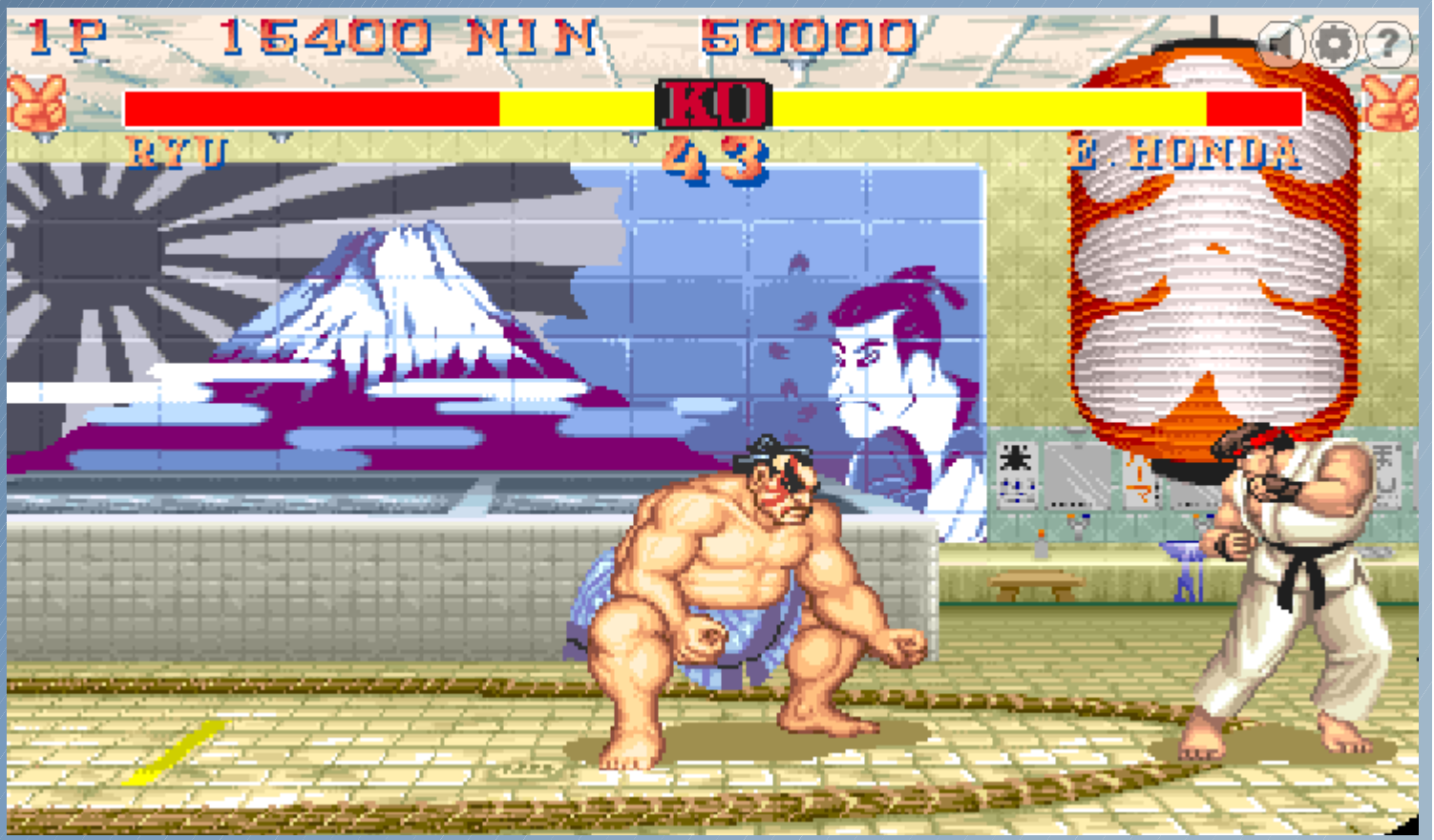
# Programación III

## Clase XXII

*Andrés Fortier*

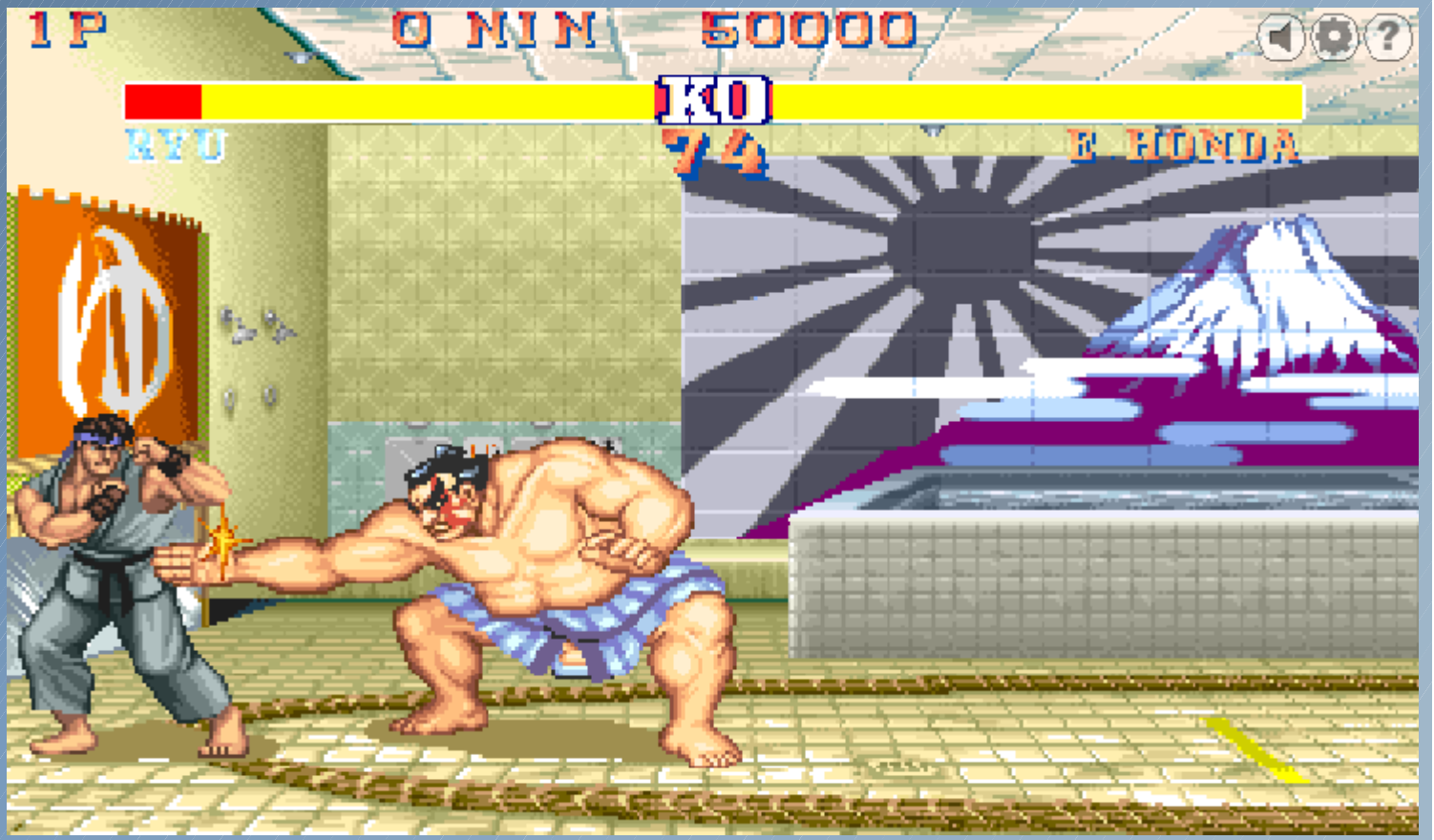
# Modelar los ataques en un juego

- Cuatro opciones:
  - Parado



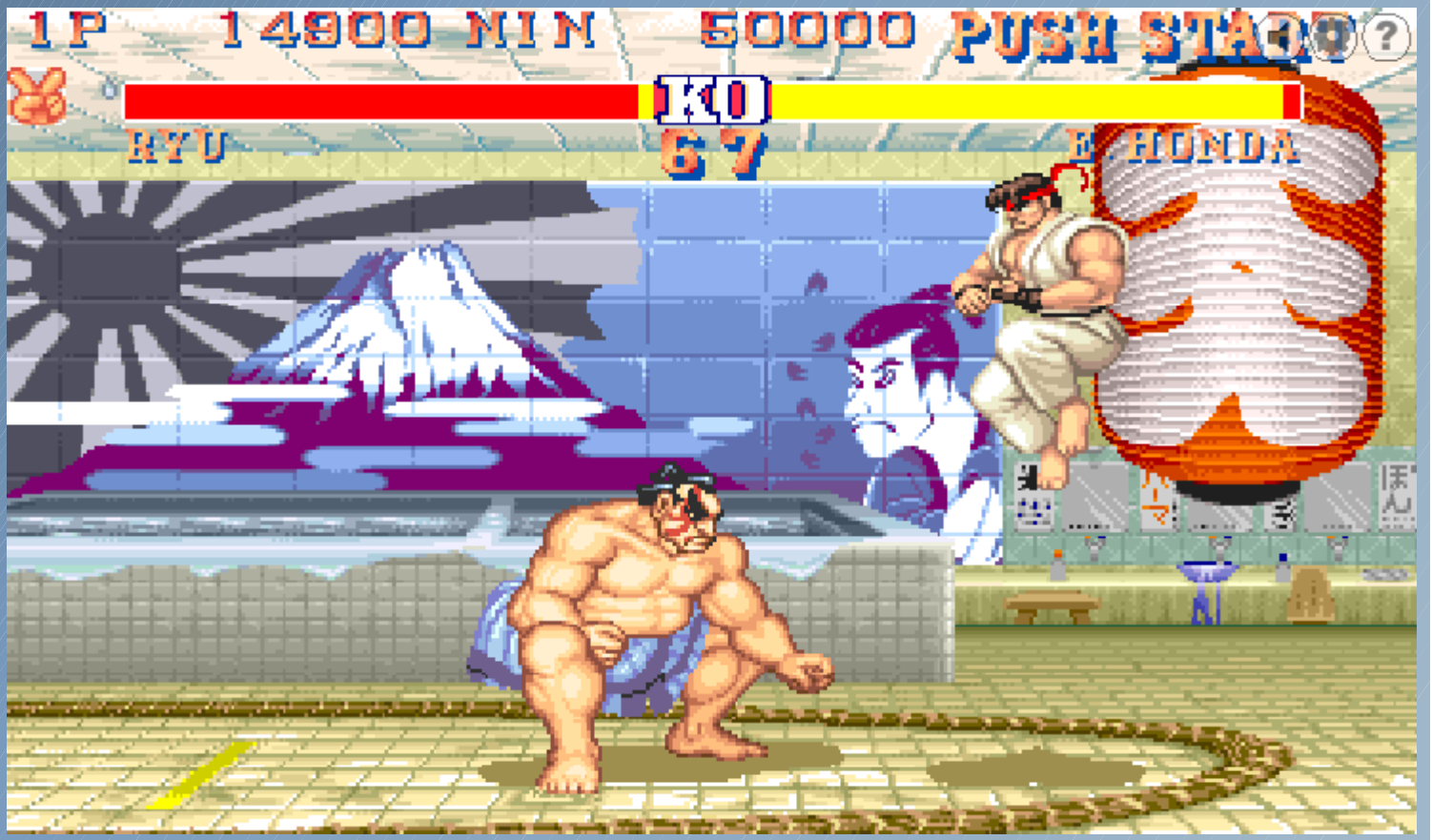
# Modelar los ataques en un juego

- Cuatro opciones:
  - Parado
  - Bloqueando



# Modelar los ataques en un juego

- Cuatro opciones:
  - Parado
  - Bloqueando
  - En el aire



# Modelar los ataques en un juego

- Cuatro opciones:
  - Parado
  - Bloqueando
  - En el aire
  - Agachado

# Modelar los ataques en un juego

- Ataques
  - Parado: resta 100 puntos.
  - Bloqueando: resta 15 puntos.
  - En el aire: no resta puntos.
  - Agachado: resta 70 puntos.

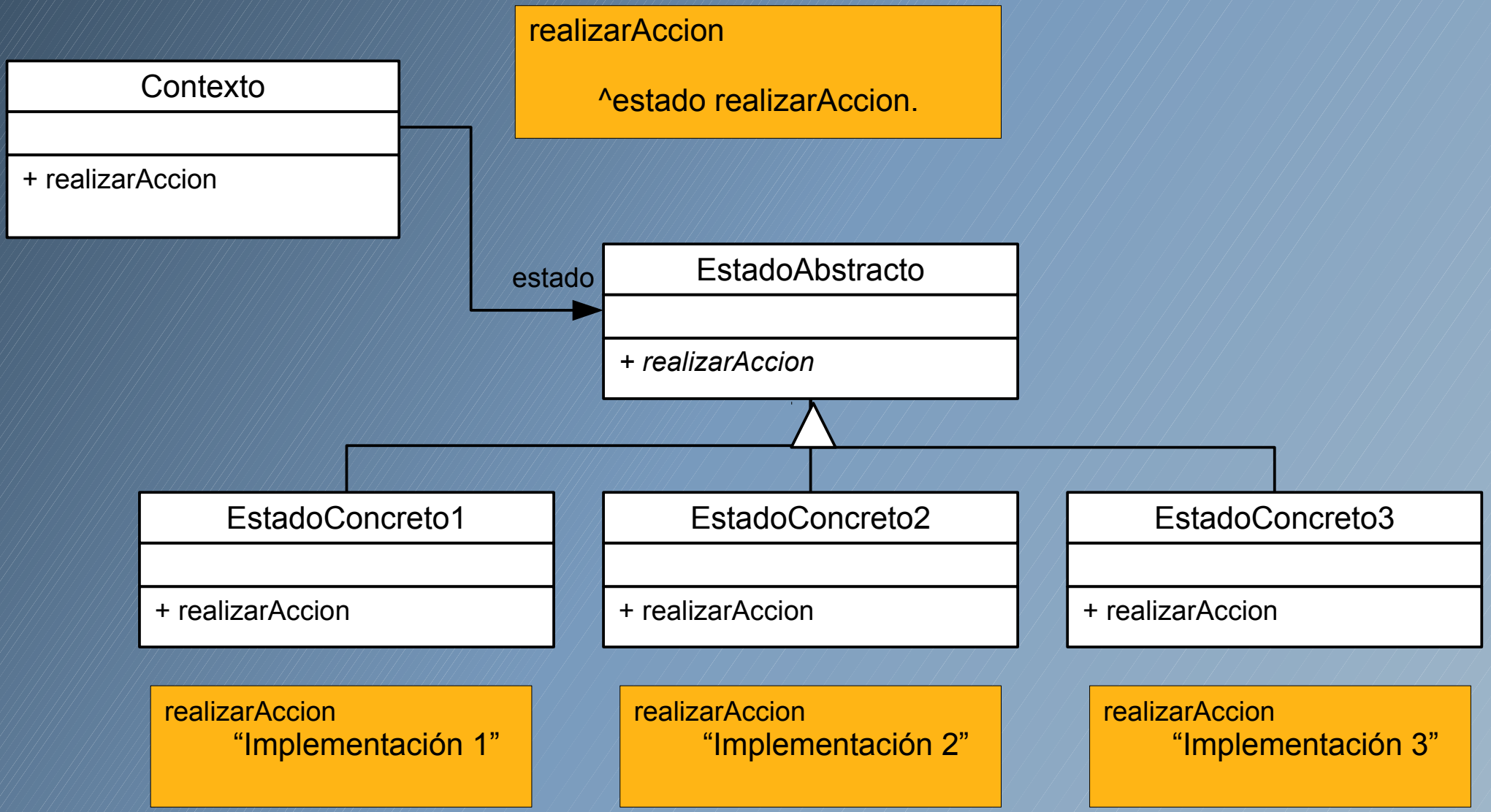
# Movimientos

- Al tocar las teclas izq/der el jugador se mueve (pensemos en moverAdelante / moverAtrás).
- El movimiento depende de su situación:
  - Parado: 10 unidades en el eje x.
  - En el aire: no se mueve.
  - Agachado: se mueve 4 unidades en el eje x.
  - Bloqueando: pasa a parado (no se cubre), está 250ms sin cubrirse, se mueve 10 unidades, está 250ms sin cubrirse, se cubre nuevamente.

# State (Design Pattern)

- Objetivo
  - Permitir que un objeto modifique su comportamiento en base a su estado interno.
- Participantes
  - Contexto (el jugador)
    - Es quien debe modificar su comportamiento en base al estado interno.
    - Define el protocolo público.
  - Estado abstracto
    - Define la interfaz que todo estado debe implementar.
  - Estados concretos
    - Proveen la implementación concreta para los casos particulares

# State (Design Pattern)



# State (Design Pattern)

- Colaboraciones
  - El contexto delega los mensajes dependientes del estado a las clases concretas que representan dicho estado.
  - El contexto puede pasarse como parámetro para que los estados lleven a cabo su trabajo.
  - El contexto es la interface pública para los clientes. Los clientes no saben nada de la implementación con estados del contexto.
  - La decisión de realizar una transición puede estar en el contexto o en los estados.

# State (Design Pattern)

- Consecuencias
  - Se genera una partición clara de los posibles estados.
  - Todo el comportamiento dependiente de un estado está localizado y encapsulado.
  - Las transiciones son explícitas.

