Arquitectura Hexágonal

Cree su aplicación para que funcione sin una interfaz de usuario o una base de datos de tal forma que pueda ejecutar pruebas de regresión automatizadas, trabajar aún cuando la base de datos no este disponible y conectar aplicaciones sin la intervención del usuario.

> Una traducción del articulo de Alistair Cockburn: Hexagonal Architecture (https://web.archive.org/web/20180126145415/http://alistair.cockburn.us/Hexagonal+architecture). Todas las gracias a él por permitirnos traducir su articulo

(https://web.archive.org/web/20180126145415/http://alistair.cockburn.us/Hexagonal+architecture+pic+1-to-4+socket.jpg)

El patrón: Puertos y Adaptadores (”objeto- estructural”)

Nombre alternativo: ‘’Arquitectura hexagonal”

Intención

Permite que una aplicación sea usada de la misma forma por usuarios, programas, pruebas automatizadas o batch scripts, y sea desarrollada y probada en aislamiento (isolated) de sus eventuales dispositivos y bases de datos en tiempo de ejecución.

Todo evento externo (i.e. como pedidos http) que llega a la aplicación por un puerto, es convertido, a través de un adaptador especifico a la tecnología del evento externo, en una llamada a un procedimiento o un mensaje entendible por la aplicación. (Por ello) la aplicación es “felizmente” ignorante de la naturaleza de los dispositivos de entrada. Cuando la aplicación tiene algo que enviar, lo hace por un puerto a un adaptador, el cual crea las señales apropiadas, necesarias por la tecnología receptora (humana o automatizada). La aplicación tiene una interacción semántica con los adaptadores de todos sus lados, sin saber realmente la naturaleza de las cosas del otro lado de los adaptadores.

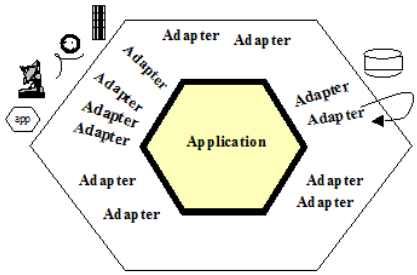

Figura 1

Motivación

Por muchos años, una de las grandes pesadillas en las aplicaciones de software ha sido la infiltración de la lógica del negocio en el código de la interfaz de usuario. Ello trae un problema triple:

- Primero, el sistema no puede ser fácilmente probado con suites de pruebas automatizadas porque parte de la lógica que se necesita probar depende de detalles visuales que cambian a menudo, como el tamaño de un campo de entrada o la posición de un botón
- Por la misma razón, se vuelve imposible cambiar su uso de uno dirigido por personas a un sistema de funcionamiento por lotes
- Por aún la misma razón, se vuelve difícil o imposible permitir al programa ser guiado (o usado) por otro programa cuando se desee.

La solución tentativa, repetida en muchas organizaciones, es crear una nueva capa en la arquitectura, con la promesa que esta vez, realmente y verdaderamente, ninguna lógica del negocio será puesta en esta nueva capa. Sin embargo, al no tener un mecanismo para detectar cuando una violación a esa promesa ocurre, unos años más tarde, la organización ve que la nueva capa esta mezclada con la lógica de negocio y el problema otra vez reaparece.

Imagina ahora que “cada” pieza de funcionalidad que la aplicación ofrece estuviera disponible a través de una API (application programmed interface) o llamadas a funciones. En este caso, el departamento de pruebas o QA puede ejecutar scripts de pruebas automatizadas para la aplicación con el fin de detectar cuando cualquier nuevo código rompe una función que antes funcionaba. Los expertos del negocio pueden crear casos de pruebas automatizados, antes que se completen los detalles de la GUI, que indican cuando los programadores han acabado su trabajo correctamente (y estas pruebas se convierten en pruebas que se ejecutan en el departamento de pruebas). La aplicación puede ser desplegada “sin una cabeza”, solo la API está disponible, cualquier otro programa puede hacer uso de su funcionalidad—esto simplifica el diseño general de suites de aplicaciones y además permite que las aplicaciones de servicio business-to-business (b2b, negocio a negocio) se utilicen entre sí, sin la intervención humana, a tráves la web. Finalmente, la función automatizada de las pruebas de regresión detecta cualquier violación en la promesa de mantener la lógica de negocio separada de la capa de presentación. La organización puede detectar, y luego corregir, la fuga de la lógica.

Un interesante problema similar existe con lo que es normalmente considerado “el otro lado” de la aplicación, donde la lógica de la aplicación está atada a una base de datos externa u otro servicio. Cuando el servidor de base de datos se cae o se somete a un importante re-trabajo o reemplazo, los programadores no pueden trabajar porque su trabajo está atado a la presencia de la base de datos. Esto trae costos de demora y a menudo malos sentimientos entre las personas.

No es obvio que los dos problemas estén relacionados, pero hay una simetría entre ellas que se indica en la naturaleza de la solución.

Naturaleza de la solución

Los problemas del lado del usuario y del lado del servidor son realmente causados por el mismo error de diseño y programación—el enredo entre la lógica de negocio y la interacción con entidades externas. La asimetría a usar no es la que existe entre el lado "izquierdo" y "derecho" de la aplicación (el que dirige y el dirigido por la aplicación, o la de los lados "superior" e "inferior" como se ve en la arquitectura 3 capas – la capa "superior", la presentación, dirige a la aplicación y la capa "inferior", fuente de datos, es dirigida o usada por la aplicación), sino entre los lados "interno" y "externo" (la aplicación en sí y su contexto) de la aplicación. Se debe obedecer la regla que todo código "interno" no debe fugarse (no debe estar) a la parte "externa".

Al remover por un momento cualquier asimetría izquierda-derecha o superior-inferior, se puede ver que la aplicación se comunica con agentes externos a través "puertos". La palabra "puerto" tiene la intención de traer recuerdos de los "puertos" en un sistema operativo, donde cualquier dispositivo que cumple con los protocolos de un puerto puede ser conectado (plug-in) y en los "puertos" de gadgets electrónicos, donde otra vez, cualquier dispositivo que cumpla con los protocolos mecánicos y eléctricos puede ser conectado.

- El protocolo de un puerto es dado por el propósito de la comunicación entre dos dispositivos.

El protocolo toma la forma de una interfaz de programación de aplicación (API).

Para cada dispositivo externo hay un "adaptador" que convierte la definición de la API en las señales que necesita ese dispositivo y viceversa. Una interfaz gráfica de usuario o GUI es un ejemplo de un adaptador que transforma los movimientos de una persona en la API de su puerto. Otros adaptadores que están en el mismo puerto (el lado del usuario, ver la Figura 2) son los arneses de pruebas automatizadas (test harness) como FIT o Fitnesse, controladores de lotes (batch drivers), y cualquier código necesario para la comunicación entre las aplicaciones en la empresa o la red (i.e. HTTP o Web Sockets).

En el otro lado de la aplicación, la aplicación se comunica con una entidad externa para obtener datos. Típicamente, el protocolo es un protocolo de base de datos. Desde la perspectiva de la aplicación, si la base de datos se mueve de una base de datos SQL a un archivo plano o a cualquier otro tipo de base de datos (i.e. NoSQL), la comunicación a través de la API no debería cambiar. Por ello, los adaptadores adicionales para el mismo puerto pueden ser: un adaptador para SQL, un adaptador para archivos planos, y más importante, un adaptador para una base de datos "mock" (simulada), una que sea en memoria y que no dependa de la presencia de la base de datos real.

Muchas aplicaciones solo tienen dos puertos: el diálogo con el lado del usuario y con el lado de la base de datos. Esto les da una apariencia asimétrica, lo que hace parecer natural construir la aplicación en una arquitectura unidimensional de tres, cuatro o cinco capas.

Hay 2 problemas con este tipo de esquemas. En primer lugar y lo peor, la gente tiende a no tomar las "líneas" en los esquemas de capas en serio. Dejan que la lógica de la aplicación se fugue (esté fuera) de los límites de la capa, causando los problemas mencionados antes. Segundo, podría haber más de 2 puertos en la aplicación, así que la arquitectura no encaja en el esquema de capas unidimensional.

La arquitectura hexagonal, o puertos y adaptadores, resuelve estos problemas al observar la simetría del caso: hay una aplicación interna que se comunica con cosas externas a través de un número de puertos. Los elementos externos de la aplicación pueden ser tratados con simetría.

El hexágono tiene el propósito de resaltar visualmente:

(a) la asimetría interna-externa y la naturaleza similar de los puertos, con el fin de alejarse de la imagen unidimensional de las capas y todo lo que ella evoca, y

(b) la presencia de un número definido de diferentes puertos— 2, 3 o 4 (cuatro es lo mayor que he encontrado hasta hoy).

