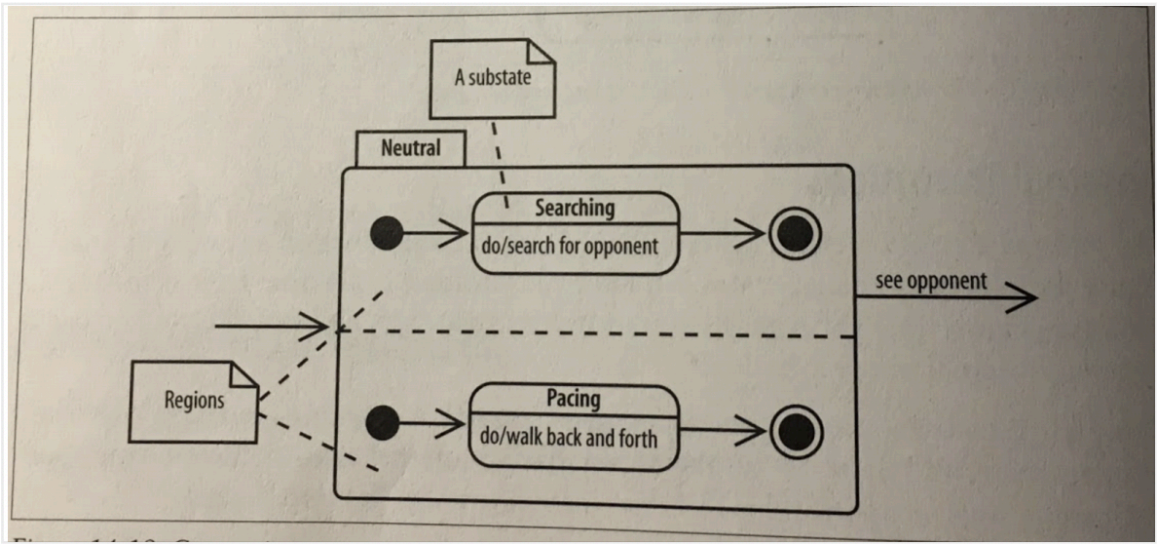
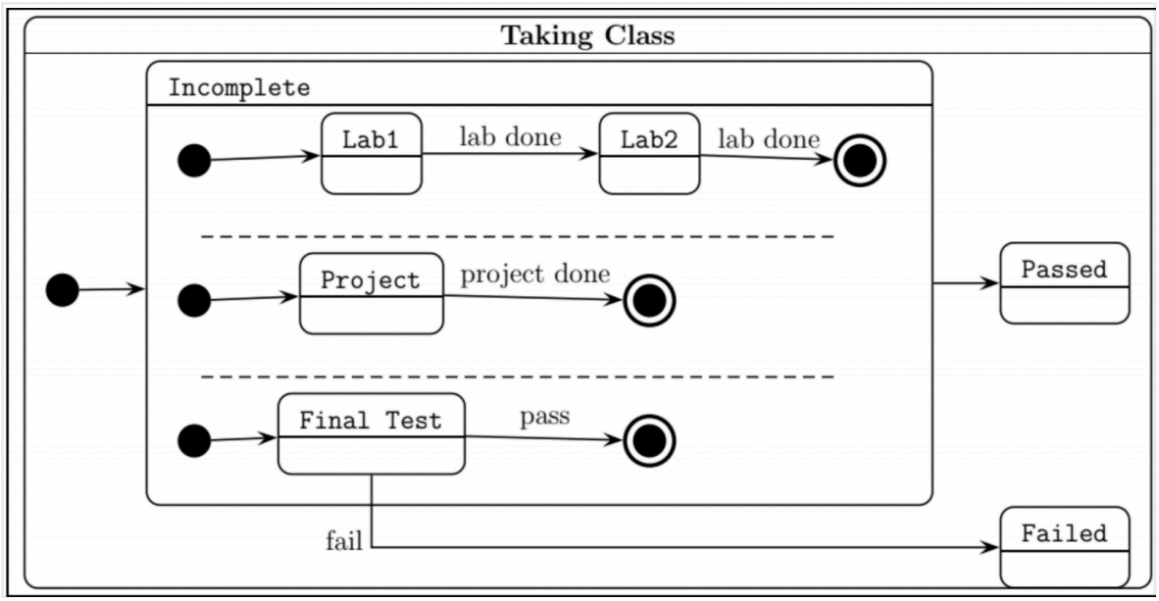
Clase 15 - DA2 Teórico
22/11/2021

Estados paralelos o estados concurrentes:

- Los podemos implementar a través del elemento UML denominado “regiones ortogonales”.
- Las regiones ortogonales son las que se definen con la línea punteada separando las diferentes máquinas embebidas que se ejecutan en paralelo
- Los estados simples (los que no son compuestos): no tienen regiones ortogonales.
- Los estados compuestos: contienen al menos una región ortogonal.
  - Estado compuesto simple: contiene exactamente una región ortogonal (todos los ejemplos anteriores que vimos de máquinas con estados compuestos)
  - Estado compuesto ortogonal: contiene al menos dos regiones ortogonales (es el escenario donde modelamos concurrencia)
    - Cada región, va a tener su propio estado de inicio y su propio estado de fin

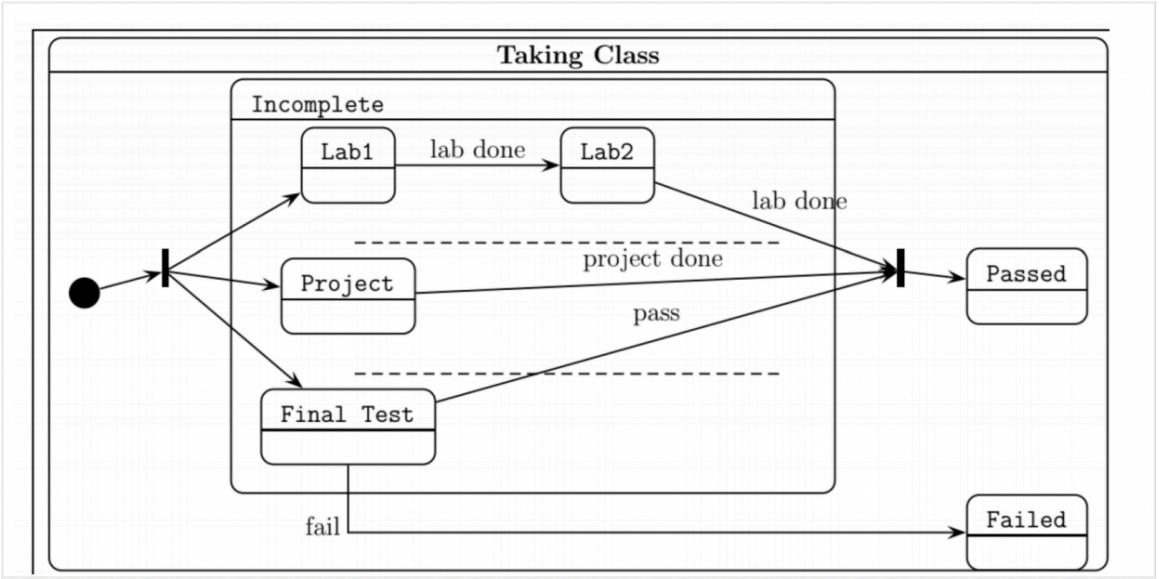




En el escenario de esta “carrera” el estudiante tiene que hacer 3 cosas en paralelo para poder aprobar:

- Dos laboratorios (donde uno es consecuencia del otro)
- Un proyecto
- Un test final

Cuando cada región o cada maquina paralela termine, ahi es cuando se aprueba el curso.
Y también puede pasar, que sin importar lo que pasa en las otras regiones, si el estudiante pierde el test final, se sale del estado compuesto y se pierde el curso/carrera.



