POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

24. *State*

Luis Fernández Muñoz
https://www.linkedin.com/in/luisfernandezmunyoz
setillofm@gmail.com

INDICE

1. Problema

- *La explicación del comportamiento de una persona depende de su estado de ánimo (p.e. cansado, bloqueado, proactivo, …) en cada una de las tareas a comprender (p.e. dormir, estudiar, trabajar, …). Esto complica la comprensión de su comportamiento. Según surgen nuevos estados de ánimo (p.e. enamorado, enfermo, …) se complica cada vez más la comprensión de cada tarea en cada estado.*
- *Una alternativa más sencilla sería la explicación por separado de todas las tareas para cada estado de ánimo particular. Con el advenimiento de nuevos estados no se complica ninguna explicación para estados anteriores o futuros.*

- *Permite a un objeto alterar su comportamiento cuando su estado interno cambia pareciendo que el objeto cambia su clase*

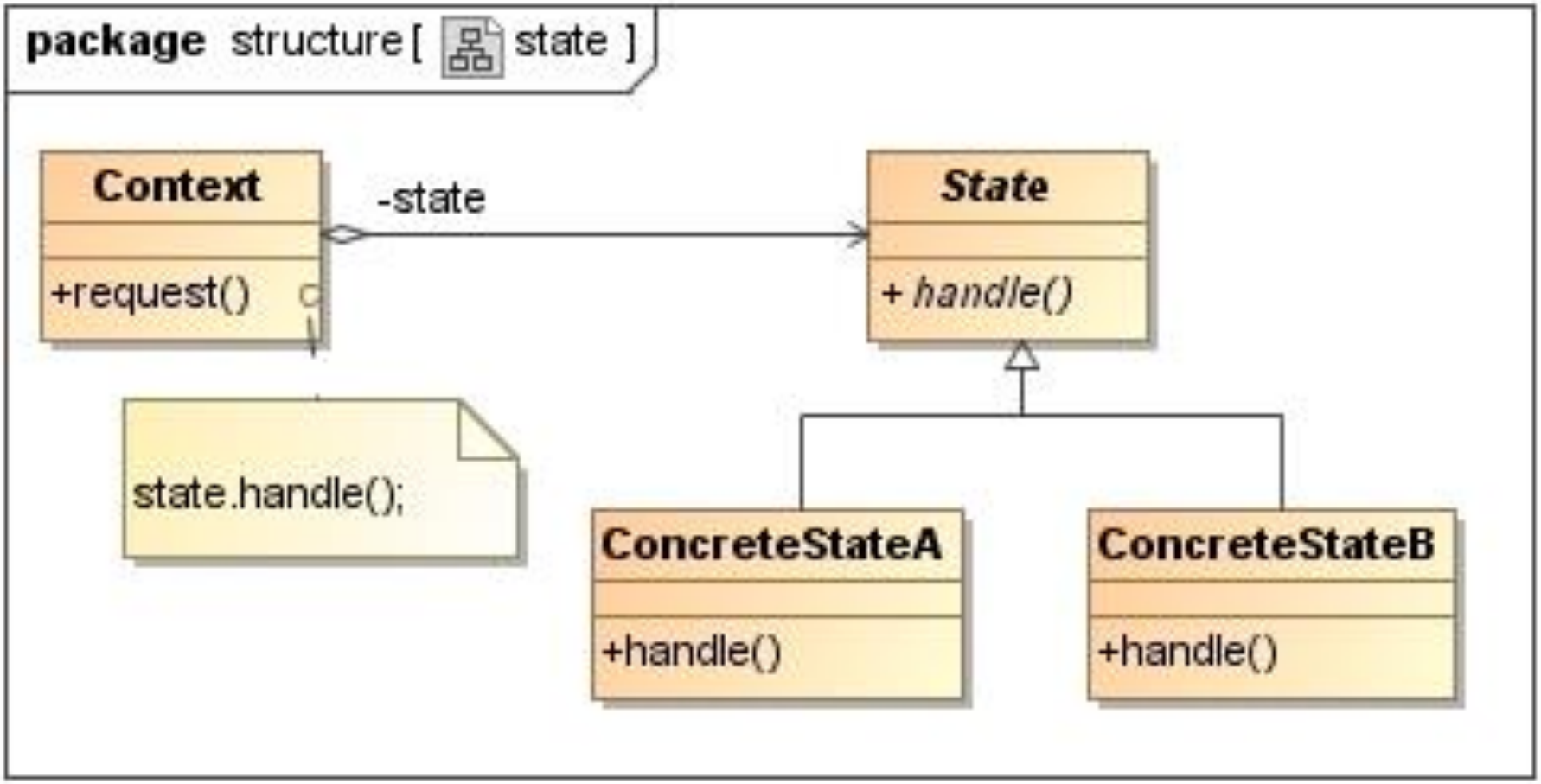
1. Problema

- Usa valores de datos para definir los estados internos y hacer que las operaciones de un contexto comprueben dichos estados explícitamente. Pero en ese caso tendríamos sentencias condicionales repartidas por toda la implementación de contexto. Añadir un nuevo estado podría requerir cambiar varias operaciones, complicando el mantenimiento.
- Cuando un objeto define su estado actual únicamente en términos de valores de datos internos, sus transiciones entre estados carecen de una representación explícita; solo aparecen como asignaciones a determinadas variables.

