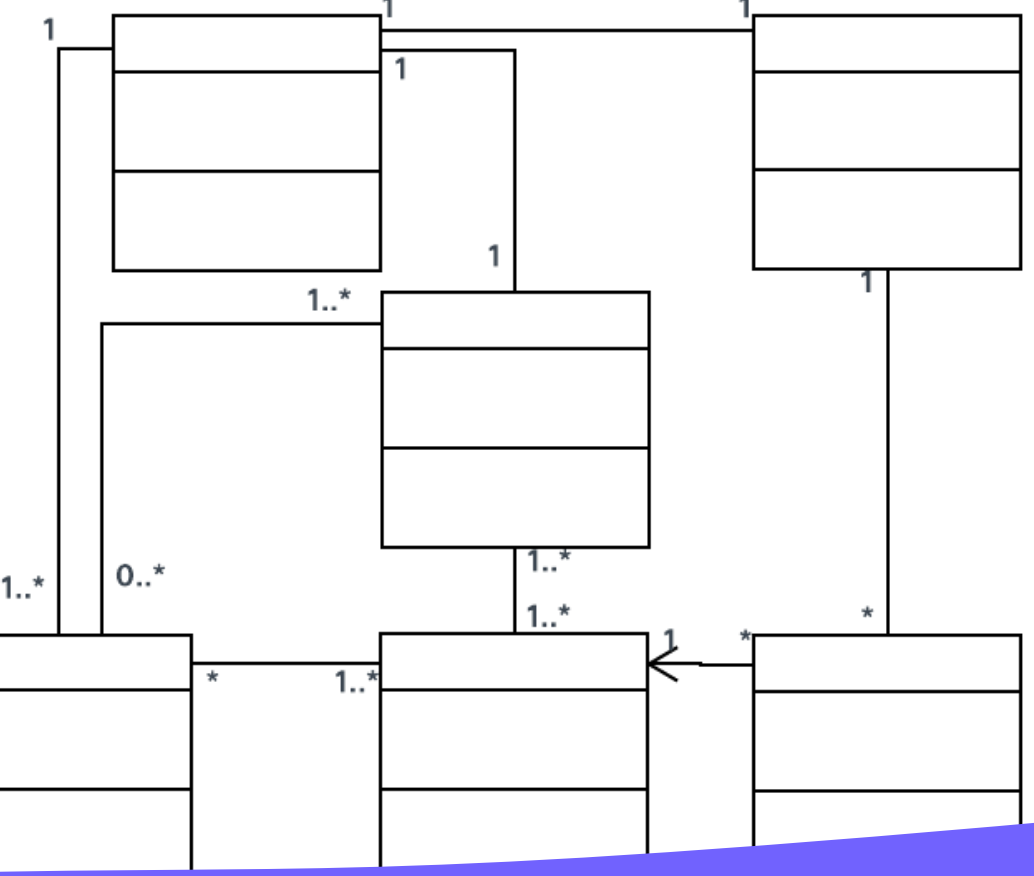
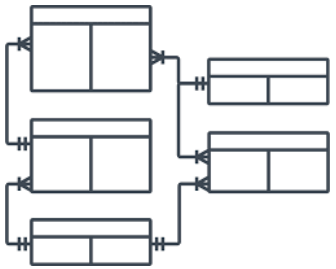




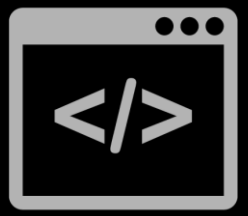
Diseño de Sistemas

UTN.BA
DPTO. INGENIERÍA EN SISTEMAS DE INFORMACIÓN
CÁTEDRA DISEÑO DE SISTEMAS

Agenda

- Testing
- Interfaces entre componentes: Interfaces entrantes y salientes. Ambientes. Cuestiones de Sincronismo.
- Mocking.
- Patrón Adapter.

Testing & Test Unitarios



¿Qué es Testing?

Testing es la verificación dinámica de la adecuación del Sistema a los requerimientos/requisitos.

Según la IEEE, el testing es una actividad en la cual un Sistema o componente de software es ejecutado, bajo condiciones específicas, para que los resultados de dicha ejecución sean observados y/o registrados; y a partir de los mismos realizar una evaluación de algún aspecto del Sistema o componente.

¿Qué es Testing?

- El testing busca encontrar fallas en el producto.
- Busca hacerlo lo más rápido y barato posible (busca la eficiencia).
- Busca encontrar la mayor cantidad de fallas
- Intenta no detector fallas que en realidad no lo son.
- Pretende encontrar las fallas más importantes.

¿Qué es Testing?

Una prueba es exitosa si encuentra fallas.

Testing – Conceptos generales

- ***Equivocación***: es una acción humana que produce un resultado incorrecto.
- ***Defecto***: paso, proceso o definición de dato incorrecto. También puede considerarse como la ausencia de cierta característica.
- ***Falla***: resultado de ejecución incorrecto (o distinto al valor esperado).

Testing – Conceptos generales

- Una *equivocación* lleva a uno o más *defectos* que están presentes en el Código.
- Un *defecto* lleva a cero, una o más *fallas*.
- La *falla* es la manifestación del *defecto*.
- Una *falla* tiene que ver con uno o más *defectos*.

Testing – Conceptos generales

- ***Condiciones de prueba***: son descripciones de situaciones que quieren probarse ante las cuales el Sistema debe responder.

- ***Casos de prueba***: son lotes de datos necesarios para que se dé una determinada condición de prueba.

Testing – Tipos de Tests

- *Pruebas Unitarias (de caja negra y de caja blanca)*
- *Pruebas de Integración*
- *Pruebas de Aceptación de Usuario (UAT)*
- *Pruebas de Stress*
- *Pruebas de Volumen y Performance*
- *Pruebas de Regresión*
- *Pruebas Alfa y Beta*

Tests Unitarios

Los tests unitarios se realizan sobre unidades de código claramente definidas.

Con la definición anterior, los tests unitarios podrían validar el correcto funcionamiento de una función en particular, o de un método de un objeto.

Tests Unitarios

Se suele realizar un test por cada método o por cada función para probar que cada parte aislada funciona de forma correcta.

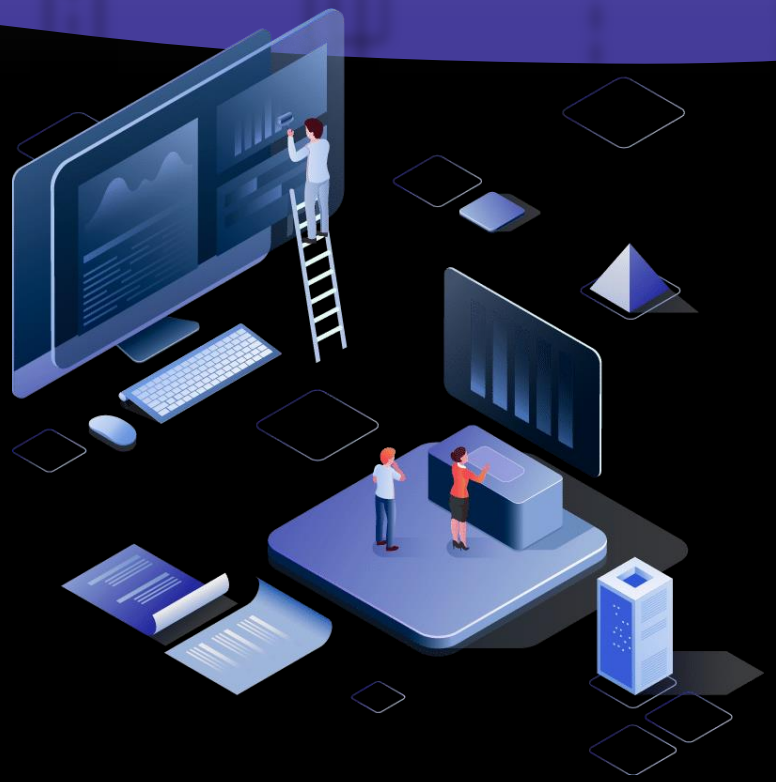
Tests Unitarios

- Generalmente las realizan las mismas personas que construyeron el módulo que se quiere probar
- Los módulos altamente cohesivos son los más sencillos de probar

Tests Unitarios

- Particularmente, en Java solemos utilizar JUnit como framework de testeo unitario.

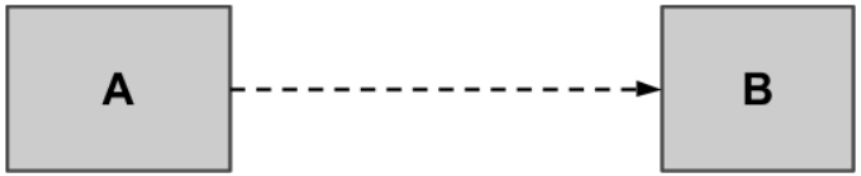
Interfaces entre Componentes



Interfaces entre Componentes

Sabiendo que un Sistema es un conjunto de Componentes que se relacionan para cumplir un objetivo en común:

¿Cómo se relacionan dichos componentes?



Interfaces entre Componentes

Consideraciones al momento de Diseñar:

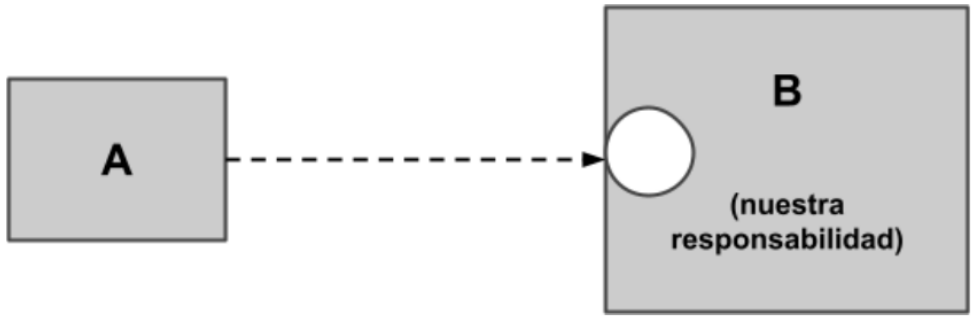
- *Dirección y sentido de la comunicación*
  - ***A*** llama a ***B***
  - ***B*** llama a ***A***
  - La comunicación es ***bidireccional***

- *Ambiente en el que residen los componentes*
  - Los componentes están en el mismo ambiente
  - Los componentes están en ambientes separados

- Sincronismo/Asincronismo en peticiones

Interface Entrante

La interface entrante es aquel contrato que define qué datos necesita nuestro componente (componente B en el gráfico) para poder realizar determinada tarea/ejecutar cierta funcionalidad.
Se debe tener en cuenta cómo utilizarán los terceros nuestro componente.



Interface Saliente

La interface saliente es aquel contrato que define qué datos necesita el componente al cual nos queremos integrar (componente B en el gráfico) para poder realizar determinada tarea/ejecutar cierta funcionalidad.
En este caso, se considera que nuestro componente A está “consumiendo” al componente B.

Ambientes

Al integrar componentes debemos tener en cuenta dónde están ubicados físicamente los mismos:

- En el mismo Sistema
- En Sistemas diferentes

Ambientes – Mismo Sistema

En el caso que ambos componentes se encuentren en el mismo Sistema, la integración entre ellos resultará sencilla, aunque:

- Se debe prestar atención al grado de acoplamiento que se podría generar entre ambos componentes, reduciendo el mismo al máximo posible.
- Se debe evitar la ruptura del encapsulamiento del componente integrado.
- Se debe diseñar una integración Testeable.

Ambientes – Sistemas diferentes

En el caso que los componentes se encuentren en Sistemas diferentes, la integración entre ellos resultará –quizás- más compleja que el caso anterior. En este caso:

- Debemos pensar cuál será el mecanismo para que ambos componentes se comuniquen (API REST, por ejemplo.)

Cuestiones de Sincronismo

¿Qué hacer cuando llamamos a un componente? ¿Quedarnos esperando? ¿Seguir trabajando?

Integraciones Sincrónicas

*Cuando un componente **A** hace uso de alguna funcionalidad expuesta por el componente B y **se queda esperando la respuesta**, entonces A está integrándose de forma **sincrónica** con B para la utilización de dicha funcionalidad.*

- Es la forma más simple de trabajar porque es fácil de razonar.
- Es la forma en la que trabaja el envío de mensajes entre objetos por defecto.

Integraciones Sincrónicas

Pero … ¿qué sucedería si consideramos que, en el siguiente caso, el Cotizador necesita realizar una Request a un componente externo y sabemos que la latencia es amplia?

```
dolarHoy = cotizador.obtenerCotizacion()
```

Estaríamos penalizando a todo el resto del Sistema por tan solo esperar la respuesta del cotizador para poder continuar con la labor.

Integraciones asincrónicas

*Cuando un componente **A** hace uso de alguna funcionalidad expuesta por el componente B y **no se queda esperando la respuesta**, entonces A está integrándose de forma **asincrónica** con B para la utilización de dicha funcionalidad.*

- El componente A puede no quedarse esperando la respuesta de B por alguno de los siguientes motivos:
  - No necesita el resultado/la respuesta en ese momento para continuar con la ejecución.
  - No le interesa el resultado/la respuesta.
  - No le estaba pidiendo nada, sino que le estaba "avisando de algo".

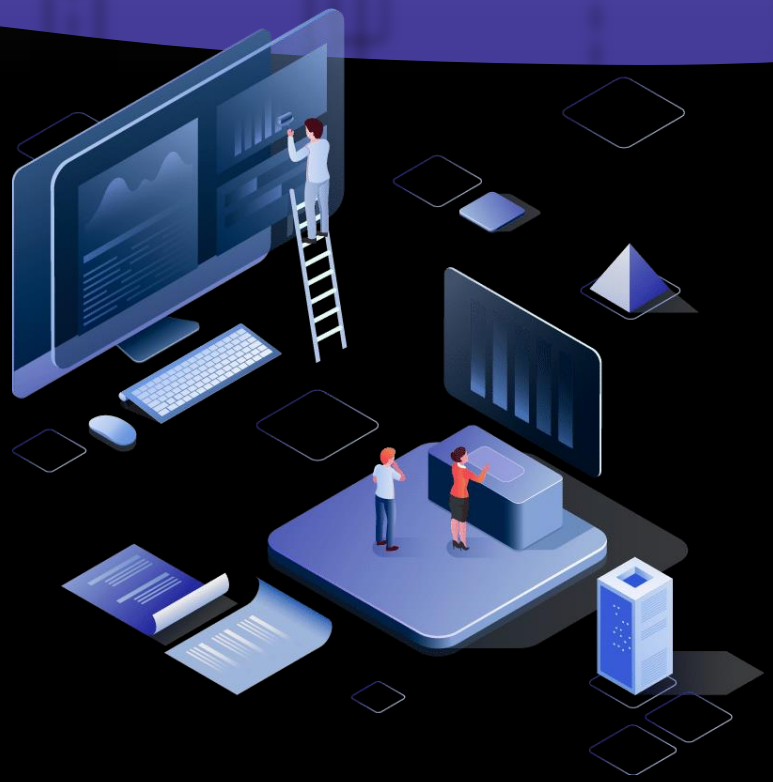
- No es la forma en la que estamos acostumbrados a pensar el común de las cosas.
- Es una buena manera de no penalizar toda la ejecución del Sistema.

Integraciones asincrónicas

Algunas de las formas que tiene A de enterarse, en un momento posterior, el resultado que generó el componente B, ante la ejecución de dicha funcionalidad, son:

- Preguntarle a B “cuál es el estado de la ejecución” o “si ya terminó”
  - Ante una mala utilización de este mecanismo, podríamos generar una espera activa en A.
- Leer periódicamente algún espacio de memoria compartido con B, en el cual B depositará el resultado.
  - Ante una mala utilización de este mecanismo, podríamos generar una espera activa en A.
- Pasarse como parámetro para que B le avise cuando haya terminado.
- Enviarle a B un callback para que ejecute luego de haber terminado la ejecución.

Mocking



Mocking

"En programación orientada a objetos, los objetos impostores, o mock objects, son objetos que simulan el comportamiento de los objetos reales, de forma controlada. Generalmente son programados para verificar el comportamiento de otro objeto, de forma parecida a que los diseñadores de autos utilizan muñecos (crash test dummies) para verificar el comportamiento de un auto durante un accidente."

Mocking

Podríamos utilizar un objeto mockeado cuando tengamos un objeto que:

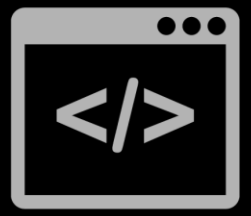
- Provea valores no controlables o aleatorios (por ejemplo, la fecha actual o la temperatura ambiente)
- Tenga estados difíciles de crear o reproducir (por ejemplo, un error de red)
- Tenga efecto colateral (por ejemplo, una base de datos, la cual debe ser inicializada antes del test)
- Tenga restricciones de performance (por ejemplo, al correr 200 veces una consulta a la base de datos)
- Aún no exista o su comportamiento pueda cambiar
- Deba incluir información y métodos exclusivamente para propósitos de testeo (no para su objetivo real)

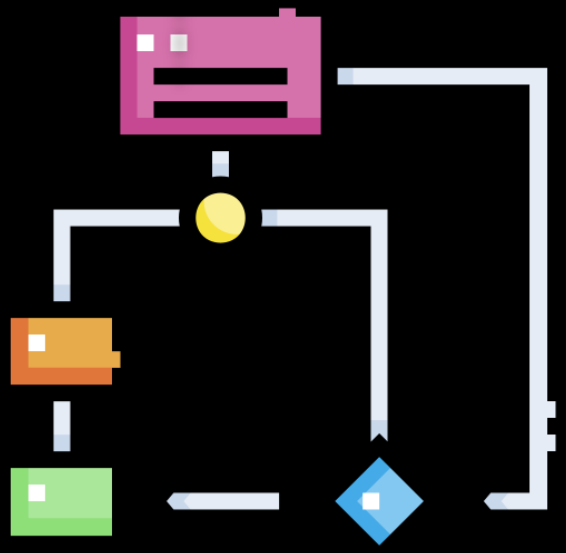
Mocking

Para mockear objetos se suele utilizar mockito (Java) o realizar una implementación propia.

Recomendamos la lectura del siguiente artículo para comprender la utilización de los mockeos.

Patrón Adapter





Patrón Adapter

Es un patrón Estructural

¿Qué hace?

- Encapsula el uso (llamadas/envío de mensajes) de la clase que se quiere adaptar en otra clase que concuerda con la interfaz requerida.

Patrón Adapter

Se sugiere su utilización cuando:

- Se requiere seguir adelante con el diseño y/o implementación (código) sin conocer exactamente cómo, quién y cuándo resolverá una parte necesaria; y solamente se conoce qué responsabilidad tendrá dicha parte.

- Se requiere usar una clase que ya existe, pero su interfaz no concuerda con la que se necesita.

Patrón Adapter

Componentes:

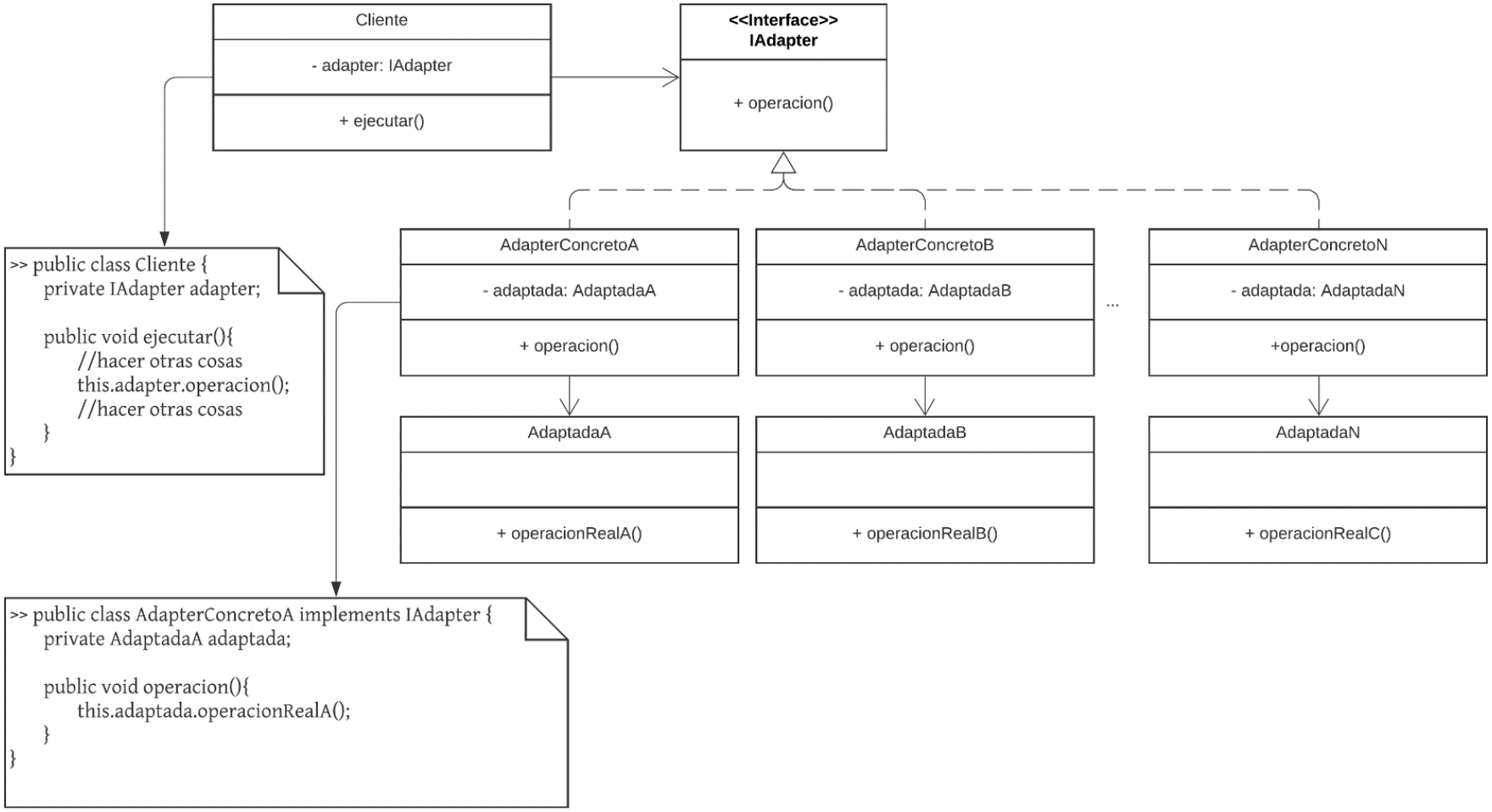
- ***Interface Adapter***: Define la firma del método que se utilizará como "puente" de acoplamiento a nuestro dominio.

- ***Clases de Adapters concretos***: Clases que implementan la interface Adapter, que se encargan de acoplar los componentes externos al dominio. Guardan referencia o hacen uso de las clases Adaptadas. Llaman a los métodos "reales" que resuelven el problema.

- ***Clases Adaptadas***: Clases externas al dominio, o no, que posee firmas incompatibles, o que todavía no conocemos (puede que todavía no estén creadas). Estas clases no se pueden/deben tocar.

- ***Cliente***: Clase que tiene referencia a la interface Adapter, cuyos objetos van a hacer uso de la funcionalidad que ésta les brinda.

Patrón Adapter



Gracias