# Envío de pagos

- Tenemos un sistema que usa PayPal para enviar pagos a usuarios.
- Usamos una clase provista por PayPal para hacer el envío.

| Usuario |
| --- |
| idPaypal |
| enviar: unMonto a: unUsuario<br>... |

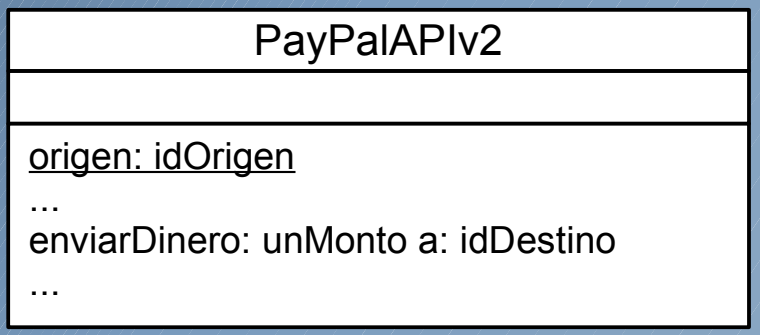
| PayPalAPI |
| --- |
| |
| ...<br>enviar: unMonto de: idOrigen a: idDestino<br>... |

```
Usuario>>enviar: unMonto a: unUsuario

	| api |
	api := PayPalAPI new.
	api enviar: unMonto de: self idPaypal a: unUsuario idPaypal
```

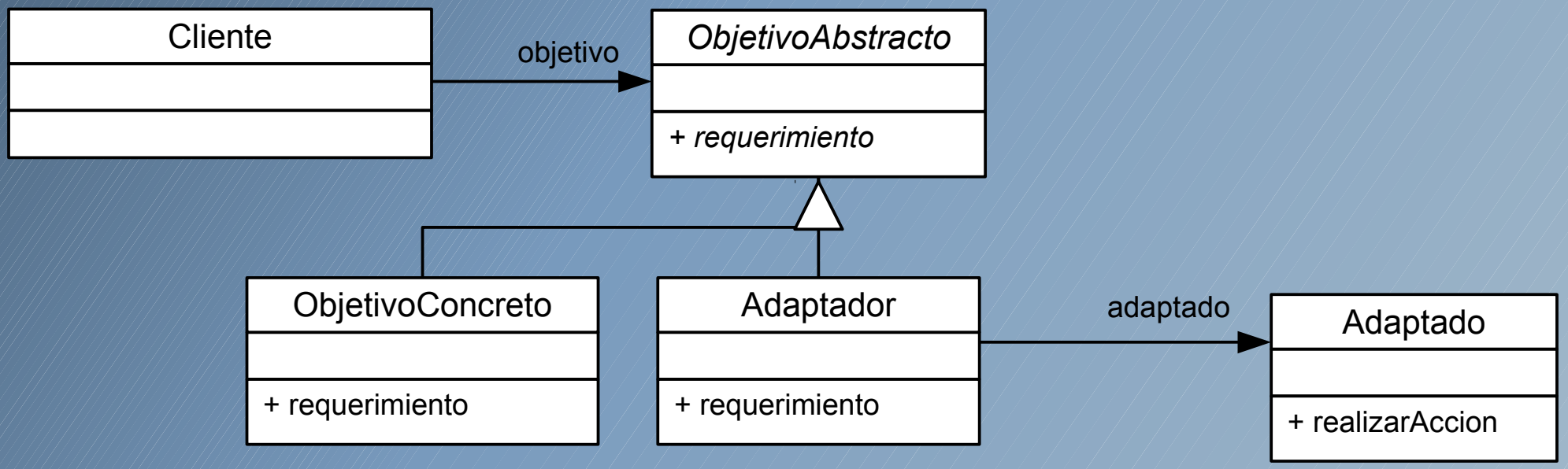
# Envío de pagos

- Paypal sacó una nueva versión de su API. Las cuentas nuevas deben usarla y las existentes deben usar la API anterior.
- ¿Cómo soportamos este nuevo requerimiento?



# Adapter

- Objetivo
  - Convertir al interface de un objeto en la que otro objeto cliente espera.
  - Permite que dos objetos que originalmente no son compatibles puedan colaborar.



# Adapter

- Participantes
  - Cliente: el que envía el/los mensaje(s) polimórfico(s).
  - Adaptado: provee una implementación concreta del protocolo, pero no es compatible con lo esperado por el cliente.
  - Adaptador: convierte el mensaje en uno que pueda manejar el adaptado.
  - ObjetivoAbstracto: define el protocolo.
  - ObjetivoConcreto: implementa el protocolo en forma concreta.

# Adapter

- Aplicabilidad
  - En un sistema existente, hay que incorporar una clase cuyo protocolo no es compatible con el esperado.
  - Es necesario crear un diseño que interactúe con diversas implementaciones existentes y que pueda acomodarse a clases desconocidas.
- A considerar
  - ¿Cuánto trabajo hace el adaptador?