1. Problema

- Úsese el patrón estado en cualquiera de los dos siguientes casos:
  - El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado
  - Las operaciones tiene largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Este estado se suele representar por una o más constantes enumeradas. Muchas veces son varias las operaciones que contienen esta misma estructura condicional. El patrón *State* pone cada rama de la condición en una clase aparte. Esto nos permite tratar al estado del objeto como un objeto de pleno derecho que puede variar independientemente de otros objetos.

2. Solución


package structure [state]
Context
+request()
-state
State
+ handle()
state.handle();
ConcreteStateA
+handle()
ConcreteStateB
+handle()

2. Solución

- El patrón *State* no especifica qué participante define los criterios para las transiciones entre estados. Si estos criterios son fijos, entonces pueden implementarse enteramente en el Contexto. No obstante, es generalmente más flexible y conveniente que sean las propias subclases de Estado quienes especifiquen su estado sucesor y cuándo llevar a cabo la transición. Esto requiere añadir una interfaz al Contexto que permita a los objetos Estado asignar explícitamente el estado actual del Contexto.
  - Descentralizar de esta forma la lógica de la transición facilita modificar o extender dicha lógica definiendo nuevas subclases de Estado. Una desventaja de la descentralización es que una subclase de Estado conocerá al menos a otra, lo que introduce dependencias de implementación entre subclases

2. Solución

- Una alternativa basada en tablas que hagan corresponder entradas con transiciones de estado. Para cada estado, una tabla hace corresponder cada posible entrada con un estado sucesor. Este enfoque convierte código condicional (y funciones virtuales, en el caso del patrón *State*) en una tablas de búsqueda.
  - La principal ventaja de las tablas es su regularidad: se pueden cambiar los criterios de transición modificando datos en vez del código del programa. Hay, no obstante, algunos inconvenientes:
    - Es menos eficiente que una llamada a una función
    - Los criterios de transición son menos explícitos
    - Debe ser aumentado para realizar algún tipo de procesamiento arbitrario con cada transición

2. Solución

- Una cuestión de implementación que hay que ponderar es si crear los objetos Estado sólo cuando se necesitan y destruirlos después o si crearlos al principio y no destruirlos nunca.
  - La primera elección es preferible cuando no se conocen los estado en tiempo de ejecución y los contextos cambia de estado con poca frecuencia. Este enfoque evita crear objetos que no se usarán nunca, lo que puede ser importante si los objetos Estado guardan una gran cantidad de información.
  - El segundo enfoque es mejor cuando los cambios tienen lugar rápidamente es cuyo caso querremos evitar destruir los estado, ya que pueden volver a necesitarse de nuevo en breve. Los costes de creación se pagan al principio y no existen costes de destrucción. No obstante, este enfoque puede no ser adecuado ya que el Contexto debe guardar referencias a todos los estados en los que pueda entrar

2. Solución

- Cambiar el comportamiento de una determinada petición podría lograrse cambiando la clase del objeto en tiempo de ejecución, pero esto no es posible en la mayor parte de los lenguajes de programación orientados a objetos. Las excepciones incluyen Self y otros lenguajes.

3. Consecuencias

- El patrón *State* sitúa en un objeto todo el comportamiento asociado con un determinado estado. Como todo el código dependiente del estado reside en una subclase de Estado, pueden añadirse fácilmente nuevos estados y transiciones definiendo nuevas subclases.
  - Esto incrementa el número de clases y es menos compacto que un única clase. Pero dicha distribución es realmente buena si hay muchos estados, que de otro modo necesitarían grandes sentencias condicionales.
  - La lógica que determina las transiciones entre estados no reside en sentencias *if* o *switch* monolíticas, sino que se reparte entre la subclases del Estado. Al encapsular cada transición y acción en una clase estamos elevando la idea de un estado de ejecución a objetos de estado en toda regla. Esto impone una estructura al código y hace que su intención sea más clara

3. Consecuencias

- Introducir objetos separados para los diferentes estados hace que las transiciones sean más explícitas. Además, los objetos Estado pueden proteger la Contexto frente a estados internos inconsistentes, ya que la transiciones entre estados son atómicas desde una perspectiva del Contexto – tienen lugar cambiando una variable (el objeto variable del Contexto, Estado), no varias.
- En caso de que los objetos Estado no tengan variables – es decir, si el estado que representan está totalmente representado por su tipo – entonces varios contextos pueden compartir un mismo objeto Estado. Cuando se comparten los estados de este modo, son en esencia pesos ligeros que no tienen estado intrínseco, sino solo comportamiento.

4. Relaciones

- ***Singleton*** para los objetos *State*
- ***Flyweight*** para compartir los objetos *State*.