El hexágono no es un hexágono por que el número seis es importante, sino más bien para permitir que la gente que hace el esquema tenga espacio para insertar puertos y adaptadores cuando lo necesite, sin estar restringido a un esquema unidimensional en capas. El término “arquitectura hexagonal” viene de este efecto visual.

El término “puerto y adaptadores” reúne los “propósitos” de las partes del esquema. Un puerto identifica una comunicación útil. Típicamente, habrá muchos adaptadores para cualquier puerto, para varias tecnologías que puedan conectarse a un puerto. Típicamente, estás podrían ser: un contestador automático, un teléfono táctil o por voz, una GUI, un test harness, un batch driver, una interfaz http, una interfaz directa programa a programa (i.e. Llamadas a Procedimientos Remotos- RPC), una base de datos mock (en memoria) o una real (quizá distintas bases de datos para el desarrollo, pruebas y producción).

En la sección Notas de Aplicación, la simetría izquierda-derecha será vista otra vez. Sin embargo, el propósito principal de este patrón es enfocarse en la asimetría interior-exterior, pretendiendo que todos los elementos externos son idénticos desde la perspectiva de la aplicación.

Estructura

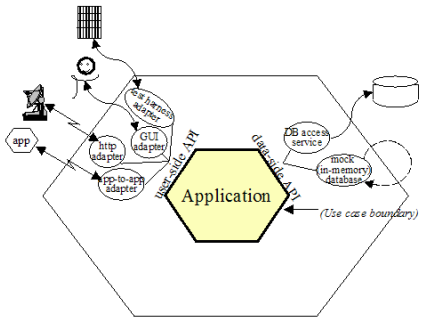

Figura 2

La Figura 2 muestra una aplicación que tiene 2 puertos activos y muchos adaptadores para cada puerto. Los 2 puertos son el lado que controla la aplicación y el lado de obtención de datos. Este esquema indica que la aplicación puede ser igualmente dirigida por suites de pruebas automatizadas a nivel de sistema, por un usuario humano, por una aplicación remota a través de http , o por otra aplicación local. En el lado de los datos, la aplicación puede ser configurada para ejecutarse sin acoplamiento de bases de datos externas usando una base de datos de reemplazo en memoria, o “mock” o puede ser ejecutada bajo la base de datos de pruebas o producción. La especificación funcional de la aplicación, quizá en Casos de Uso, es hecha bajo la interfaz del hexágono interno y no en cambio en cualquier tecnología externa que podría ser usada.

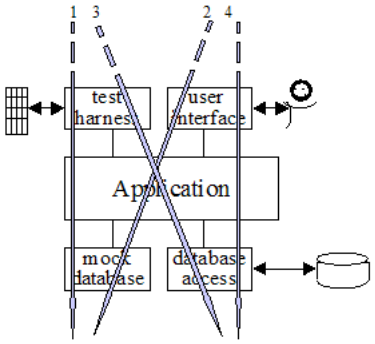

Figura 3

La Figura 3 muestra la misma aplicación puesta en un esquema de una arquitectura en 3 capas. Para simplificar el esquema solo 2 adaptadores son mostrados por cada puerto. Este esquema tiene la intención de mostrar como múltiples adaptadores encajan en las capas superior e inferior, y la secuencia en la cual varios adaptadores son usados durante el desarrollo del sistema. Las flechas numeradas muestran el orden en el cual el equipo podría desarrollar y usar la aplicación:

1. Con un test harness FIT conduciendo la aplicación y usando una base de datos mock (en memoria) que sustituye la base de datos real
2. Agregando una GUI a la aplicación, aún ejecutnado la aplicación bajo una base de datos mock
3. En pruebas de integración, con scripts de pruebas automatizadas (e.g., de Cruise Control—este artículo es un poco antiguo, ejemplos más actuales son Jenkins, Travis CI o Go) ejecutando la aplicación en una base de datos real que contiene datos de prueba
4. En el uso real, con una persona usando la aplicación para acceder a la base de datos real.

Código de ejemplo

Afortunadamente, la aplicación más simple que demuestra los puertos y adaptadores viene de la documentación de FIT. Es una aplicación simple de procesamiento de descuentos:

```
discount(amount) = amount * rate(amount)
```

En nuestra adaptación, el monto(amount) viene del usuario y la razón(rate) viene de la base de datos, así que habrán 2 puertos. Lo implementaremos en etapas:

- Con pruebas pero con una razón constante en vez de una base de datos mock,
- luego con la GUI,
- luego con una base de datos mock de tal forma que pueda ser cambiada a una real.

Gracias a Gyan Sharma del IHC por proveer el código para este ejemplo.

Etapa 1: Aplicación FIT con constante como base de datos mock

Primero creamos los casos de prueba en una tabla HTML (vea la documentación de FIT para ello):

TestDiscounter

amount	discount()
100	5
200	10

Note que los nombres de las columnas serán los nombres de las clases y funciones en nuestro programa. FIT contiene formas para alejarse de esta forma “programadora“ (escribir como código), pero para este artículo es más fácil dejarlo así.

Sabiendo cuales serán los datos de prueba, crearemos el adaptador del lado del usuario, usando el ColumnFixture de FIT:

```
        import fit.ColumnFixture;          public class TestDiscounter extends ColumnFixture          {              private Discounte
r app = new Discounter();              public double amount;              public double discount()              { return app.discount(am
ount); }          }
```

Eso es todo para este adaptador. Hasta aquí, las pruebas se ejecutan por la línea de comandos (mira el libro FIT para el path que necesitarás). Usaremos este:

```
set FIT_HOME=/FIT/FitLibraryForFit15Feb2005 java -cp %FIT_HOME%/lib/javaFit1.1b.jar%FIT_HOME%/dist/fitLibraryForFit.jarsrcbin fit.FileR
unner test/Discounter.html TestDiscount_Output.html
```

FIT produce un archivo de salida con colores que nos muestran que paso(o que fallo, en caso que hallamos hecho un error de ortografía—typo— en algún lugar).

En este punto el código está listo para ser registrado, integrado a Cruise Control o cualquier máquina de construcción (build) automatizada , y ser incluida en el suite de construcción y prueba.

Etapa 2: Aplicación UI con constante como base de datos

Voy a dejar que crees tu propio UI y que esta conduzca (use) a la aplicación Discounter (de descuentos), dado que el código es un poco largo para incluirlo aquí. Algunas lineas clave en el código son estás:

```
             ...         Discounter app = new Discounter();         public void actionPerformed(ActionEvent event)          {
...             String amountStr = text1.getText();             double amount = Double.parseDouble(amountStr);             discount = a
pp.discount(amount));             text3.setText( "" + discount );             ...
```

En este punto la aplicación puede ser usada en una demo y en pruebas de regresión. Ambos adaptadores del lado del usuario funcionan y pueden ser ejecutados.

Etapa 3: Aplicación (FIT o UI) con base de datos mock

Para crear un adaptador reemplazable para el lado de la base de datos, crearemos: una “interfaz” al repositorio, un “RepositoryFactory” (fábrica de repositorios, vea el patrón Factory) que producirá bien la base de datos mock o la real y el mock en memoria para la base de datos.

```
      public interface RateRepository  {    double getRate(double amount);  } public class RepositoryFactory  {    public RepositoryFac
tory() {  super(); }    public static RateRepository getMockRateRepository()     {       return new MockRateRepository();    } } public
class MockRateRepository implements RateRepository  {    public double getRate(double amount)     {       if(amount <= 100) return 0.0
1;       if(amount <= 1000) return 0.02;       return 0.05;     }  }
```

Para conectar este adaptador con la aplicación Discounter, necesitamos actualizar la aplicación para que acepte el adaptador de repositorio a usar y hacer que el adaptador del lado del usuario (FIT o UI) pase como argumento el repositorio (real o mock) al constructor de la aplicación en sí (Discounter). Aquí les muestro la aplicación actualizada

y el adaptador para FIT que pasa como argumento un repositorio mock (el código del adaptador FIT a escoger al pasar el adaptador del repositorio mock o el real es grande y no agrega mucha nueva información, así que omito esa versión aquí).

```
      import repository.RepositoryFactory; import repository.RateRepository; public class Discounter  {    private RateRepository rateR
epository;    public Discounter(RateRepository r)     {       super();       rateRepository = r;     }    public double discount(double
amount)     {       double rate = rateRepository.getRate( amount );        return amount * rate;     } } import app.Discounter; import
fit.ColumnFixture; public class TestDiscounter extends ColumnFixture  {    private Discounter app =         new Discounter(RepositoryFa
ctory.getMockRateRepository());    public double amount;    public double discount()     {       return app.discount( amount );    } }
```

Eso concluye la implementación de la versión más simple de la arquitectura hexagonal.

Para una implementación diferente, usando Ruby y Rack para el uso en el navegador, mire https://github.com/totheralistair/SmallerWebHexagon (https://web.archive.org/web/20180126145415/https://github.com/totheralistair/SmallerWebHexagon)

Notas de su aplicación

La asimetría izquierda-derecha

El patrón puertos y adaptadores es deliberadamente escrito, pretendiendo que todos los puertos son fundamentalmente similares. Esa pretensión es útil a un nivel de arquitectura. En la implementación, los puertos y adaptadores se muestran en 2 formas, las cuales las llamo “primaria” y “secundaria, por razones obvias..prontamente. También podrían ser llamadas adaptadores “directoras” y “dirigidas”.

El lector atento habrá notado que en todos los ejemplos dados, los fixtures de FIT son usados en los puertos del lado izquierdo y los mocks en el derecho. En la arquitectura de 3 capas, FIT está en la capa superior y los mocks en el lado inferior.

Esto esta relacionado a la idea en los casos de uso “actores primarios” y “actores secundarios”. Un “actor primario” es un actor que dirige la aplicación (lo que saca del lado inactivo para realizar una de sus dichas funciones). Un “actor secundario” es aquel que la aplicación dirige, bien para obtener respuestas o simplemente para notificarlo. La distinción entre “primario” y “secundario” radica en quien dispara (trigger) o está a cargo de la conversación.

El adaptador de prueba natural para sustituir a un actor “primario” es FIT, dado que es un framework diseñado para leer un script y dirigir a la aplicación. El adaptador de prueba natural para sustituir a un actor “secundario” es una base de datos mock, dado que está diseñada para responder a consultas o registrar eventos de la aplicación.

Estas observaciones nos ayudan a comprender el diagrama de contexto de casos de uso y dibujar los “puertos primarios” y “adaptadores primarios” en el lado izquierdo (o superior) del hexágono, y los “puertos secundarios” y “adaptadores secundarios” en el lado derecho (o inferior) del hexágono.

La relación entre los puertos/adaptadores primarios y secundarios y su implementación respectiva en FIT y mocks es útil tenerla en cuenta, pero debería ser usado como consecuencia de usar la arquitectura de puertos y adaptadores (no para un análisis excesivo). El beneficio último de la implementación de los puertos y adaptadores es la habilidad de poder ejecutar la aplicación en un modo totalmente aislado.

Casos de Uso y los limites de la aplicación

Es útil usar el patrón de arquitectura hexagonal para reforzar la forma preferida de escribir los casos de uso.

Un error común es escribir los casos de uso que contienen un conocimiento íntimo de las tecnologías externas a cada puerto (cuantas veces no hemos escrito y el usuario aprieta el botón de color azul o el registro de la persona se guarda en la base de datos). Estos casos de uso han ganado justificadamente un mal nombre en la industria por ser largos, difíciles de leer, aburridos, frágiles y costosos de mantener.

Al entender la arquitectura de puertos y adaptadores, podemos ver que los casos de uso deberían generalmente ser escritos en los límites de la aplicación (dentro del hexágono), para especificar las funciones y eventos soportados por la aplicación, sin importar la tecnología. Estos casos de uso son más cortos, fáciles de leer, menos costosos de mantener y más estables en el tiempo.

¿Cuantos puertos?

Lo que exactamente es y no es un puerto es en gran medida una cuestión de gustos. En un extremo cada caso de uso podría tener su propio puerto, produciendo cientos de puertos en muchas aplicaciones. Alternativamente, uno podría imaginar juntar todos los puertos primarios y todos los puertos secundarios de tal forma que solo hubieran dos, el del lado izquierdo y el derecho.

Ninguno de estos extremos es bueno.

El sistema del clima descrito en “Usos conocidos” tiene cuatro puertos naturales: el feed del clima, el administrador, los subscriptores notificados, la base de datos de subscriptores. Un controlador de una máquina de café tiene 4 puertos: el usuario, la base de datos que contiene las recetas y los precios, los dispensadores y la caja de monedas. Un sistema de medicación de un hospital puede tener 3: uno para los enfermeros, una para la base de datos de prescripciones, y uno para el controlador de controlador del dispensador de medicamentos.

