

9

State

Diseño del Software

Grado en Ingeniería Informática del Software

2024-2025

Patrón de comportamiento, de objetos

Permite a un objeto alterar su comportamiento cuando cambia su estado interno. Parecerá como si el objeto hubiera cambiado su clase.

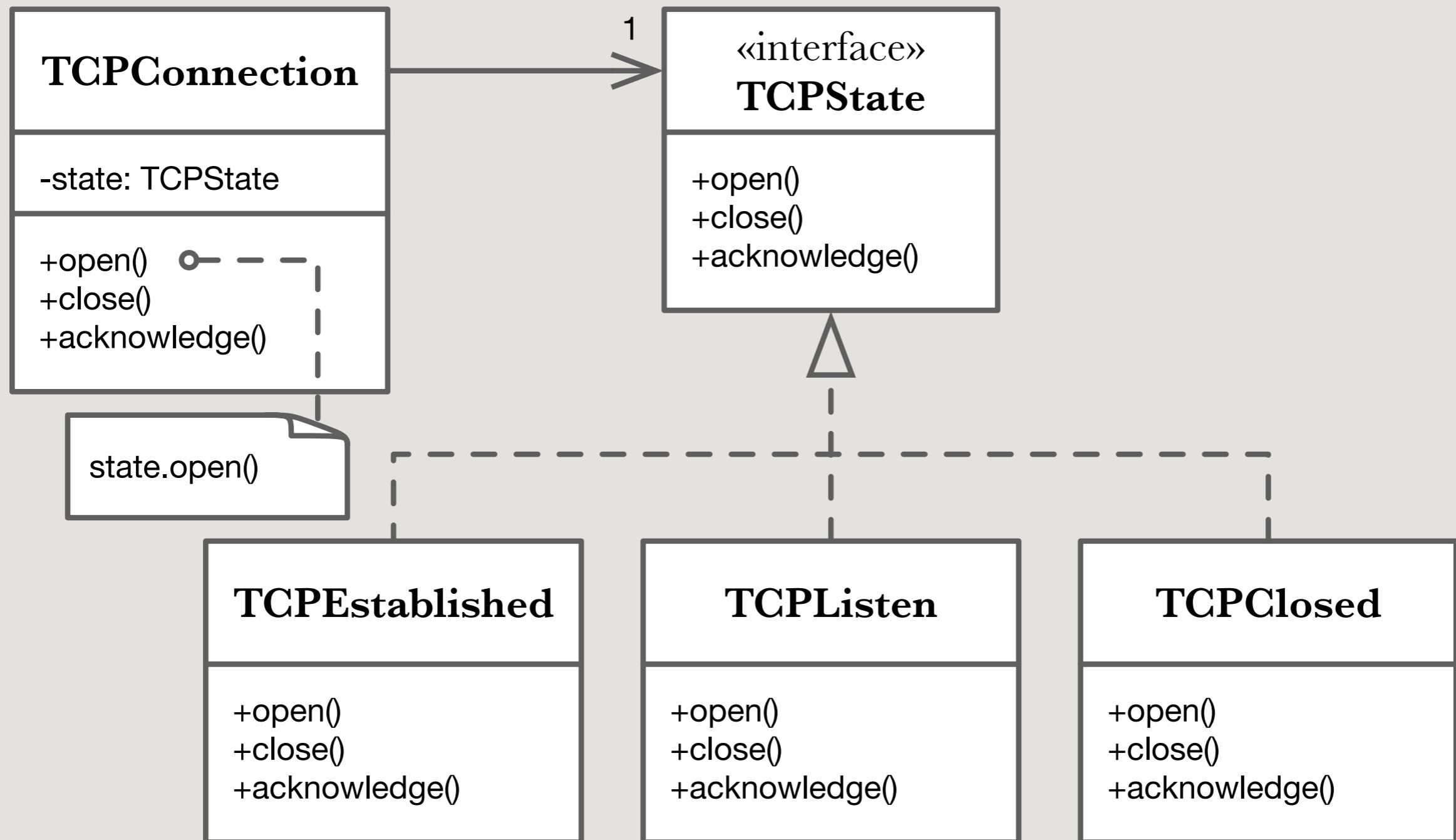
**También conocido
como
Estados como objetos**

Motivación

Sea una clase `TCPConnection` que representa una conexión de red.

Un objeto TCPConnection puede estar en uno de varios estados diferentes: establecida, escuchando, cerrada... Cuando un objeto TCPConnection recibe peticiones de otros objetos, responde de forma diferente dependiendo de su estado actual.

Por ejemplo, el efecto de una llamada a open depende de si la conexión está cerrada o establecida.



Aplicabilidad

Úsese el patrón State en
cualquiera de los casos siguientes

**El comportamiento de un objeto
depende de su estado, y este
puede cambiar en tiempo de
ejecución.**

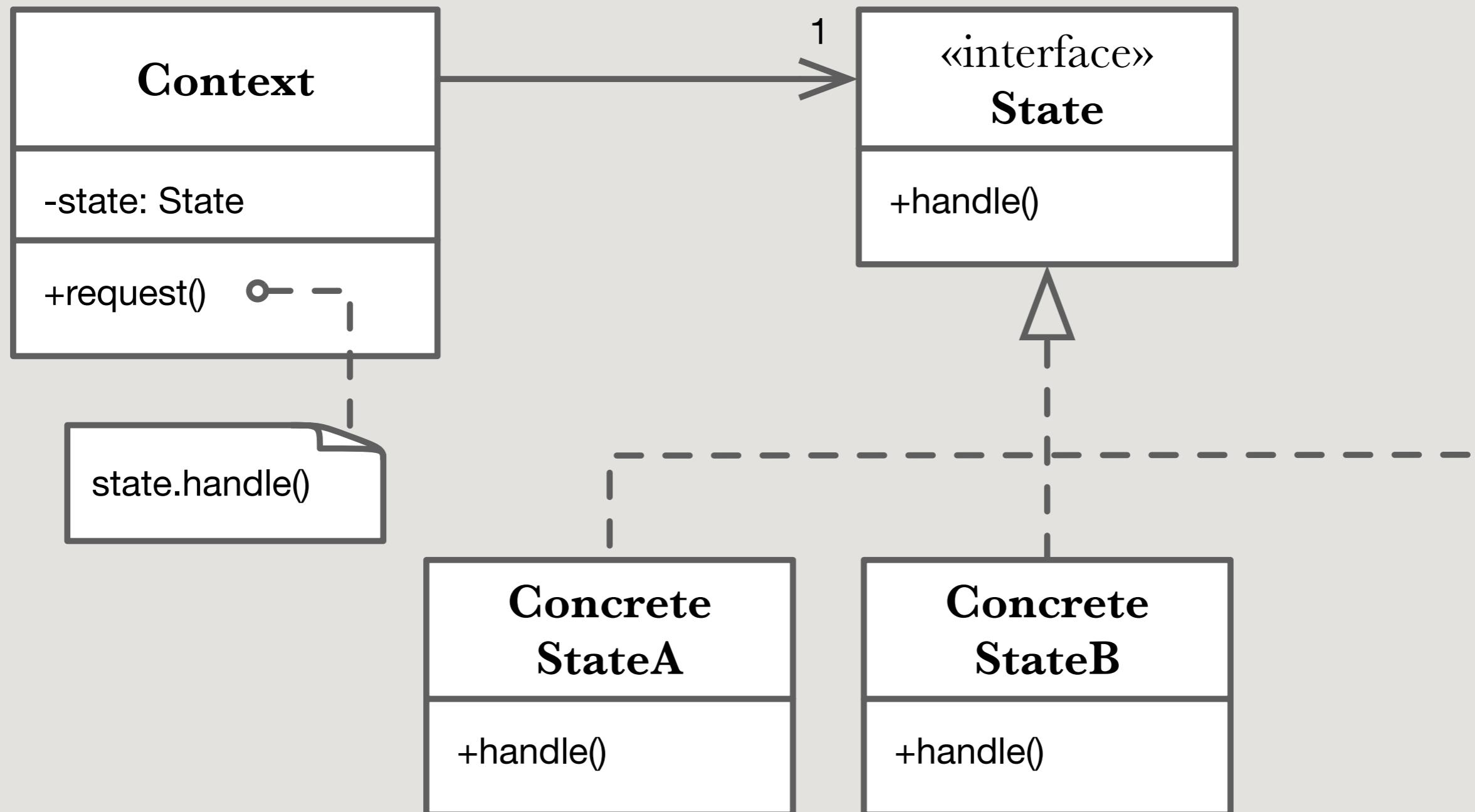
Las operaciones tienen múltiples sentencias condicionales que tratan con los estados.

Siendo este estado representado normalmente como una serie de constantes.

A menudo son varias las operaciones en las que se repite esa misma lógica condicional.

El patrón State mueve cada rama condicional a una clase aparte. Esto permite tratar al estado como un objeto de pleno derecho que puede variar independientemente de otros.

Estructura



La estructura del patrón de diseño State, tal y como aparece en el GoF

Participantes

Context

(TCPConnection)

Define la interfaz que interesa a los clientes.

Mantiene una referencia a una subclase de estado concreto que representa el estado actual.

State

(TCPState)

Define la interfaz para encapsular el comportamiento asociado con el estado del contexto.

ConcreteState

(TCPEstablished, TCPListen, TCPClosed)

Cada subclase implementa las operaciones anteriores para ese estado concreto.

Colaboraciones

El contexto delega las operaciones dependientes del estado al objeto que representa el estado actual.

**El contexto podría pasarse a sí mismo
como parámetro.**

Para que el estado acceda al contexto si es necesario.

Los clientes solo tratan con el contexto.

Una vez que el contexto es inicializado en un determinado estado, los clientes no necesitan (ni deberían) tratar directamente con los estados.

O bien el contexto o bien los estados concretos deciden cuándo se pasa de uno a otro estado.

Consecuencias

**Localiza el
comportamiento
específico del estado y
lo aísla en un objeto**

El patrón State pone todo el comportamiento asociado con un estado particular en un objeto.

Dado que todo el código específico de estado vive en una subclase de estado concreta, se pueden añadir fácilmente* nuevos estados y transiciones definiendo nuevas subclases.

En lugar de tener sentencias condicionales dispersas por toda la implementación del contexto, normalmente en varias operaciones.

* No siempre es tan fácil

Lo veremos en la sección de
implementación.

El patrón State evita este problema pero puede introducir otro, ya que el patrón distribuye el comportamiento de los distintos estados entre varias subclases State.

Esto aumenta el número de clases y es menos compacto que una sola clase. Pero esta distribución es buena si hay muchos estados* que de otro modo requerirían grandes sentencias condicionales.

* O si pueden aparecer otros nuevos.

Encapsulating each state transition and action in a class elevates the idea of an execution state to full object status. That imposes structure on the code and makes its intent clearer.

**Hace explícitas las
transiciones entre
estados**

Implementación

**¿Quién define las
transiciones
entre estados?**

El patrón State no especifica qué participante se encarga de las transiciones.

Si el criterio es siempre el mismo, puede ser el propio contexto.

Normalmente es más flexible y apropiado que sean las propias subclases de los estados quienes designen a su **estado sucesor y cuándo se hace la **transición**.**

Esto requiere añadir una operación al contexto para cambiar su estado de forma explícita.

Esto nos otorga mucha flexibilidad, pudiendo modificar la lógica o añadir estados nuevos simplemente definiendo nuevas subclases.

Tiene el inconveniente, no obstante, de que cada estado concreto conocerá al menos a otro, dando lugar a dependencias de implementación entre las subclases.

Alternativas basadas en tablas

Una tabla podría asignar, para cada estado, cada entrada posible a un estado sucesivo.

Los criterios de transición pueden cambiarse modificando los datos en lugar de cambiar el código del programa.

Habrá que ampliar dicha tabla si es necesario realizar alguna acción en cada transición.

El patrón de estado modela el comportamiento específico de cada estado, mientras que el enfoque basado en tablas se centra más en definir las transiciones entre estados.

Uso de herencia dinámica

El cambio de comportamiento para una operación concreta podría lograrse cambiando la clase del objeto en tiempo de ejecución.

Los pocos lenguajes que lo permiten harían innecesario el uso del patrón como tal, lo implementarían de forma nativa.

Acciones de entrada y salida

A veces es necesario realizar alguna acción no en respuesta a los métodos del estado, sino al entrar^{*}en un estado.

* (0 al salir, o ambas).

Para implementar esto en el patrón State, podríamos añadir métodos de entrada y salida a la interfaz State, como enter() y exit(). Cuando el contexto cambia de estado, se llamaría al método exit() en el estado actual y luego al método enter() en el nuevo estado.

Este concepto —acciones de entrada y salida de los estados— se trata abundantemente en las máquinas de estados (como los diagramas de estado de UML), pero no se aborda explícitamente en el libro.

Este concepto —acciones de entrada y salida de los estados— se trata abundantemente en las máquinas de estados (como los diagramas de estado de UML), pero no se aborda explícitamente en el libro.

Así es como se podría llevar a cabo en la práctica:

Este concepto —acciones de entrada y salida de los estados— se trata abundantemente en las máquinas de estados (como los diagramas de estado de UML), pero no se aborda explícitamente en el libro.

Así es como se podría llevar a cabo en la práctica:

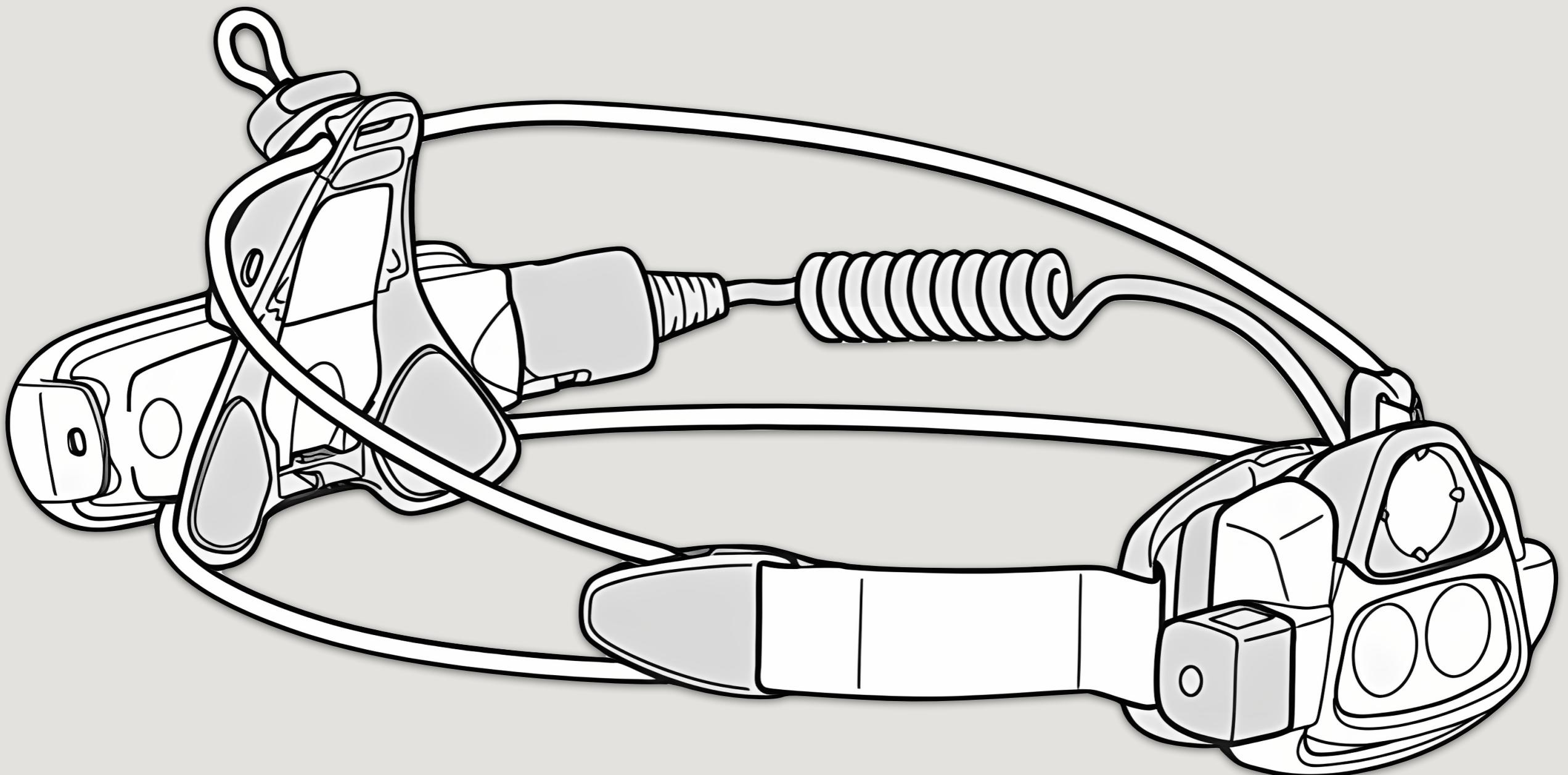
- 1 Acciones de entrada: Cuando un contexto pasa a un nuevo estado, se puede llamar a una acción de entrada para realizar cualquier inicialización o ejecutar determinadas acciones que deben producirse en cuanto el estado se activa.

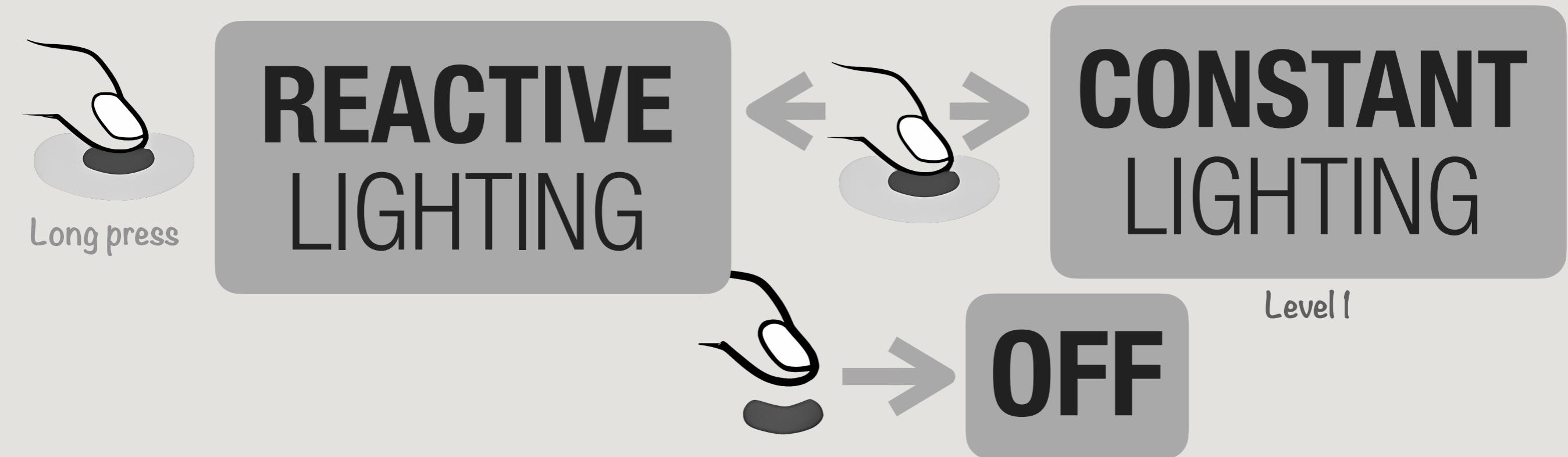
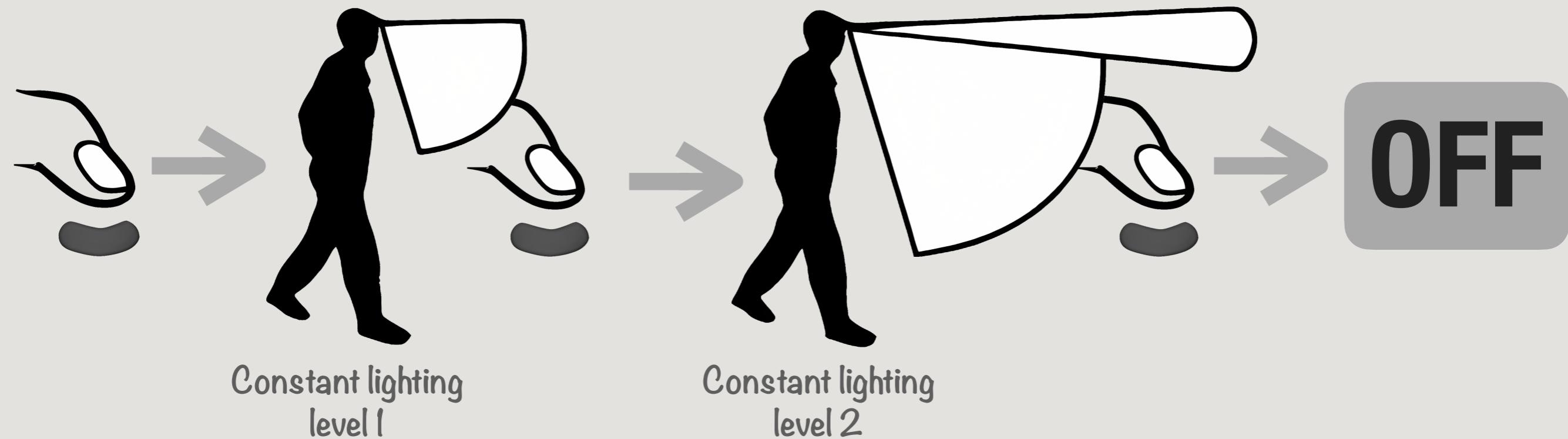
Este concepto —acciones de entrada y salida de los estados— se trata abundantemente en las máquinas de estados (como los diagramas de estado de UML), pero no se aborda explícitamente en el libro.

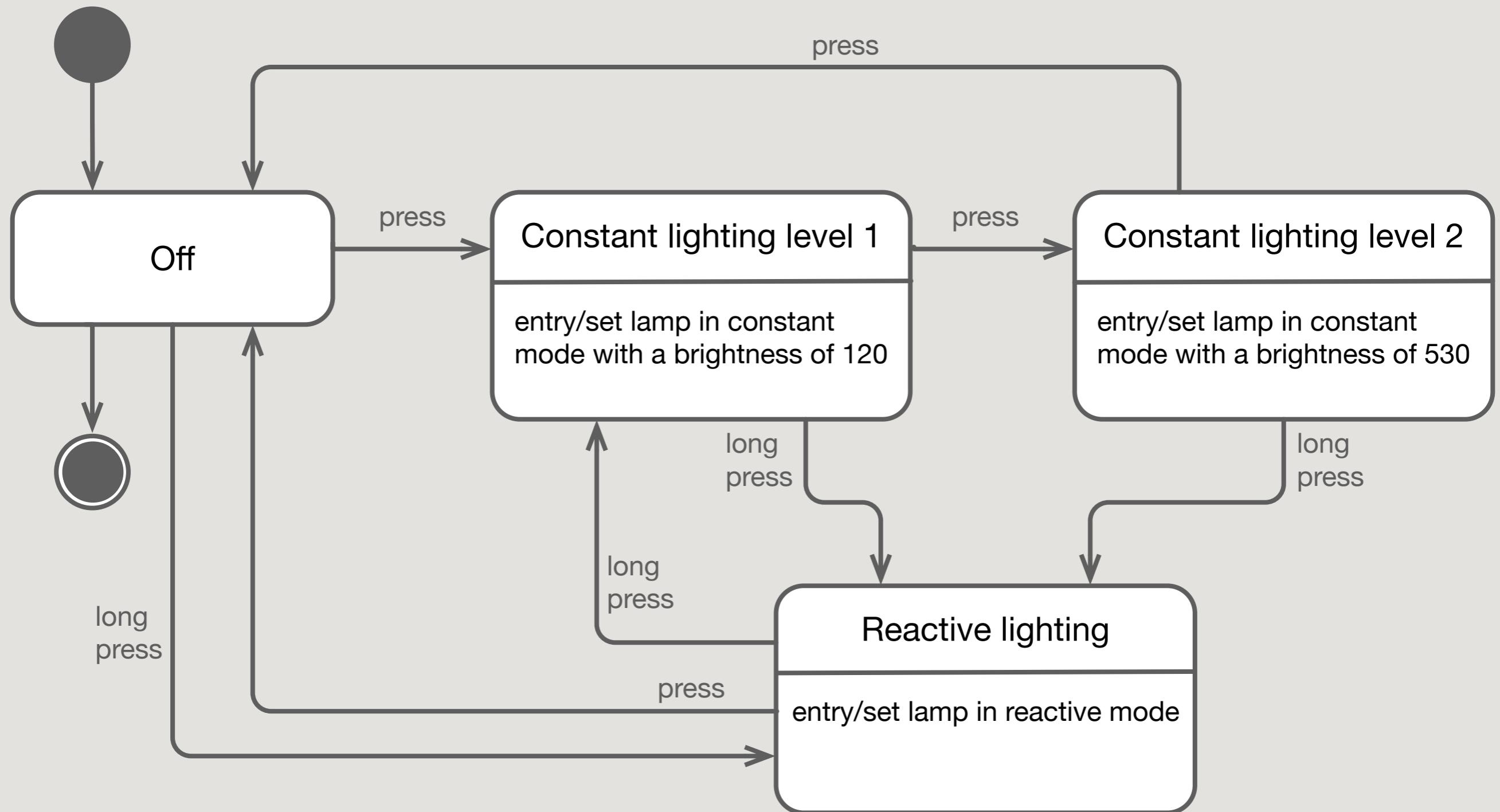
Así es como se podría llevar a cabo en la práctica:

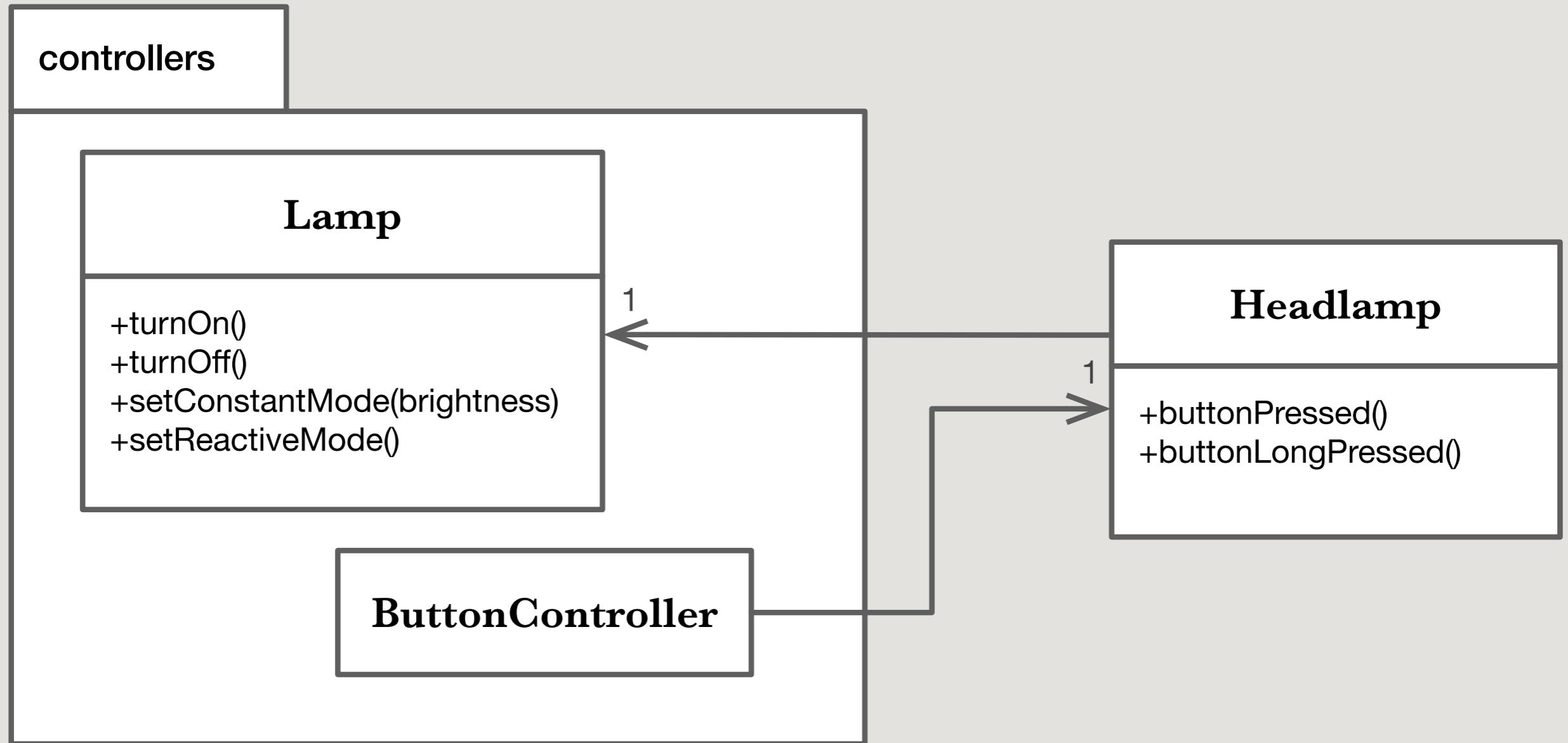
2 Acciones de salida: Por el contrario, las acciones de salida se ejecutan cuando el contexto **abandona** un estado. Estas acciones pueden encargarse de la limpieza, el registro guardar información específica del estado saliente.

Código de ejemplo



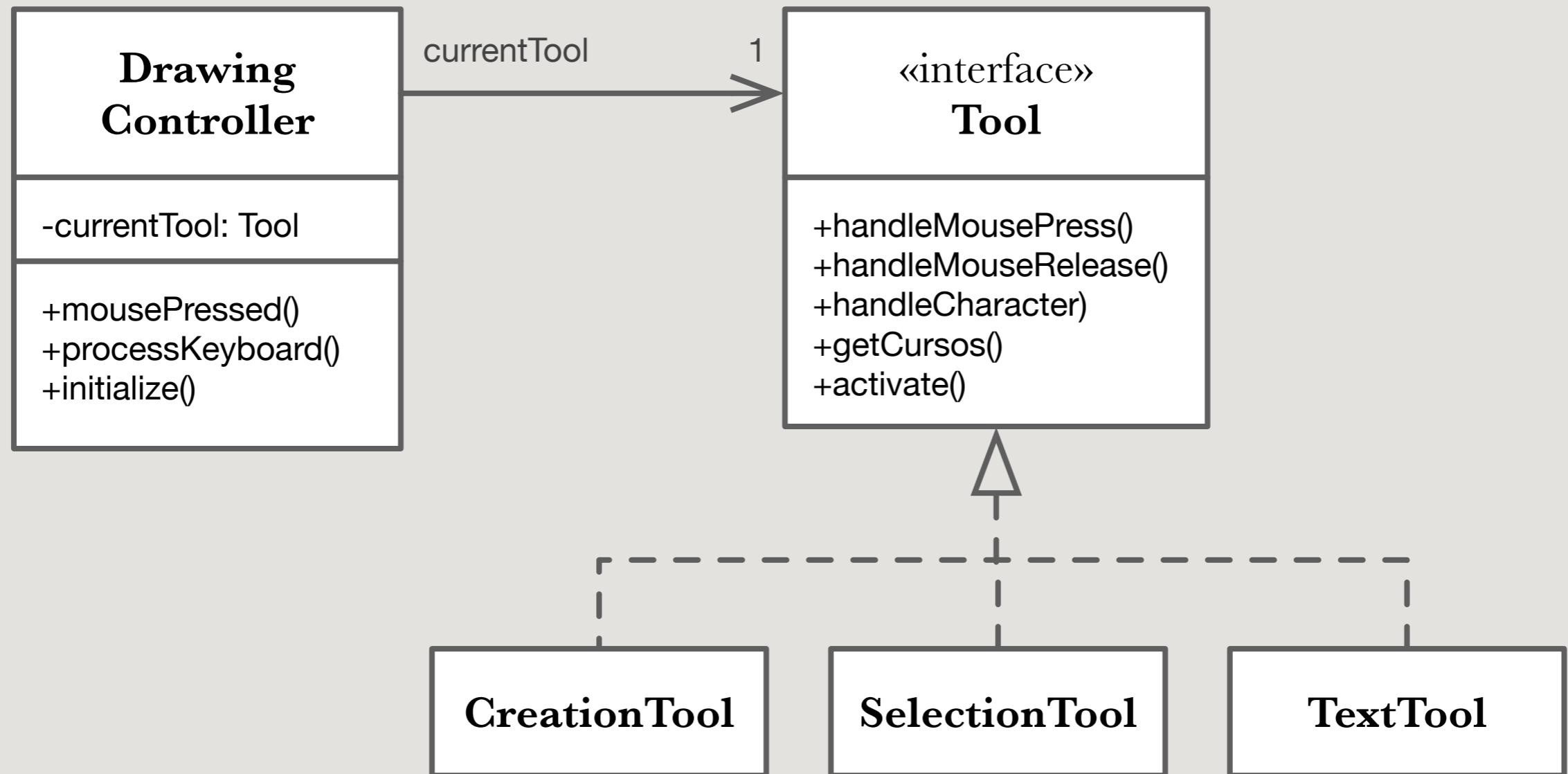






¿Cómo sería?

Usos conocidos



Frameworks de editores gráficos HotDraw y Unidraw

Patrones relacionados

Singleton

A veces los estados pueden ser singletons.

State y Strategy

Aunque esto no se menciona en el libro, es una pregunta recurrente.

¿En qué se diferencia
del patrón Strategy?



Strategy Pattern

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

When is it a [StrategyPattern](#) and when is it [StatePattern](#)? I have heard the quote "eventually everything looks like a Strategy" (can't remember the attribution), but I still can't see how a Strategy is distinct from a State.

Isn't it active (Strategy) vs. passive (State) ? For instance, you will compare the time, cost or distance estimates for different modes of travel (on foot, by metro, by car, for instance) by passing your parameters to appropriate TravelStrategy objects; whereas a TravelMode could use the [StatePattern](#) to say of a given Traveler object that she prefers, at various times, to use her car, take the metro, or go on foot.

Strategy and State are pretty similar. Both have a "context", an original object that is split into pieces, which are the strategy or the state, and both organize those pieces into a class hierarchy so that instead of having a case statement to choose between the pieces, the context sends a message to the current piece, and the code is selected using polymorphism. However, when I am using them I tend to think differently.

A strategy is an algorithm. Strategies often have internal variables that record the state of the algorithm. At the end of the algorithm, the variables might record the result, but in general they are only meaningful during the execution of the algorithm. A strategy is either selected by an outside agent or by the context. A strategy tends to have a single "start" method, and it calls all the rest. There is a lot of cohesion between the methods of a strategy.

In contrast, a state usually has no variables. ("State has no state", is what I say.) A state usually selects the next state of its context. A state tends to have lots of unrelated methods, so there is little cohesion between the methods of a state.

Perhaps this is not enough reason to distinguish between the two patterns. But I find enough uses of pure State and Strategy that I don't mind those times

A strategy is an algorithm. [...] A strategy is either selected by an outside agent or by the context. A strategy tends to have a single “start” method, and it calls all the rest. There is a lot of cohesion between the methods of a strategy.

In contrast, [...] a state usually selects the next state of its context. A state tends to have lots of unrelated methods, so there is little cohesion between the methods of a state.

Perhaps this is not enough reason to distinguish between the two patterns. But I find enough uses of pure State and Strategy that I don't mind those times when the design ends up half one and half the other.

—Ralph Jhonson

En cuanto a su intención

Una estrategia representa **distintas formas** de hacer lo mismo. En el caso del patrón State, sin embargo, los diferentes estados pueden llevar a cabo **acciones muy diferentes** en respuesta a las mismas peticiones.

Número de métodos

Y cómo se relacionan entre sí

Es muy frecuente que la interfaz del Strategy tenga un único método, mientras que en el caso del State solemos tener varios métodos que, además, no tienen por qué estar relacionados entre sí.

(Más que por el hecho de que, por definición, todos ellos varían o pueden variar a la vez dependiendo del estado).

Signatura de los métodos

En el patrón State, sus métodos suelen corresponderse con esas mismas operaciones públicas en el contexto, y con la misma signatura.

Salvo por que quizás reciban al contexto como parámetro.

En el caso del Strategy, muchas veces se trata de una operación de más bajo nivel necesaria para llevar a cabo alguna de las responsabilidades de la clase.

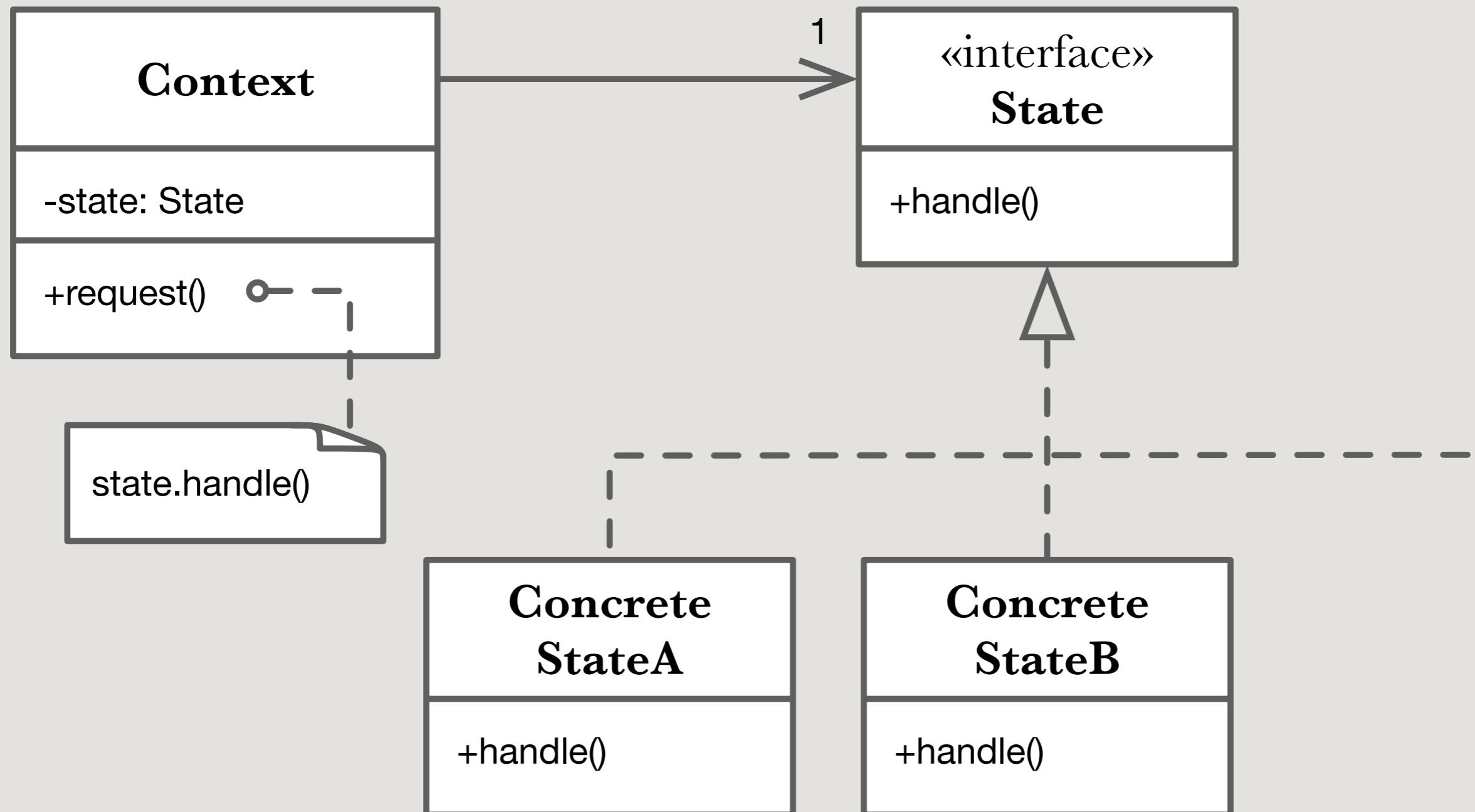
Cambios de estado

El patrón State involucra algún tipo de transición entre los estados.

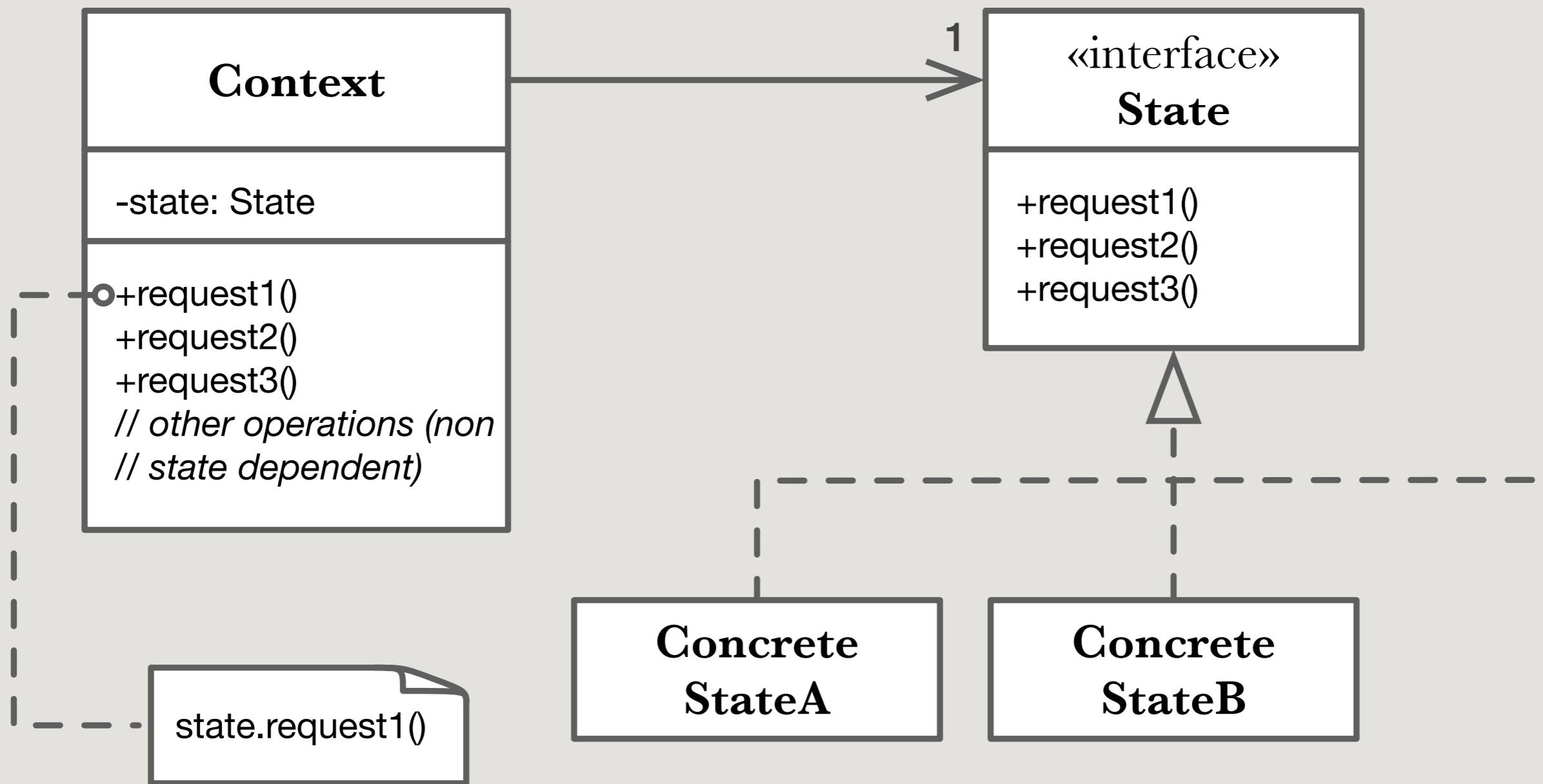
Es más, suelen ser los estados concretos los responsables de dichos cambios de estado, mientras que las estrategias son, por definición, totalmente independientes unas de otras.

Según las diferencias señaladas hasta ahora

Una estructura
mejorada del patrón



La estructura original del patrón, tal como aparece en el libro



Lo que refleja mejor la estructura típica del patrón State

Y lo que probablemente sea su diferencia más significativa

Y lo que probablemente sea su diferencia más significativa

El patrón State debería ser transparente para el cliente.

Por el contrario, el patrón Strategy, por definición, requiere que los clientes lo configuren y a menudo lo cambien en tiempo de ejecución.