Estas dos maquinas son equivalentes, con la diferencia que la

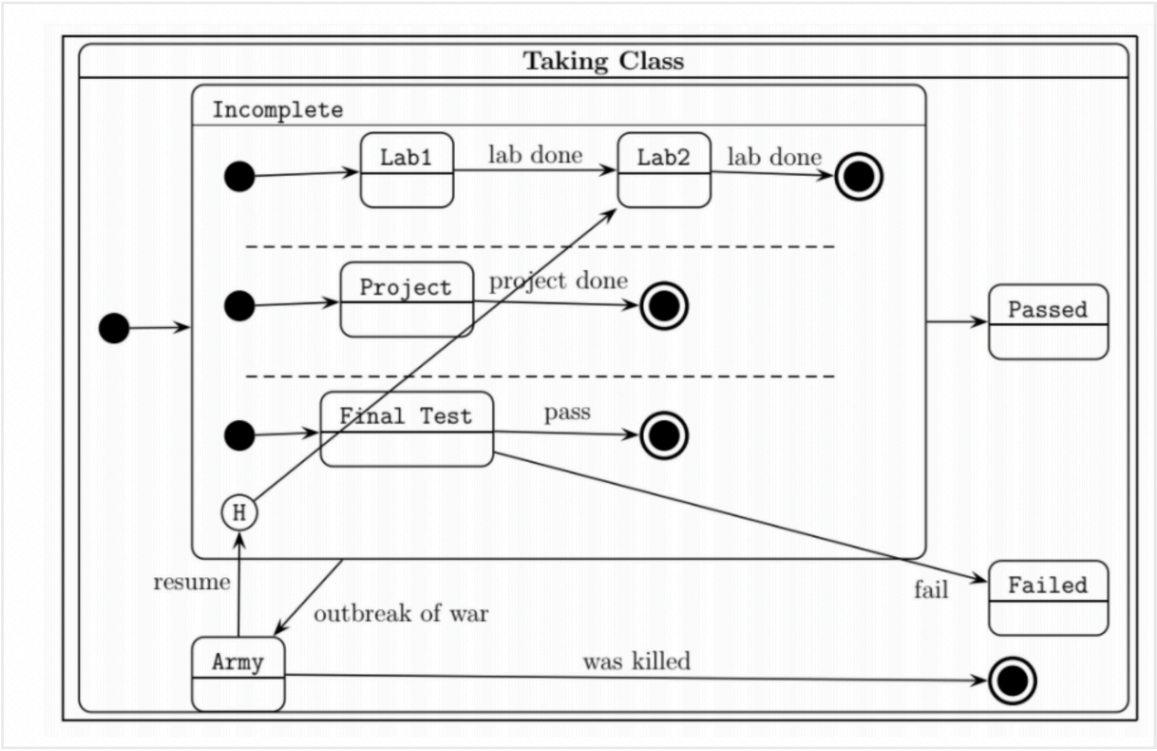
segunda utiliza la notación “Fork y Join”, con el fin de no tener que definir un estado de inicio y uno de fin en cada una de las 3 regiones ortogonales.

Historia en diagramas de estado:

Es un elemento dentro de la notación UML de diagramas de estado que nos permite “marcar a recordar” el último subastado activo antes de salir del estado compuesto.

Existen dos tipos de historia:

- Shallow (H): graba o recuerda el ultimo subastado activo solo del estado compuesto al cual la historia pertenece. Recuerda solo 1 nivel de profundidad.
- Deep (H*): recuerda el último subastado activo dentro del estado compuesto al cual la historia pertenece, aplicando a todos los niveles de profundidad de dicho subastado.

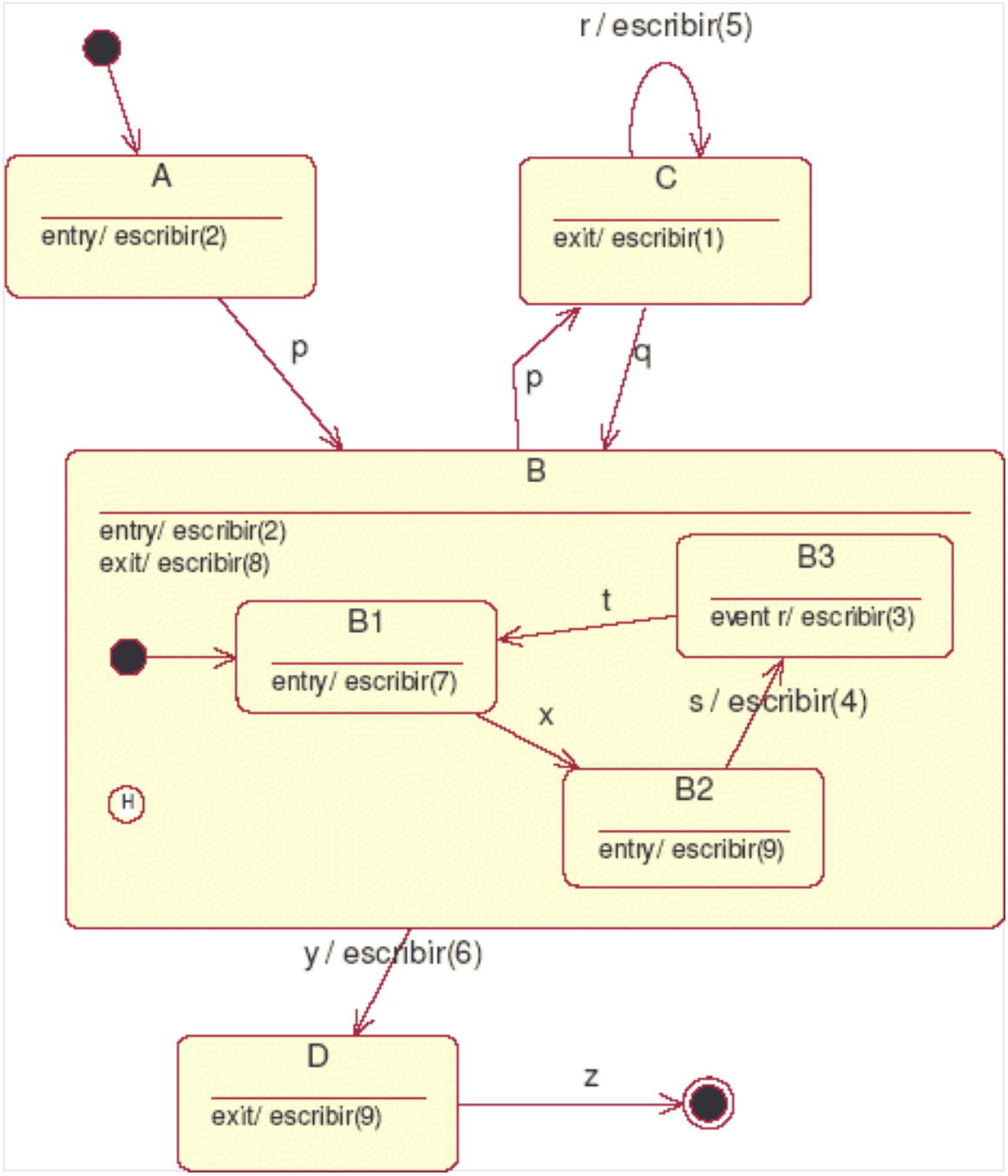


Ejercicio:

Identifique la salida escrita por la máquina de estados que recibe los siguientes eventos:

p x p q s t p y z r q y z

(Asuma que la máquina no se tranca al procesar eventos que no corresponden)



p x p q s t p y z r q y z

Evento	Estado	Salida
-	A	2
p	B	2
-	B1	7
x	B2	9
p	C	8
q	B	1 - 2
-	B2	9
s	B3	4
t	B1	7
p	C	8
y	- (me quedo en C)	(No escribo nada)
z	- (me quedo en C)	(No escribo nada)
r	C	1 - 5
q	B	1 - 2
-	B1	7
y	D	8 - 6
z	Fin	9

2 - 2 - 7 - 9 - 8 - 1 - 2.....

Implementación por códigos numéricos:

La basamos en el ejemplo del Molinete (Turnstile)

Vamos a tener 3 clases principales:

- **Program:** donde tenemos el punto de entrada del programa, donde terminamos ejecutando la máquina, tirándole una secuencia de eventos para que los procese.
- **Turnstile:** es donde modelamos la máquina en sí. En otras palabras, es donde mantenemos el estado actual y donde definimos la lógica para procesar un nuevo evento, cambiar de estado y donde disparamos las acciones asociadas a cada transición. Tiene un método público ProcessEvent que será invocado desde afuera (en este caso dede la clase Program)

y es el que permite decirle a la máquina “Procesadme este nuevo evento”.
- **TurnstileController:** es una clase que será usada por el Turnstile (se le inyecta al turnstile) y que tiene definida toda la lógica de cada una de las acciones disparadas por Turnstile (tiene un método por cada acción a ejecutar en cada transición) Vamos a tener 4 métodos (1 para Sonar Alarma, 1 para Destrancar, 1 para Muchas Gracias y 1 para Trancar)

Analizando esta solución (ventajas vs desventajas):
- Se podría definir una interfaz para el TurnstileController, de manera que podamos eventualmente reemplazar de forma sencilla las implementaciones de las acciones dentro del Controller.
- Si quisiéramos extender la maquina **agregando nuevos estados,** deberíamos agregar nuevas ramas “case” al switch dentro de ProcessEvent. Esta implementación viola OCP porque no es abierta a la extensión.
- Si quisiéramos extender la máquina **agregando nuevos eventos y/o transiciones**, deberíamos agregar la capacidad de procesar dichos eventos dentro de cada uno de los switch asociados a cada estado. Esto también viola OCP.
- Esta implementación es compleja de mantener y/o usar cuando tenemos muchos estados, dado que vamos a tener tantos flujos de ejecución como estados (N) y eventos (M) tenga. En otras palabras, vamos a tener NxM condiciones o flujos posibles de ejecución, y si los valores de N y M son altos, la maquina es muy compleja y llena de condicionales. No es lo que queremos.

Implementación mediante el patrón State:

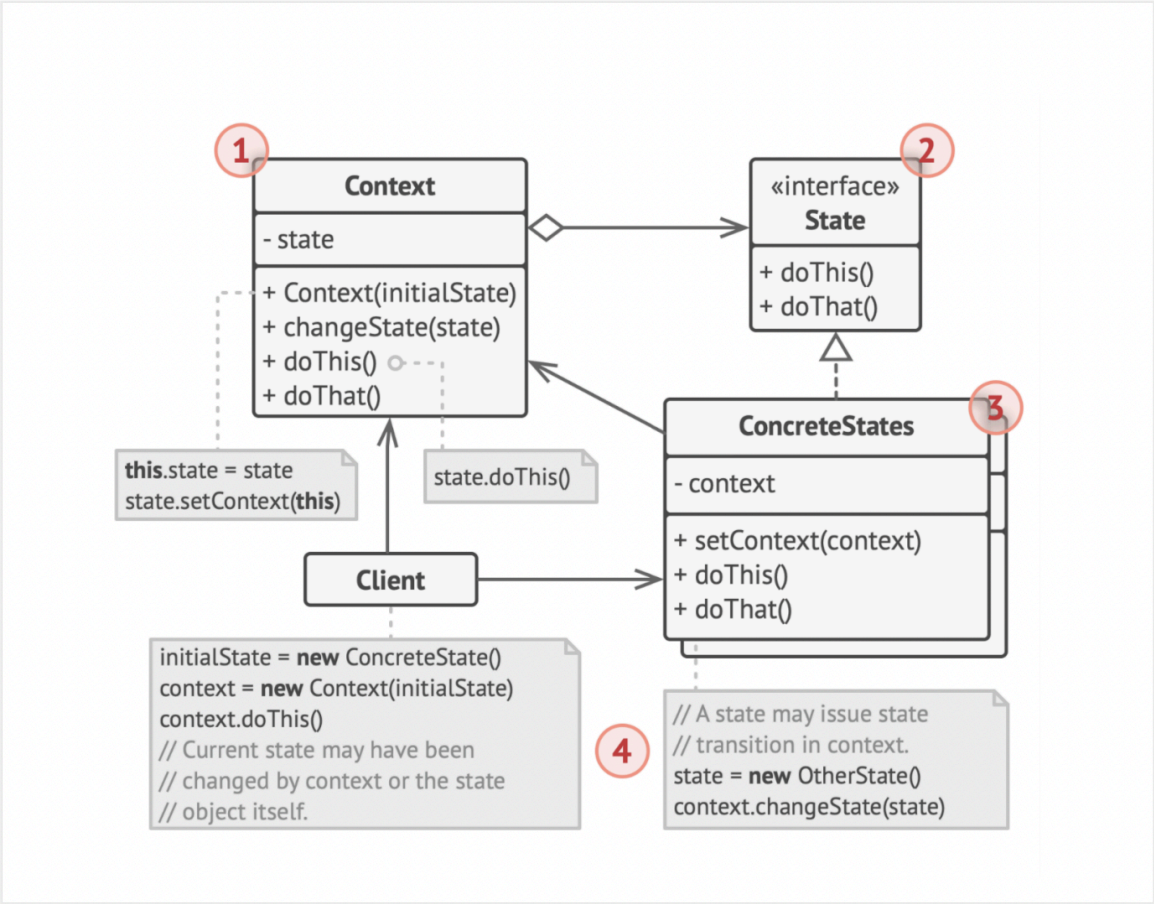
Clasificación: Comportamiento

Motivación:
Permitir que un objeto cambie su comportamiento cuando cambia su estado. En otras palabras, permite implementar una máquina de estados.

Aplicación:

Cuando tenemos cierto comportamiento que depende del estado y este cambia en tiempo de ejecución. En ciertas operaciones que son muy complejas y que basan decisiones/condiciones con posibles valores predeterminados y que a su vez dichas condiciones se evalúan en diferentes lados para determinar el comportamiento final. - (Varios ifs/switches o condiciones boleadas que cambian el comportamiento)

Estructura genérica:



1. **Clase Context:** tiene una referencia a uno de los estados concretos y le delega la responsabilidad de resolver aquellas operaciones que sean específicas del estado. En este caso, dichas operaciones son doThis() y doThat(). El Context se comunica con el estado concreto a través de la interfaz State. El contexto expone un setter (changeState()) para que se le pueda pasar un nuevo objeto estado.
2. **Interfaz State:** es una interfaz donde declaramos todos los métodos que son específicos al estado (en otras palabras, declaramos los eventos). Vamos a tener tantos métodos en la

interfaz State, como eventos tenga para procesar. Estos métodos deben tener sentido para todos o la mayoría de los estados concretos porque no querríamos tener estados concretos que tengan métodos que no se usen o que no apliquen.
3. **Una o varias clases ConcreteState:** son las implementaciones concretas de cada estado, brindando la forma de procesar cada uno de los eventos para ese estado en concreto. En caso de que tengamos código repetido entre estados concretos podríamos llegar a proveer una clase abstracta intermediaria que encapsule ese comportamiento común (similar a lo que hicimos con el Controller en la solución anterior). Puede llegar a suceder que los estados concretos mantengan una referencia al objeto Context, a través del cual pueden obtener información que sea útil así como también disparar transiciones.
4. **Comportamiento:** tanto el contexto como los estados concretos puede sestear el nuevo estado del contexto y llevar a cabo la transición del estado reemplazando el estado actual que el contexto tiene.

Detalle de implementación:

¿Quién realiza el cambio de estado? Es decir, ¿quién define las transiciones?

El patrón en sí no especifica quien es el responsable de cambiar de estado. Esto puede ser llevado a cabo por cada estado concreto o en el contexto.

- Si las transiciones son fijas, puede hacerse en el contexto
- La opción mas flexible es que las transiciones sean disparadas por los propios estados, requiriendo que haya un método SetState() o ChangeState() público en el contexto. Y a su vez, se requiere que cada estado concreto se conozca entre sí (o al menos a aquellos estados que se ven involucrados en una misma transición).
- La última opción sería definir y cargar los estados de destino en algún lugar como una BD. Esto puede llegar a ser complejo cuando hay transiciones a diferentes estados desde un

mismo estado de origen (esta lógica puede llegar a ser difícil de hacer sin tener que preguntar por el estado destino).

¿Cuándo instanciamos los estados?

- Instanciamos cada vez que se precisen (esto tiene sentido cuando no conocemos los estados y estos cambian poco)
- Crear todos los estados de una en la creación del contexto (esto tiene sentido cuando tenemos muchos estados y/o estos cambian rápidamente). Esto trae como consecuencia que el contexto deba conocer a cada estado hijo (los subastados)

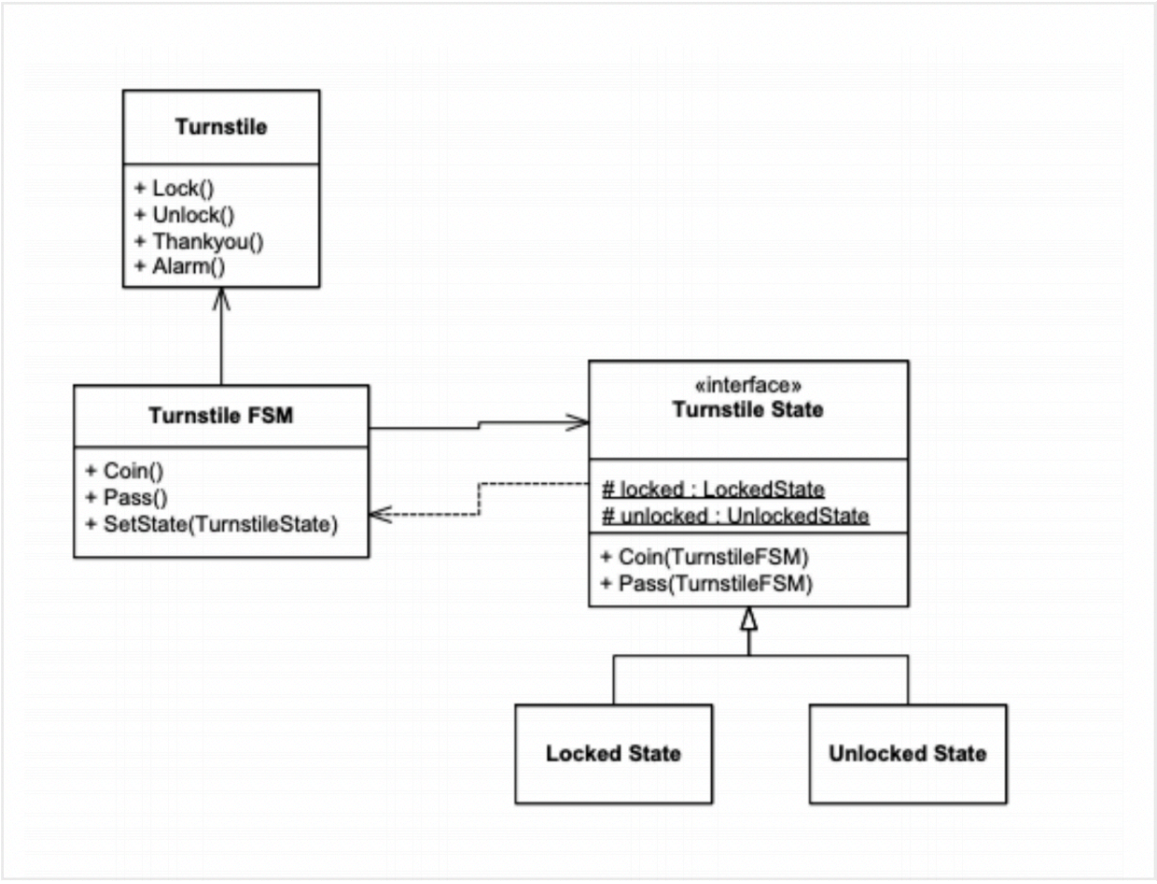
Ventajas:

1. Es más extensible ante hacer if/else o switch/case a lo largo de todo el código del contexto
2. Centraliza todos los posibles comportamientos de los distintos estados en un único lugar. Encapsula todos esos comportamientos en cada subastado.
3. Es fácil de agregar un nuevo estado, simplemente heredo del State abstracto o de la interfaz State
4. La transición entre estados es explícita y atómica (no es más que una operación/método) que es visible desde el punto de vista del contexto.

Desventajas:

1. Aumenta el número de clases (lo que antes teníamos en una sola clase ahora lo tenemos en por lo menos cuatro). La solución es más compleja y menos compacta que si tuviéramos una clase sola. De todas maneras, esa clase sola hubiera estado llena de opciones case/if-else lo cual tampoco es deseable.

Ejemplo de molinete:
https://github.com/ORT-DA2/Teorico-Implementacion-Maquinas-de-Estado


Turnstile
+ Lock()
+ Unlock()
+ Thankyou()
+ Alarm()
Turnstile FSM
+ Coin()
+ Pass()
+ SetState(TurnstileState)
«interface»
Turnstile State
locked : LockedState
unlocked : UnlockedState
+ Coin(TurnstileFSM)
+ Pass(TurnstileFSM)
Locked State
Unlocked State