No parece haber algún daño en escoger un número “equivocado” de puertos, por lo que está sigue siendo una cuestión de intuición. Aún así prefiero escoger un número pequeño de puertos: 2,3 o 4, como se describió antes y en Usos conocidos.

Usos conocidos

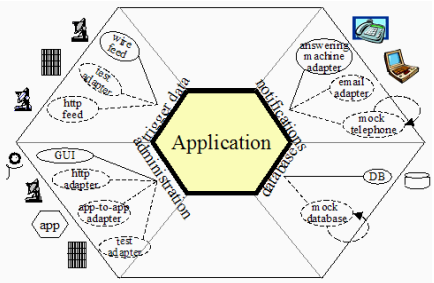

Figura 4

La Figura 4 muestra una aplicación con cuatro puertos y muchos adaptadores para cada puerto. Esta fue derivada de una aplicación que escuchaba alertas del servicio nacional del clima de terremotos, tornados, incendios e inundaciones, y esta notificaba a las personas por sus teléfonos. En la época que discutíamos de este sistema, las interfaces del sistema fueron identificadas y discutidas por “tecnología, vinculada al propósito”. Había una interfaz para datos disparados (trigger-data) que llegaban de un cable alimentador, uno para los datos de notificación que se enviaban a las máquinas de respuesta, una interfaz administrativa implementada en una GUI, y una interfaz para la base de datos para obtener los datos de sus subscriptores.

La gente estaba luchando porque necesitaban agregar una interfaz http del servicio del clima, una interfaz de email para sus subscriptores, y necesitaban encontrar una forma agrupar y desagrupar su creciente suite de aplicaciones para las distintas preferencias de compra de los clientes. Tenían miedo que estuvieran entrando en una pesadilla de mantenimiento y pruebas, ya que tenían que implementar, probar y mantener versiones separadas para todas las combinaciones y permutaciones.

Su cambio en el diseño fue organizar las interfaces del sistema “por propósito” en vez de por tecnología, y hacer que las tecnologías sean sustituibles (en todos los lados) por cada adaptador. Inmediatamente escogieron que funciones incluir: el feed de http y la notificación por email (los nuevos adaptadores se muestran en el diagrama con lineas discontinuas). Por hacer cada aplicación independientemente ejecutable a través de su API, ellos podían agregar un adaptador por aplicación a agregar y desagrupar el suite de la aplicación, conectando sub-aplicaciones según se necesitaba. Finalmente, por hacer cada aplicación independientemente ejecutable, con pruebas y adaptadores, ganaron la habilidad de hacer pruebas de regresión a sus aplicaciones con script de pruebas aútonomas.

Mac, Windows, Google, Flickr, Web 2.0

A inicios de los 90, las aplicaciones de MacIntosh como los procesador de textos requerían tener interfaces controladas por API, de modo que otras aplicaciones y scripts escritos por usuarios podían acceder a todas las funciones de la aplicación. Las aplicaciones de escritorio de Windows han hecho lo mismo (no tengo el conocimiento histórico para decir quien lo hizo primero, ello no es relevante al punto que trato de mostrar).

La tendencia actual (2005) en las aplicaciones web es publicar una API y permitir que otras aplicaciones web accedan a su API directamente. Por lo tanto, es posible publicar publicar datos de crimines locales sobre un mapa de Google, o crear aplicaciones las funciones de archivado y anotación de fotos de Flickr.

Todos estos ejemplos son acerca de hacer la API de los puertos “primarios” visible. No vemos la información de los puertos secundarios.

Almacenando salidas

Este ejemplo escrito por Willem Bogaerts en el wiki C2 :

> Encontré algo similar, principalmente era distinto por que mi capa de aplicación manejaba cosas que no debía hacer, tenia una fuerte tendencia a volverse un switchboard para teléfonos. Mi aplicación generaba una salida, la mostraba al usuario y luego tenia la posibilidad de guardar la salida. Mi principal problema era que no siempre era necesario guardar la salida. Así que mi aplicación generaba una salida, tenia que ponerla en un buffer y presentarla al usuario. Luego, si el usuario quería guardar la salida, la aplicación obtenía los datos del buffer y lo guardaba.
>
> Esto no me gustó en lo absoluto. Entonces se me ocurrió una solución: Tener un controlador de la presentación con facilidades de almacenamiento. Ahora la aplicación. Ahora la aplicación ya no enviaba las salidas en diferentes direcciones, sino simplemente enviaba la salida al controlador de la presentación. Era el controlador de la presentación que ponía las respuestas en el buffer y le daba al usuario la posibilidad de almacenarlas.
>
> La arquitectura tradicional en capas enfatiza la diferencia entre la “UI” y el “almacenamiento”. La arquitectura puertos y adaptadores puede reducir la salida a ser simplemente otra vez “salida”.

Un ejemplo anónimo del wiki C2

"En un proyecto en el que trabajé, usamos la metáfora de un sistema para los componentes de un sistema de estéreo. Cada componente tenia interfaces definidas, de las cuales cada una tenia un propósito especifico. Podíamos conectar componentes en casi ilimitadas formas usando simples cables y adaptadores."

Desarrollo de equipos grandes y distribuidos

Aún este está en pruebas y por lo tanto propiamente no cuenta como un uso del patrón. Sin embargo, es interesante considerarla.

Los equipos en diferentes ubicaciones construyen una arquitectura hexagonal, usando FIT y mocks de tal forma que las aplicaciones o componentes pueden ser autónomamente probadas. CruiseControl se ejecuta cada media hora y corre todas las aplicaciones usando una combinación de FIT y mocks. A medida que se completan subsistemas y bases de datos, los mocks son reemplazados con bases de datos de prueba.

Separando el desarrollo de la UI y la lógica de la aplicación

Aún este está en pruebas y por lo tanto propiamente no cuenta como un uso del patrón. Sin embargo, es interesante considerarla.

El diseño de la UI es inestable, dado que no se ha decidido la tecnología o una idea aún. La arquitectura de los servicios del backend no ha sido decidida, y de hecho cambiara muchas veces a lo largo de los siguientes 6 meses. Sin embargo, el proyecto ha comenzado oficialmente y el tiempo sigue corriendo.

El equipo de la aplicación crea pruebas para FIT y mocks para aislar su aplicación (de cualquier cosa que venga: bases de datos, frameworks, etc), y crea funcionalidades probables y demostrables para mostrar a sus usuarios. Cuando las decisiones en los servicios de UI y backend finalmente acaben, "debería ser directo" juntarlos. Mantente atento en aprender cómo esto funciona (o pruebe usted mismo y escríbame para hacerme saber).

Patrones relacionados

Adapter (Adaptador)

El libro "Design Patterns" contiene una descripción del patrón genérico "Adapter": "Convierte una interfaz de una clase en otra interfaz que el cliente espera." El patrón puertos y adaptadores es un uso particular del patrón "Adapter".

Model-View-Controller (Modelo-Vista-Controlador- MVC)

El patrón MVC fue implementado en los inicios de 1974 en el proyecto Smalltalk. Se han dado, sobre los años, muchas variaciones, como el Modelo-Interactuador y Modelo-Vista-Presentador. Cada uno de ellos implementa la idea de puertos y adaptadores en los puertos primarios, no en los secundarios.

Objetos Mock y Loopback (bucle invertido)

"Un objeto mock es un "agente doble" usado para probar el comportamiento de otros objetos. Primero, un objeto mock actúa como una implementación falsa de una interfaz o clase, es decir imita el comportamiento externo de una verdadera implementación. Segundo, un objeto mock observa como otros objetos interactuan con sus métodos y compara el comportamiento actual con salidas esperadas. Cuando hay una diferencia, un objeto mock puede

interrumpir la prueba y reportar la anomalía. Si la diferencia no puede ser descubiertas durante las pruebas, un método de verificación llamado por el tester asegura que todas las expectativas se cumplan o se reportan los errores".—De http://MockObjects.com (https://web.archive.org/web/20180126145415/http://mockobjects.com/)

Los objetos mock, completamente implementados de acuerdo a su agenda, son usados en toda la aplicación, no solo para interfaces externas. El principal empuje del movimiento de objetos mock es la conformidad con el protocolo especificado (su interfaz) a nivel de clase y objeto individual. Tomo prestado su palabra "mock" como la mejor descripción breve de un sustituto en memoria de un actor secundario.

El patrón Loopback (bucle invertido) es un patrón explicito para crear un reemplazo interno para un dispositivo externo.

Pedestals

En "Patterns for Generating a Layered Architecture", Barry Rubel describe un patrón en la creación de un eje de simetría en un software de control que es muy similar al de puertos y adaptadores. El patrón "Pedestal" propone implementar un objeto que represente cada dispositivo de hardware dentro del sistema, y enlazar esos objetos en una capa de control. El patrón "Pedestal" puede ser usado para describir cualquier lado de la arquitectura hexagonal, pero no resalta la similaridad entro adaptadores. Además, siendo escrito para un ambiente de control mecánico, no es fácil ver como aplicar el patrón para otras aplicaciones de TI.

Checks

El lenguaje de patrones de Ward Cunningham para detectar y manejar los errores de la entrada del usuario, es buena para el manejo de errores a lo largo de los límites internos del hexágono.

Inversión de dependencias (Inyección de dependencias) y SPRING

El principio de inversión de dependencias de Bob Martin (también llamado Inyección de dependencias por Martin Fowler) indica "Los módulos de mayor nivel no deberían depender de módulos de menor nivel. Ambos deberían depender de abstracciones. Las abstracciones no deberían depender de detalles. Los detalles deberían depender de abstracciones". El patrón "Inyección de dependencias" de Martin Fowler da algunas implementaciones. Estas muestran como crear adaptadores secundarios reemplazables. El código se puede escribir directamente, como se hace en el código de ejemplo del articulo, o usando archivos de configuración y teniendo al framework de SPRING generando código equivalente.

Agradecimientos

Gracias a Gyan Sharma del Intermountain Health Care por proveer el código de ejemplo usado aquí. Gracias a Rebecca Wirfs-Brock por su libro "Object Design", el cual leído junto con el patrón "Adapter" del libro "Design Patterns" , me ayudo a entender de qué se trataba el hexágono. Gracias también a las personas en el wiki de Ward (Ward Cunningham), quienes dieron comentarios del patrón por muchos años (e.g., particularmente Kevin Rutherford's http://silkandspinach.net/blog/2004/07/hexagonal_soup.html) (https://web.archive.org/web/20180126145415/http://silkandspinach.net/blog/2004/07/hexagonal_soup.html).

Referencias y lecturas adicionales

FIT, A Framework for Integrating Testing: Cunningham, W., online at http://fit.c2.com (https://web.archive.org/web/20180126145415/http://fit.c2.com/), and Mugridge, R. and Cunningham, W., ‘‘Fit for Developing Software”, Prentice-Hall PTR, 2005.

The ‘‘Adapter” pattern: in Gamma, E., Helm, R., Johnson, R., Vlissides, J., ‘‘Design Patterns”, Addison-Wesley, 1995, pp. 139–150.

The ‘‘Pedestal” pattern: in Rubel, B., “Patterns for Generating a Layered Architecture”, in Coplien, J., Schmidt, D., ‘‘PatternLanguages of Program Design”, Addison-Wesley, 1995, pp. 119–150.

The ‘‘Checks” pattern: by Cunningham, W., online at http://c2.com/ppr/checks.html (https://web.archive.org/web/20180126145415/http://c2.com/ppr/checks.html)

The ‘‘Dependency Inversion Principle“: Martin, R., in ‘‘Agile Software Development Principles Patterns and Practices”, Prentice Hall, 2003, Chapter 11: “The Dependency-Inversion Principle”, and online at http://www.objectmentor.com/resources/articles/dip.pdf (https://web.archive.org/web/20180126145415/http://www.objectmentor.com/resources/articles/dip.pdf)

The ‘‘Dependency Injection” pattern: Fowler, M., online at http://www.martinfowler.com/articles/injection.html (https://web.archive.org/web/20180126145415/http://www.martinfowler.com/articles/injection.html)

The ‘‘Mock Object” pattern: Freeman, S. online at http://MockObjects.com (https://web.archive.org/web/20180126145415/http://mockobjects.com/)

The ‘‘Loopback” pattern: Cockburn, A., online at http://c2.com/cgi/wiki?LoopBack (https://web.archive.org/web/20180126145415/http://c2.com/cgi/wiki?LoopBack)

‘‘Use cases:” Cockburn, A., ‘‘Writing Effective Use Cases”, Addison-Wesley, 2001, and Cockburn, A., “Structuring Use Cases with Goals”, online at http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm (https://web.archive.org/web/20180126145415/http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswit