Reificando el estado

Gastón Prieto

Introducción

En este apunte vamos a ver como reificar el estado, esto significa que según en qué estado esté su comportamiento va a variar y cada estado va a ser responsable de cómo transicionar a otros estados en base a criterios definidos por cada uno.

El caso del Tamagotchi

Nos proponen hacer el siguiente sistema:

Modelar una mascota virtual, del estilo Tamagotchi, de manera que yo pueda usarla para que coma, juegue y saber si tiene ganas de jugar.

También hay que poder conocer qué tan contenta está la mascota, que es un número entero mayor o igual que 0, donde a mayor nivel, más contenta está la mascota. Además una mascota puede estar mal humor, hambrienta o contenta; y su comportamiento depende de en qué estado esté.

Cuando una mascota come, pasa lo siguiente:

- Si está hambrienta, se pone contenta.
- Si está contenta, su nivel se incrementa en una unidad.
- Si está de mal humor, y hace más de 80 minutos que está de mal humor, entonces se pone contenta.
- Si está de mal humor desde hace 80 minutos o menos, entonces no cambia nada.

Cuando una mascota juega, pasa lo siguiente:

- Si está contenta, su nivel se incrementa en dos unidades. Si llegará a jugar mas de 5 veces se pone hambrienta (jugar da hambre).
- Si está de mal humor, se pone contenta.
- Si está hambrienta se pone de mal humor.

Y por último una mascota tiene ganas de jugar si está contenta o aburrida, si está hambrienta no.

Para realizar esta solución vamos a ir atacando el problema de a poco, utilizando un enfoque iterativo/incremental.

Iteración 1

> *Modelar una mascota virtual, del estilo Tamagotchi, de manera que yo pueda **usarla** para que **coma, juegue** y saber si tiene **ganas de jugar**.*

Acá nos están comentando operaciones que sabe hacer una mascota, entonces podemos pensar que la interfaz de la mascota podría entender los mensajes *comer(), jugar() y conGanasDeJugar()*. Pero como no sabemos cómo se implementa, no vamos a ponerlos en nuestra clase, aunque si nos anotamos en otro lado que es posible que aparezca.

//TODO[1]
- [Req[2]] - Mascota sabe jugar. *mascota.jugar() ?*
- [Req] - Mascota sabe comer. *mascota.comer() ?*
- [Req] - Mascota sabe decirte si tiene ganas de jugar. *mascota.conGanasDeJugar() ?*

> *También hay que poder conocer qué tan contenta está la mascota, que es un número entero mayor o igual que 0, donde a mayor nivel, más contenta está la mascota*

Como nos dicen que la mascota tiene un nivel de contenta el cual va variando, entonces podríamos ponerle un atributo *nivelContenta*.

> *Además una mascota puede estar mal humor, hambrienta o contenta;*

Como son 3 valores bien conocidos, vamos a utilizar enums para modelarlos. Así nos evitamos problemas de inconsistencias que podrían pasar si usamos *Strings*.

El diagrama de clases que nos queda al momento es:

[1] Se dice TODO List a una lista de cosas pendientes por hacer, en general muchos IDE's soportan comentarios que arrancan con //TODO o //FIXME y te los muestran en una solapa especial. También se puede usar un anotador para esto.
[2] Req = Requerimiento

Iteración 2

Para esta iteración vamos a agarrar el siguiente requerimiento:

> *Cuando una mascota come, pasa lo siguiente:*
>
> - *Si está hambrienta, se pone contenta.*
> - *Si está contenta, su nivel se incrementa en una unidad.*
> - *Si está de mal humor, y hace más de 80 minutos que está de mal humor, entonces se pone contenta.*
> - *Si está de mal humor desde hace 80 minutos o menos, entonces no le pasa nada, no cambia nada.*

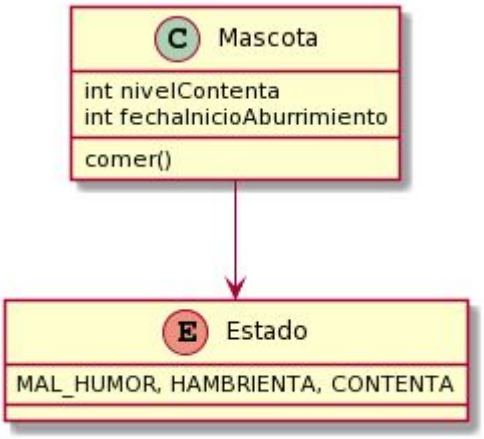
Como vimos anteriormente uno de los mensajes que podía entender la mascota era *comer()*, así que vamos a modelarlo.

Tenemos varios caminos:

- Un **if** según el estado del enum.
- ~~**Herencia**~~, descartada porque la mascota cambia de estado dinámicamente.
- Usar **composición**

Vamos a ir por la del condicional y analizamos cómo queda:

```
clase Mascota
  metodo comer()
    if estado == Estado.CONTENTA
      this.nivelContenta += 1
    else if estado == Estado.HAMBRIENTA
      this.estado = Estado.CONTENTA
    else // Mal Humor
      if minutosMalHumor() > 80
         this.estado = Estado.CONTENTA
```



Vamos a hacer un análisis de esta solución:

- Vemos que hay algunos atributos que solo se usan en ciertos estados de la mascota, por ejemplo los minutos de mal humor y el nivel de contenta. Por lo tanto hay que tener mucho cuidado de mantener estos estados de forma consistente.

- Adelantandonos un poco a lo que sabemos que hay que hacer, este if seguro se va a repetir en *jugar()*, entonces esto nos da una idea de que necesitamos flexibilidad a la hora de soportar nuevas operaciones en los estados.

Vayamos por el camino de composición y delegarle al estado el comportamiento de saber cómo se juega. Para esto seguramente el estado entienda un mensaje *comer()*.

Arrancamos primero por el estado contenta, analizamos si podemos seguir dejandolo como enum

- Al enum le podemos agregar comportamiento. ✔️
- Cuando la mascota *come* aumenta su nivel de contenta, entonces tenemos dos opciones
    - Dejar el nivel de contenta en la mascota.
    - Poner el nivel de contenta en el estado.

La pregunta que tenemos que hacernos es si cuando deja de estar contenta se pierde ese nivel o es algo que cuando vuelva a estar contenta lo va a recuperar.

Fuimos a preguntarle y tuvimos esta conversación:

- ***Analista**: "Cada vez que vuelve a estar contenta arranca de 0, porque eso posteriormente lo vamos a usar para …."*
- ***Nosotros:** "ok, ok, pero mira que eso entra en otra iteración :)"*
- ***Analista:** "Obvio ;)"*

```
clase Mascota
   metodo comer()
     this.estado.comer()
```

```
clase Contenta
   metodo comer()
      this.nivel = this.nivel + 1
```

Seguimos con Hambrienta, y nos damos cuenta que no tenemos la mascota para setearle el nuevo estado.

```
clase Hambrienta
   metodo comer()
      ???.setEstado(new Contenta())
```

Entonces tenemos dos opciones,

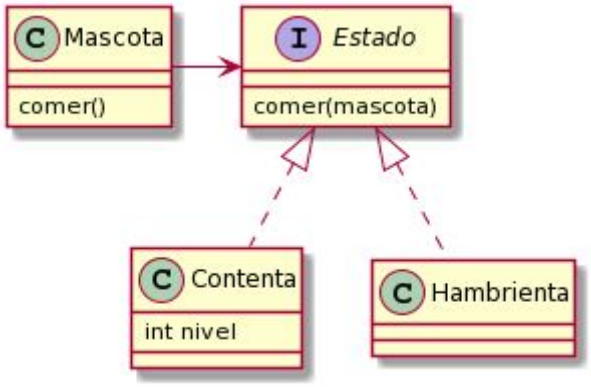
- Tenerlo como atributo en el estado.
- Recibirlo por parametro.

Por una cuestión de simplicidad en la instanciación, voy a elegir recibirlo por parametro, aunque esto rompa la firma original que había pensado para los estados.

```
clase Mascota
   metodo comer()
    estado.comer(this)
```

```
clase Contenta
   metodo comer(mascota)
      this.nivel = this.nivel + 1

clase Hambrienta
   Metodo comer(mascota)
      mascota.setEstado(new Contenta())
```

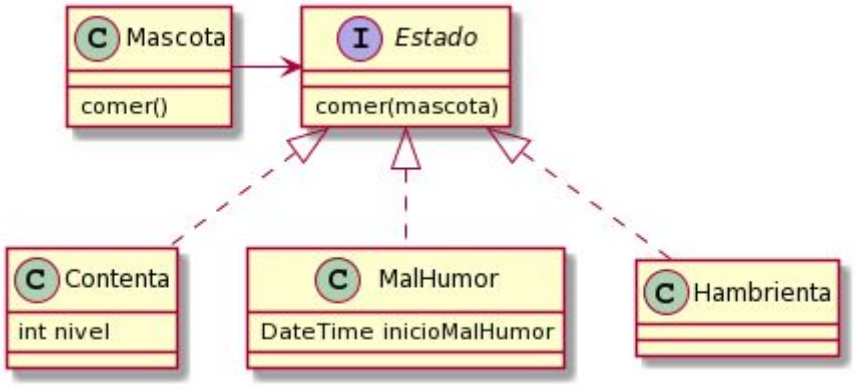


Completamos con el estado MalHumor

```
clase MalHumor
   metodo comer(mascota)
     If this.minutosMalHumor() > 80
           mascota.setEstado(new Contenta())

   metodo miutosMalHumor()
     retornar this.inicioMalHumor.diferenciaEnMinutos(Date.now())
```

Vemos que el impacto de agregar este nuevo estado es mínimo. Quedando la siguiente solución.



Analizando el diseño lo único raro que vemos es que setear a la mascota el estado contenta está repetido en dos de los estados, podríamos delegarle esta operación a la mascota. Pero

lo vamos a dejar para más adelante, anotando esto en nuestra lista de tareas pendientes.
Y nuestra lista de pendientes quedo:

//TODO

- [Req] - Mascota sabe jugar. *mascota.jugar() ?*
- ~~[Req] - Mascota sabe comer. *mascota.comer() ?*~~
- [Req] - Mascota sabe decirte si tiene ganas de jugar. *mascota.conGanasDeJugar() ?*
- [Ref[3]] - Mascota debería saber el mensaje, mascota.alegrarse()?

Iteración 3

Cuando una mascota juega, pasa lo siguiente

- *Si está contenta, su nivel se incrementa en dos unidades. Si llegará a jugar mas de 5 veces se pone hambrienta (jugar da hambre).*
- *Si está de mal humor, se pone contenta.*
- *Se pone de mal humor si esta hambrienta.*

Vamos a agregar esta operación nueva a la mascota y a los estados.

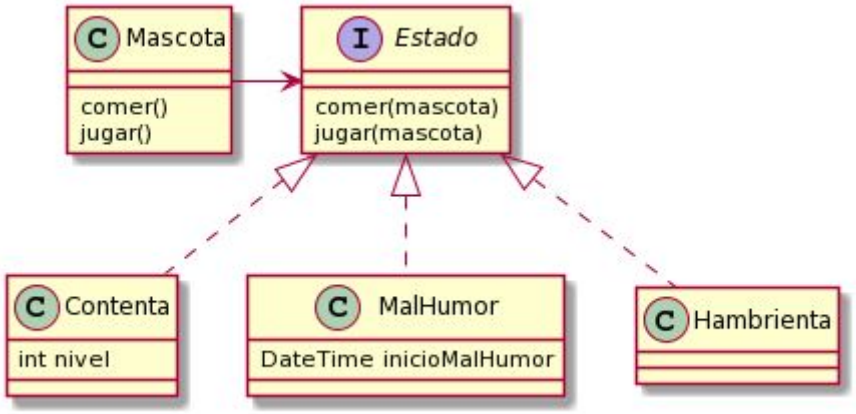
```
clase Mascota
   metodo jugar()
     estado.jugar(this)
```

```
clase Contenta
   metodo jugar(mascota)
     this.nivel += 2
     this.cantidadJuegos += 1
     if (this.cantidadJuegos == 5)
       mascota.setEstado(new Hambrienta())
```

```
clase Hambrienta
   metodo jugar(mascota)
      mascota.setEstado(new MalHumor())
```

```
clase MalHumor
   metodo jugar(mascota)
     mascota.setEstado(new Contenta())
```

[3] Refactor



Para el requerimiento de *conGanasDeJugar* es hacer exactamente lo mismo que antes, retornando un true o false según sea el caso.

Actualizamos nuestra lista de tareas

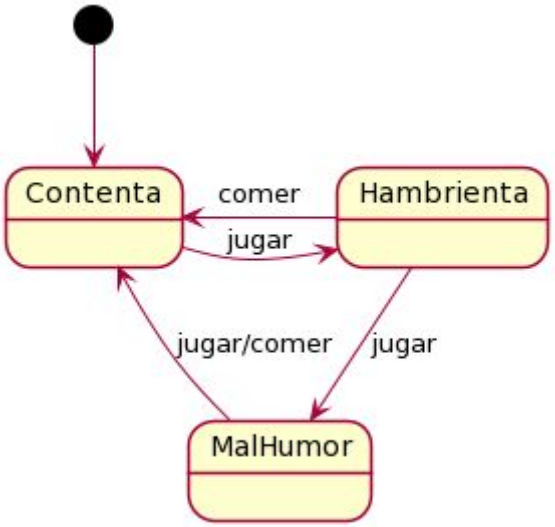
//TODO

- ~~[Req] - Mascota sabe jugar. *mascota.jugar() ?*~~
- ~~[Req] - Mascota sabe comer. *mascota.comer() ?*~~
- ~~[Req] - Mascota sabe decirte si tiene ganas de jugar. *mascota.conGanasDeJugar() ?*~~
- [Ref] - Mascota debería saber el mensaje, mascota.alegrarse()?

Vemos como gracias a haber modelado los estados como objetos, agregar el comportamiento de *jugar y conGanasDeJugar* fue trivial.

Analizando la solución

Usamos composición para modelar el comportamiento cambiante de la mascota. Una cosa interesante de esta composición es que es el estado es el que decide cuándo cambiar hacia otro estado y no un agente externo. Podemos ver estas transiciones si hacemos un diagrama de estados[4].

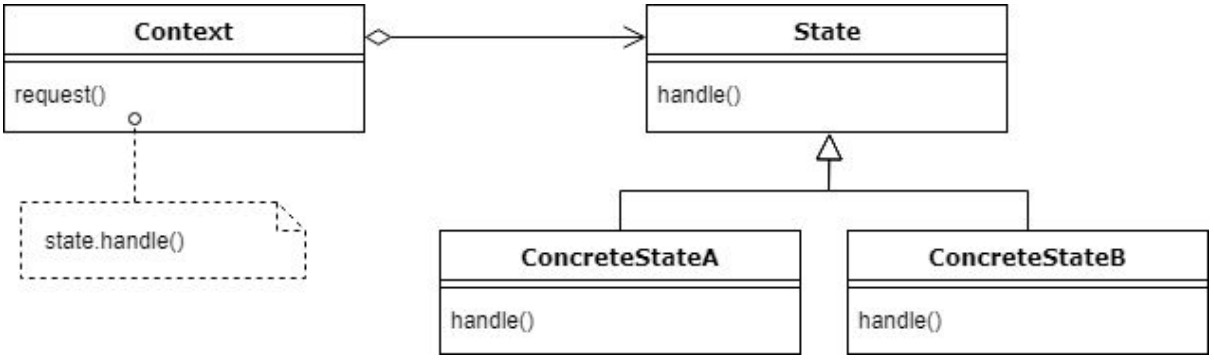


Entonces las responsabilidades de cada estado en particular van a ser:

1. Realizar el comportamiento correspondiente a lo que pide el dominio (*jugar/comer)*
2. Decidir si transiciona a un nuevo estado o no.

Esta idea de cosificar el estado para determinar el comportamiento de un tercero usando composición y que dichos estados puedan transicionar entre sí se ve plasmada en el patrón State.

Responsabilidades en el patrón State



[4] Guías para comunicar un diseño: Diagrama de estados

- ***Context***: es el que expone la operación a terceros y mantiene el estado actual. En nuestro ejemplo es la clase *Mascota*
- ***State***: Es la interfaz que contiene las operaciones de las cuales son responsables los estados, en nuestro es la interfaz *Estado*.
- ***ConcreteState***: Cada uno de los estados concretos que saben responder a la operación y si fuera necesario cambiarle el estado al ***Context***.

No todo lo que dice Estado es un State

Ante la palabra estado uno automáticamente piensa en un state y no siempre es así. Tiene que haber comportamiento en todos los estados y transiciones entre ellos para justificarlo. Muchas veces nos podría alcanzar con un booleano o con un enum que represente en qué estado estamos.

> Tenemos reservas hechas, que un operador manualmente va a verificar si el usuario que la compró cumple con los criterios definidos por la empresa, en caso de cumplir con dicho criterio pasa a un estado comprada y en caso contrario a rechazada. En caso de que esté en estado comprada se le podrá pedir el monto total de la compra.

Como vemos en esta situación, las operaciones que entienden los estados no son polimórficas, porque a veces se pueden hacer y otras veces no, incluso los más probable es que estas operaciones se usan en contextos distintos, por lo tanto no vale la pena forzar un polimorfismo inexistente. Podríamos llegar a una solución que use un enum para los estados solamente.

```
clase Entrada
   constructor()
     this.estado = Estado.PENDIENTE

  metodo confirmar()
    this.estado = Estado.CONFIRMADA

  metodo rechazar()
    this.estado = Estado.RECHAZADA

  metodo montoTotal()
      // retornar calculoDelMonto
```

Y si queremos romper en caso de que no sea el estado correspondiente para esa operación, podemos agregar una validación en base al enum antes de cada operación

```
clase Entrada
  metodo montoTotal()
      Validate.is(this.estado, Estado.CONFIRMADA)
      // retornar calculoDelMonto
```

¿Por qué esta composición no es un Strategy?

Si bien podemos pensar que es un Strategy, porque el diagrama de responsabilidades es muy similar, las intenciones de este patrón son otras:

- Permite cambiar su comportamiento en base a su estado interno.
- Acá no es un tercero el que define qué comportamiento usar, sino que es el estado mismo el que decide cuándo cambiarlo por otro comportamiento.

Bibliografía recomendada

- Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides