between State and Strategy is in the intent.
  - With Strategy, the choice of algorithm is fairly stable.
  - With State, a change in the state of the "context" object causes it to select from its "palette" of Strategy objects. [Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, 1999, p253]
- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope (context) classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently. [Coplien, *C++ Report*, May 95, p58]

# State vs. strategy in practise – differencies in ...

- **Change of operation implementation at runtime**
- Strategy
  - Context object usually contains one of several possible ConcreteStrategy objects
  - Once created the ConcreteStrategy seldom changes at runtime. Different strategies are used in different context objects.
- State
  - Context object changes its ConcreteState object over its lifetime as the corresponding finite state machine ‘tics’.
- **What the client sees**
- Strategy
  - All ConcreteStrategies do the same thing, but differently
  - Clients of Context do not see any difference in behavior in the Context
- State
  - ConcreteStates act differently
  - Clients see different behavior in the Context
- In other words, Strategy uses polymorphism to allow *different implementations for an operation*, whereas State uses polymorphism to allow *different operations* depending on the state.