Arquitectura Hexagonal

Cree su aplicación para que funcione sin interfaz de usuario ni base de datos, de manera que usted pueda ejercitar la aplicación ejecutando pruebas de regresión automatizadas; para que funcione cuando la base de datos no esté disponible; y comunique aplicaciones entre sí sin intervención por parte del usuario.

El Patrón: *PORTS AND ADAPTERS* [1] ("Patrón Estructural de Objetos")

Nombres alternativos: *"Ports & Adapters"* , "Arquitectura Hexagonal"

Autores

Alistair Cockburn, autor del artículo original.

Juan Manuel Garrido de Paz, autor de esta traducción.

Propósito

Permitir que una aplicación sea ejecutada indistintamente por usuarios, programas, pruebas automatizadas, o archivos *batch* ; y que sea desarrollada y probada por separado, sin los posibles dipositivos y bases de datos de los que dependa en tiempo de ejecución.

Cuando llega un evento del mundo exterior, un adaptador dependiente de la tecnología lo convierte en una llamada a un procedimiento utilizable, o en un mensaje, de un puerto [2]. La aplicación ignora completamente la naturaleza del dispositivo de entrada. Cuando la aplicación tiene algo que enviar, lo hace a través de un puerto a un adaptador, el cual crea las señales oportunas que necesita la tecnología del dispositivo receptor (sea éste un ser humano o un sistema automatizado). La aplicación mantiene una interacción basada en un diálogo semántico con los adaptadores que la rodean, no conoce la naturaleza de los dispositivos que están al otro lado de los adaptadores.

1 El nombre del patrón se considera un nombre propio, por lo que las referencias a él siempre se harán utilizando el nombre original en inglés *(Ports and Adapters)*. En español sería "Puertos y Adaptadores".

2 Esta frase no se corresponde con la del artículo original. Ver el Anexo "Notas sobre la traducción".

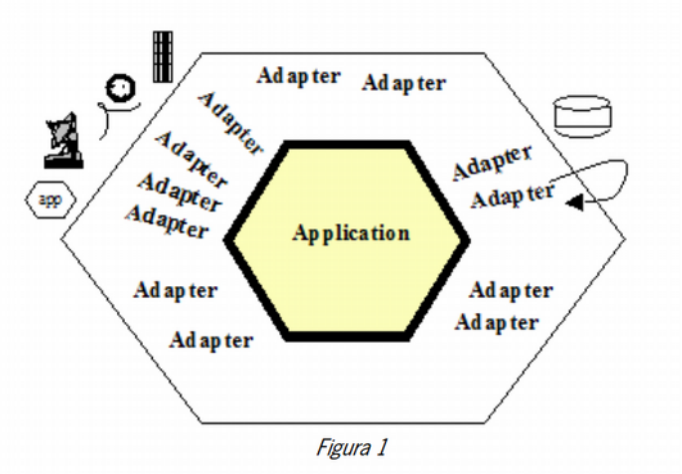

Figura 1

Motivación

Una de las grandes pesadillas de las aplicaciones software a lo largo de los años ha sido la inclusión de lógica de negocio en el código fuente de la interfaz de usuario. Esto origina tres problemas:

* Primero, el sistema no se puede probar con *suites* de pruebas automatizadas, porque parte de la lógica que se necesita probar depende de detalles visuales como el tamaño de los campos y el posicionamiento de los botones.

* Exactamente por la misma razón, resulta imposible pasar de un uso del sistema por parte de un ser humano, a una ejecución desatendida del sistema.

* Y también por la misma razón, resulta difícil o imposible hacer que el programa sea ejecutado por otro programa, cuando ésto sea interesante.

El intento de solución, llevado a cabo en muchas organizaciones, es crear una nueva capa en la arquitectura, con la promesa de que esta vez, real y verdaderamente, no se pondrá ninguna lógica de negocio en la nueva capa. Sin embargo, sin tener ningún mecanismo para detectar el incumplimiento de esa promesa, la organización se da cuenta pocos años después de que la nueva capa está repleta de lógica de negocio, y de que ha reaparecido el antiguo problema.

Imaginemos por un momento que cada funcionalidad que la aplicación ofrece estuviera disponible a través de un API [3] (Interfaz de Programación de Aplicaciones) o de la llamada a una función. En dicho caso, el departamento de pruebas o el de QA [4] pueden ejecutar *scripts* de pruebas automatizadas, para detectar cuándo un nuevo código hace que una función que previamente funcionaba, deje de funcionar. Los expertos del negocio pueden crear casos de pruebas automatizados, antes de que se hayan finalizado los detalles de la interfaz gráfica de usuario, lo cual sirve a los programadores para saber cuándo han realizado correctamente

3 Siglas en inglés de "Application Programming Interface".
4 Siglas en inglés de "Qualiity Assurance" (Aseguramiento de la Calidad).

su trabajo (y estas pruebas serán las ejecutadas por el departamento de pruebas). La aplicación se puede desplegar en modo *headless* [5], donde sólo el API está disponible, y otros programas pueden hacer uso de su funcionalidad – esto simplifica el diseño global de sistemas de aplicaciones complejas, y también permite que las aplicaciones con servicios negocio-a-negocio se llamen unas a otras a través de la web sin intervención del ser humano. Por último, las pruebas de regresión automatizadas detectan cualquier incumplimiento de la promesa de mantener la lógica de negocio fuera de la capa de presentación. La organización puede detectar, y después corregir, la fuga de lógica.

Un problema similar interesante existe en lo que normalmente se considera "el otro lado" de la aplicación, donde la lógica de aplicación está ligada a una base de datos externa u otro servicio. Cuando el servidor de base de datos se cae o está bajo mantenimiento o necesita ser reemplazado, los programadores no pueden trabajar porque su trabajo está ligado a la presencia de la base de datos. Esto genera costes de demora y a menudo malas sensaciones entre la gente.

Que los dos problemas tengan relación no es obvio, pero hay una simetría entre ambos que se muestra en la naturaleza de la solución.

Naturaleza de la Solución

Tanto el problema del lado del usuario como el del lado del servidor son causados, de hecho, por el mismo error al diseñar y programar – la confusión entre lo que es lógica de negocio y la interacción con entidades externas. La asimetría no es aquella entre los lados "izquierdo" y "derecho" de la aplicación, sino entre el "interior" y el "exterior" de la aplicación. La regla que tenemos que cumplir es que el código perteneciente a la parte "interior" no debería invadir la parte "exterior".

Olvidando por un momento cualquier asimetría entre la parte izquierda y la derecha, o entre la parte superior y la inferior, vemos que la aplicación se comunica con los sistemas externos a través de "puertos". La palabra "puerto" recuerda a los "puertos" de un sistema operativo, donde cualquier dispositivo que cumple con los protocolos de un puerto se puede conectar a él; y a los "puertos" de los aparatos electrónicos, donde igualmente, cualquier dispositivo que cumpla con los protocolos mecánicos y eléctricos puede ser conectado.

* El protocolo de un puerto viene dado por el propósito de la conversación entre los dos dispositivos.

* El protocolo adopta la forma de una "Interfaz de Programación de Aplicaciones" (API).

5 Modo de funcionamiento sin interfaz de usuario.

Para cada dispositivo externo hay un adaptador que convierte la definición del API a las señales que necesita el dispositivo y viceversa. Una "Interfaz Gráfica de Usuario" o GUI [(6)] es un ejemplo de adaptador que convierte los movimientos de una persona al API del puerto. Otros adaptadores que encajan en el mismo puerto son: *frameworks* para pruebas automatizadas como *FIT* o *Fitnesse*, programas para ejecutar archivos *batch*, y cualquier código necesario para la comunicación entre aplicaciones a través de la red.

En otro lado de la aplicación, ésta se comunica con una entidad externa para obtener datos. Típicamente, el protocolo es un protocolo de base de datos. Desde el punto de vista de la aplicación, si la base de datos pasa de ser una base de datos SQL a ser un archivo de texto plano o cualquier otro tipo de base de datos, la conversación a través del API no debería cambiar. Por tanto otros adaptadores para el mismo puerto podrían ser: un adaptador SQL, un adaptador para archivo de texto plano, y lo más importante, un adaptador para una base de datos *mock*, que resida en memoria y no dependa en absoluto de la presencia de la base de datos real.

Muchas aplicaciones sólo tienen dos puertos: el diálogo de la parte del usuario, y el de la parte de la base de datos. Esto les da una apariencia asimétrica, lo cual hace que construir la aplicación con una arquitectura unidimensional de tres, cuatro o cinco capas, parezca algo normal.

Hay dos problemas con este tipo de diagramas. El primero y peor, es que se tiende a no tomar en serio las líneas del diagrama. Se permite que la lógica de la aplicación traspase las fronteras de la capas, causando los problemas mencionados anteriormente. El segundo es que puede haber más de dos puertos en la aplicación, de manera que la arquitectura no encaja en el diagrama de capas unidimensional.

La arquitectura hexagonal, o *Ports and Adapters*, resuelve estos problemas teniendo en cuenta la simetría de la situación: hay una aplicación en la parte interior que se comunica a través de un cierto número de puertos con cosas del exterior. Las cosas de la parte externa a la aplicación se pueden tratar de manera simétrica.

Se pretende que el hexágono resalte visualmente:

(a) La asimetría entre el interior y el exterior, y la naturaleza similar de los puertos, para librarse del esquema unidimensional en capas y todo lo que conlleva.

(b) La presencia de un número determinado de diferentes puertos – dos, tres, o cuatro (cuatro es el máximo que he visto hasta la fecha).

6 Siglas en inglés de "Graphical User Interface".

El hexágono no es un hexágono porque el número seis sea importante, sino más bien para permitir que quien hace el dibujo tenga espacio para insertar puertos y adaptadores conforme lo necesite, sin estar limitado por un diagrama unidimensional en capas. El término "arquitectura hexagonal" proviene de este efecto visual.

El término *Ports and Adapters* proviene de los "propósitos" de los distintos elementos que conforman el dibujo. Un puerto identifica una conversación con un propósito. Usualmente habrá múltiples adaptadores para cada puerto, para las distintas tecnologías que se puedan conectar a dicho puerto. Como posibles ejemplos de estos adaptadores tenemos: un contestador automático, la voz humana, un teléfono, una interfaz gráfica de usuario, un *framework* para puebas automatizadas, un archivo *batch* , una interfaz *http* , una interfaz directa programa-a-programa, una base de datos *mock* (en memoria), una base de datos real (puede que distintas bases de datos para los entornos de desarrollo, pruebas y producción).

En el apartado "Notas sobre la Aplicación del Patrón", se mencionará de nuevo la asimetría entre las partes izquierda y derecha. Sin embargo, la intención principal de este patrón es centrarse en la asimetría entre la parte interior y la exterior, considerando que todas las entidades externas son idénticas desde la perspectiva de la aplicación.

Estructura

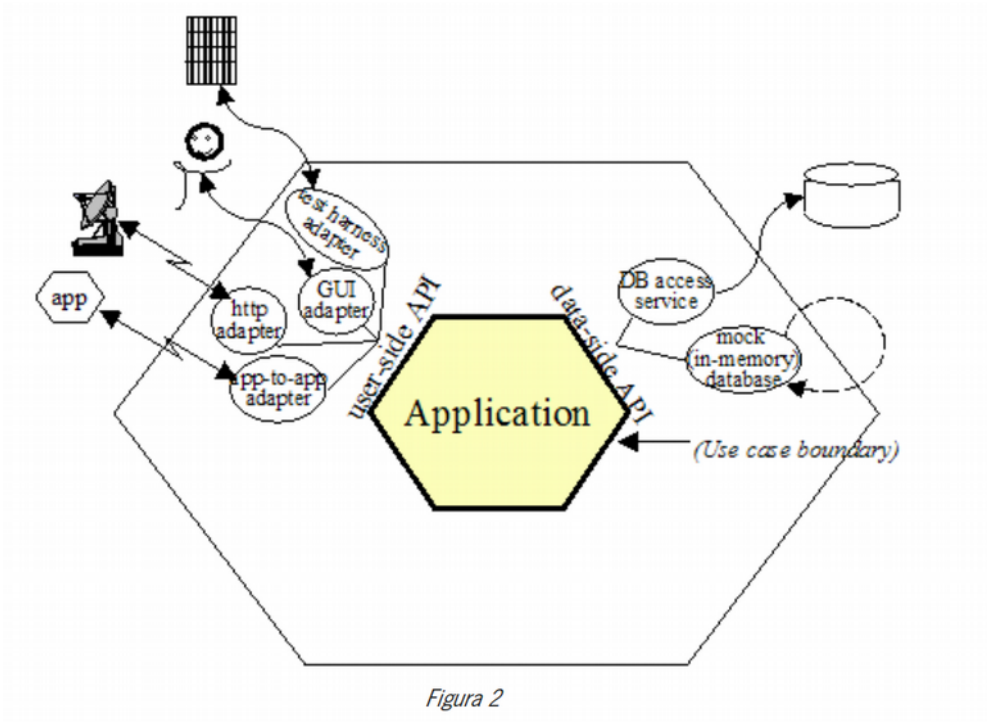

Figura 2

La figura 2 muestra una aplicación con dos puertos activos y varios adaptadores para cada puerto. Los dos puertos son el lado que controla la aplicación y el lado para recuperar datos. Este dibujo muestra que la aplicación puede ser dirigida de igual manera por una *suite* de pruebas automatizadas de regresión a nivel de sistema, por un ser humano, por una aplicación *http* remota, o por otra aplicación local. En el lado de los datos, la aplicación se puede configurar para ejecutarse desacoplada de bases de datos externas, utilizando una base de datos de sustitución Oracle en memoria, o *mock* ; o se puede ejecutar con la base de datos de pruebas o con la de producción. La especificación funcional de la aplicación, puede que en casos de uso, se hace contra la interfaz del hexágono interno, y no contra ninguna de las tecnologías externas que se podrían utilizar.

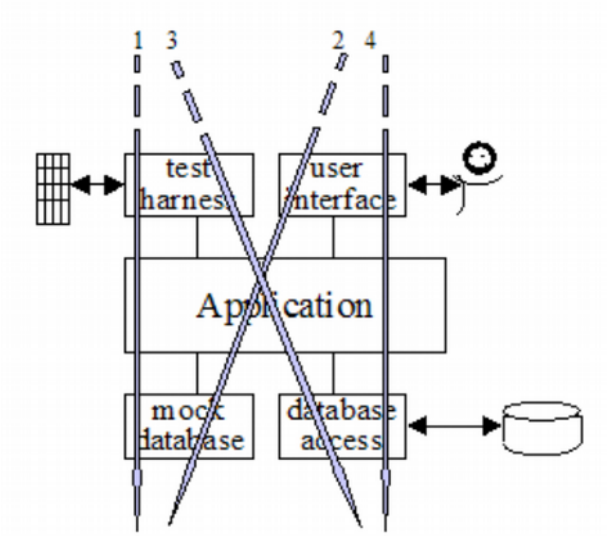

Figura 3

La figura 3 muestra la misma aplicación convertida en un diagrama de arquitectura en tres capas. Para simplificar el diagrama, sólo se muestran dos adaptadores para cada puerto. Con este diagrama se pretende mostrar cómo múltiples adaptadores encajan en las capas superior e inferior, y la secuencia en la que se usan los distintos adaptadores durante el desarrollo del sistema. Las flechas numeradas muestran el orden en el que un equipo debería desarrollar y usar la aplicación:

1. Con un *framework* de pruebas *FIT* dirigiendo la aplicación, y usando la base de datos *mock* (en memoria) sustituyendo a la base de datos real.

2. Añadiendo una GUI a la aplicación, manteniendo la base de datos *mock*.

3. Realizando pruebas de integración, con *scripts* de pruebas automatizadas (por ejemplo desde *"CruiseControl"*) dirigiendo la aplicación, contra una base de datos real que contenga datos de prueba.

4. Uso real, con una persona utilizando la aplicación para acceder a una base de datos de producción.

Código de Ejemplo

Aforturnadamente, la aplicación más simple que muestra la arquitectura *Ports and Adapters* viene con la documentación de *FIT* . Es una sencilla aplicación para calcular un descuento:

discount (amount) = amount * rate (amount); [7]

En nuestra adaptación, la cantidad vendrá dada por el usuario y el porcentaje estará en una base de datos, de manera que habrá dos puertos. Los implementamos por etapas:

- Con pruebas, pero con un porcentaje constante en lugar de una base de datos *mock*.

- Con la GUI.

- Con una base de datos *mock* que puede ser intercambiada por una base de datos real.

Gracias a Gyan Sharma de IHC por facilitar el código para este ejemplo.

Etapa 1: *FIT* | Aplicación | Constante (en lugar de una Base de Datos *mock*)

Primero creamos los casos de prueba en una tabla HTML (ver la documentación de *FIT*):

TestDiscounter

amount	discount()
100	5
200	10

Observar que los nombres de las columnas serán nombres de clases y de funciones en nuestro programa. *FIT* tiene formas de saltarse estas normas de terminología, pero para este artículo es más fácil simplemente dejarlo tal cual está.

Sabiendo cuáles serán los datos de prueba, creamos el adaptador del "lado del usuario", el *ColumnFixture* que viene con *FIT* :

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

7 "discount" significa descuento, "amount" cantidad, y "rate" porcentaje.

Esto es todo en lo que al adaptador se refiere. Hasta aquí, las pruebas se ejecutan desde la línea de comandos (ver el libro sobre *FIT* para saber qué ruta necesitaremos). Utilizamos la siguiente:

```
set FIT_HOME=/FIT/FitLibraryForFit15Feb2005
java -cp %FIT_HOME%/lib/javaFit1.1b.jar;%FIT_HOME%/dist/fitLibraryForFit.jar;src;bin
fit.FileRunner test/Discounter.html TestDiscount_Output.html
```

FIT genera un archivo de salida con colores mostrándonos qué fue satisfactorio (o qué falló, en el caso de que cometiéramos un error tipográfico en algún lugar a lo largo del proceso).

Llegados a este punto el código está listo para pasarlo a *"CruiseControl"* u otra herramienta de integración continua, e incluirlo en la *suite* de compilación y pruebas.

Etapa 2: UI | Aplicación | Constante (en lugar de una Base de Datos *mock*)

Cree usted su propia UI que ejecute la aplicación "Discounter", ya que el código es un poco largo para incluirlo aquí. Algunas de las líneas clave del código son éstas:

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

Llegados a este punto, se puede realizar una demo de la aplicación, y también probarla con *tests* de regresión. Los adaptadores del "lado del usuario" están ambos ejecutandose.

Etapa 3: (*FIT* o UI) | Aplicación | Base de datos *mock*

Para crear un adaptador reemplazable en el "lado de la base de datos" creamos: una interfaz de un repositorio; un "RepositoryFactory" que creará la base de datos *mock*, o el objeto real de servicio; y el *mock* en memoria para la base de datos.

```
public interface RateRepository
{
   double getRate(double amount);
 }
```

```
public class RepositoryFactory
{
   public RepositoryFactory() {  super(); }
   public static RateRepository getMockRateRepository()
   {
      return new MockRateRepository();
   }
}

public class MockRateRepository implements RateRepository
{
   public double getRate(double amount)
   {
      if(amount <= 100) return 0.01;
      if(amount <= 1000) return 0.02;
      return 0.05;
    }
 }
```

Para conectar este adaptador a la aplicación "Discounter", necesitamos modificar la aplicación en sí, para que acepte un adaptador de repositorio a usar, y que el adaptador (*FIT* o UI) del lado del usuario pase el repositorio a utilizar (real o *mock*) al constructor de la aplicación. A continuación se muestra la aplicación modificada y un adaptador *FIT* que le pasa un repositorio *mock* (el código del adaptador *FIT* que elige si se le pasa el adaptador *mock* del repositorio o el adaptador real, es más extenso y no añade nueva información, de manera que omito dicha versión aquí).

```
import repository.RepositoryFactory;
import repository.RateRepository;
public class Discounter
{
   private RateRepository rateRepository;
   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(RepositoryFactory.getMockRateRepository());
   public double amount;
   public double discount()
   {
      return app.discount( amount );
   }
}
```

Esto concluye la implementación de la versión más sencilla de la arquitectura hexagonal.

Para una implementación diferente, utilizando *Ruby* y *Rack* para el uso del navegador, ver https://github.com/totheralistair/SmallerWebHexagon

Notas sobre la Aplicación del Patrón

La Asimetría Izquierda-Derecha

El patrón *Ports and Adapters* está escrito deliberadamente con la intención de que todos los puertos sean similares. Esta intención es útil a nivel de arquitectura. En cuanto a su implementación, los puertos y los adaptadores muestran dos aspectos, que llamaré "primario" y "secundario", por razones que pronto serán obvias. También se podrían llamar adaptadores "directores" y adaptadores "dirigidos".

Un lector avispado se habrá dado cuenta de que en todos los ejemplos anteriores, los *fixtures* [8] de *FIT* se usan en los puertos de la izquierda y los *mocks* en los de la derecha. En la arquitectura de tres capas, *FIT* se encuentra en la capa superior y el *mock* en la capa inferior.

Esto está relacionado con la idea de "actores primarios" y "actores secundarios" en los casos de uso. Un "actor primario" es un actor que dirige la aplicación (la saca de un estado inactivo para que lleve a cabo una de las funciones que ofrece). Un "actor secundario" es aquel al que la aplicación dirige, bien para obtener repuesta de él, bien para simplemente notificarle algo. La distinción entre "primario" y "secundario" reside en quién inicia la comunicación o es el responsable de ella.

8 El concepto de *fixture* en los frameworks de pruebas de software, se refiere a la configuración previa del entorno de ejecución de las pruebas. Por ejemplo, el *ColumnFixture* de *FIT*, o *@BeforeClass* de JUnit.

El adaptador de pruebas natural para un actor "primario" es *FIT* , ya que dicho *framework* está diseñado para leer un *script* y dirigir la aplicación. El adaptador de pruebas natural para un actor "secundario" de tipo base de datos es un *mock*, ya que está diseñado para responder consultas o grabar eventos provenientes de la aplicación.

Estas observaciones nos llevan a seguir el diagrama de contexto de casos de uso del sistema, y dibujar los "puertos primarios" y "adaptadores primarios" en la parte izquierda (o superior) del hexágono, y los "puertos secundarios" y "adaptadores secundarios" en la parte derecha (o inferior) del hexágono.

La relación entre puertos/adaptadores primarios y secundarios y su respectiva implementación con *FIT* y *mocks* es algo útil a tener en cuenta, pero debería ser considerada como una consecuencia de utilizar la arquitectura *Ports and Adapters*, no como una forma de saltársela. El beneficio fundamental de una implementación de *Ports and Adapters* es la posibilidad de ejecutar la aplicación de manera totalmente aislada.

Casos de Uso y La Frontera de la Aplicación

Utilizar el patrón de arquitectura hexagonal sirve para reforzar la manera preferente de escribir casos de uso.

Un error habitual es escribir casos de uso que contienen conocimiento de la tecnología situada fuera del puerto. Estos casos de uso se han ganado justificadamente un mal nombre en la industria por ser largos, difíciles de leer, pesados, y caros de mantener.

Entendiendo la arquitectura *Ports and Adapters*, podemos ver que los casos de uso deberían estar escritos en la frontera de la aplicación (el hexágono interno), para especificar las funciones y eventos que admite la aplicación, independientemente de la tecnología externa. Estos casos de uso son más cortos, más fáciles de leer, menos caros de mantener, y más estables a lo largo del tiempo.

¿Cuántos puertos?

Lo que es exactamente un puerto y lo que no, en gran parte, es cuestión de gustos. En un extremo, a cada caso de uso se le podría dar su propio puerto, dando lugar a cientos de puertos en muchas aplicaciones. Alternativamente, podríamos imaginar unir todos lo puertos primarios y todos los puertos secundarios, de manera que sólo haya dos puertos, un lado izquierdo y un lado derecho.

Ninguno de los extremos parece óptimo.

El sistema meteorológico descrito en el apartado “Usos Conocidos” tiene cuatro puertos naturales: la fuente suministradora de la información meteorológica, el administrador, los abonados notificados, y la base de datos de abonados. El controlador de una máquina de café tiene cuatro puertos naturales: el usuario, la base de datos de tipos de café y precios, los dispensadores, y el compartimento de modedas. Un sistema de medicación de un hospital podría tener tres: uno para la enfermera, otro para la base de datos de prescripciones, y otro para los dispensadores de medicación.

No parece que exista ningún perjuicio concreto al elegir un número “incorrecto” de puertos, de manera que sigue siendo una cuestión de intuición. Mi elección tiende a ser un número pequeño, dos, tres o cuatro puertos, tal y como se describe anteriormente y en el apartado “Usos Conocidos”.

Usos Conocidos

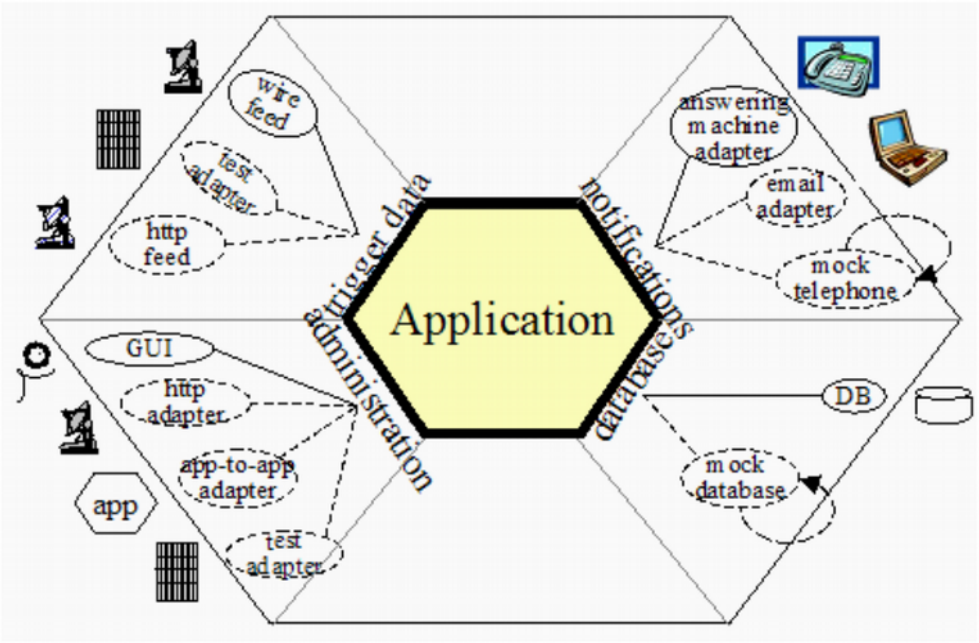

Figura 4

La figura 4 muestra una aplicación con cuatro puertos y varios adaptadores para cada puerto. Proviene de una aplicación que escuchaba alertas emitidas por el servicio meteorológico nacional acerca de terremotos, tornados, incendios e inundaciones, y notificaba a la gente en sus teléfonos o en el contestador automático. Cuando se discutió este sistema sus interfaces se identificaron por “tecnología, ligada al propósito”. Había una interfaz para los datos de entrada que llegan por cable, una para la notificación de datos que serán enviados a contestadores automáticos, una interfaz de administración implementada en una GUI, y una interfaz de base de datos para recuperar los datos de los abonados.

Se estaba realizando un gran esfuerzo, debido a que se necesitaba añadir una interfaz *http* para el servicio meteorológico, una interfaz de email para sus abonados, y se tenía que encontrar una forma de empaquetar y desempaquetar la *suite* de la aplicación, que iba creciendo cada vez más, para las distintas preferencias de compra de los clientes. Se temía estar enfrentándose a una pesadilla de mantenimiento y pruebas, ya que se tenían que implementar, probar y mantener versiones separadas para todas las combinaciones y permutaciones.

El cambio en el diseño fue organizar las interfaces del sistema "por propósito" en vez de por tecnología, y hacer que las tecnologías fuesen sustituibles (en todos los lados del hexágono) por adaptadores. Enseguida se adquirió la habilidad necesaria para incluir la recepción de los datos vía *http* y la notificación por email (los nuevos adaptadores se muestran en la figura con línea discontinuas). Haciendo que cada aplicación fuese ejecutable en modo *headless* mediante APIs, se pudo añadir un adaptador aplicación-a-aplicación y desempaquetar la *suite* de la aplicación, conectando las sub-aplicaciones bajo demanda. Finalmente, haciendo que cada aplicación fuese ejecutable de manera completamente aislada, con adaptadores *test* y *mock* , se adquirió la capacidad de aplicar pruebas de regresión a las aplicaciones, con *scripts* autónomos de pruebas automatizadas.

Mac, Windows, Google, Flickr, Web 2.0

A principios de los 90, se requería que aplicaciones *MacIntosh* tales como procesadores de texto tuvieran interfaces API, de manera que aplicaciones y *scripts* escritos por el usuario pudieran acceder a toda la funcionalidad de las aplicaciones. Las aplicaciones *Windows* de escritorio han evolucionado para tener la misma capacidad (no tengo conocimiento histórico para decir cuál fue primero, ni es relevante en este caso).

La tendencia actual (2005) en aplicaciones web es publicar un API y permitir a otras aplicaciones web acceder a dichas APIs directamente. Así, es posible publicar datos sobre la criminalidad local en un mapa de *Google* , o crear aplicaciones web que sean capaces de archivar y realizar anotaciones en fotos de *Flickr*.

Todos estos ejemplos tratan de hacer visibles las APIs de los "puertos primarios". No vemos ninguna información aquí sobre los puertos secundarios.

Salida Almacenada

Willem Bogaerts escribió el siguiente ejemplo en la wiki C2:

"Encontré algo similar, pero principalmente porque mi capa de aplicación tenía una fuerte tendencia a terminar convirtiéndose en una centralita telefónica que gestionara cosas que no debería. Mi aplicación generaba una salida, la mostraba al usuario y después tenía posibilidad de almacenarla también. Mi principal problema era que no se necesitaba guardarla siempre. Entonces, mi aplicación generaba una salida, tenía que almacenarla temporalmente (en un *buffer*) y presentarla al usuario. Después, cuando el usuario decidía que quería guardar la salida, la aplicación recuperaba el *buffer* y la almacenaba de manera definitiva.

Esto no me gustaba en absoluto. Entonces encontré una solución: Tener un control de presentación con posibilidad de almacenamiento. Ahora la aplicación no canalizaría más la salida en diferentes direcciones, sino que simplemente la enviaría al control de presentación. Es el control de presentación el que guardaría temporalmente la respuesta y daría al usuario la posibilidad de almacenarla definitivamente.

La arquitectura tradicional en capas enfatiza que la 'UI' y el 'almacenamiento' son distintos. La arquitectura *Ports and Adapters* puede hacer que la salida sea simplemente 'salida' otra vez."

Ejemplo Anónimo de la Wiki C2

"En un proyecto en el que trabajé, utilizábamos la 'Metáfora de Sistema' de un sistema estéreo formado por componentes. Cada componente tiene interfaces definidas, cada una de ellas con un propósito específico. Así podemos conectar los componentes entre sí de innumerables formas, usando simples cables y adaptadores."

Desarrollo con un Equipo Amplio Distribuido

Todavía está en fase de pruebas así que no cuenta como un uso propiamente dicho del patrón. Sin embargo es interesante considerarlo.

Equipos ubicados en diferentes lugares utilizan todos la Arquitectura Hexagonal, usando *FIT* y *mocks* de manera que las aplicaciones o componentes se pueden probar aisladamente. *CruiseControl* construye los ejecutables cada media hora y ejecuta todas las aplicaciones usando la combinación *" FIT + mock "*. Conforme se van completando los subsistemas de la aplicación y las bases de datos, los *mocks* se reemplazan por bases de datos de pruebas.

Desarrollo por Separado de la UI y de la Lógica de Aplicación

Todavía está en una fase temprana de pruebas así que no cuenta como un uso del patrón. Sin embargo es interesante considerarlo.

El diseño de la UI es inestable, ya que aún no se ha decidido acerca de una tecnología directora o una metáfora. La arquitectura de los servicios *back-end* no se ha decidido, y de hecho cambiará probablemente varias veces a lo largo de los seis próximos meses. En todo caso, el proyecto ha comenzado oficialmente y el tiempo corre.

El equipo crea pruebas *FIT* y *mocks* para aislar su aplicación, y crea funcionalidad que se pueda probar, para mostrarla a sus usuarios. Cuando las decisiones acerca de la UI y de los servicios *back-end* finalmente encajen, debería ser trivial añadir dichos elementos a la aplicación. Permanezca atento para aprender cómo se resuelve este asunto (o inténtelo usted mismo y escríbame para hacérmelo saber).

Patrones Relacionados

Adaptador

El libro *"Dessign Patterns"* contiene una descripción del patrón "Adaptador" genérico: "Convierte la interfaz de una clase en otra interfaz que el cliente espera". El patrón *Ports and Adapters* es un uso particular del patrón "Adaptador".

Modelo-Vista-Controlador

El patrón MVC fue implementado muy pronto, en 1974, en el proyecto Smalltalk. Ha tenido muchas variantes a lo largo de los años, como por ejemplo "Modelo-Interactor" y "Modelo-Vista-Presentador". Cada una de ellas implementa la idea de *Ports and Adapters* en los puertos primarios, no en los secundarios.

"Objetos Mock" y *"Loopback"*

"Un objeto *'mock'* es un 'agente doble' usado para probar el comportamiento de otros objetos. En primer lugar, un objeto *'mock'* actúa como una implementación falsa de una interfaz o clase que imita el comportamiento externo de una implementación real. En segundo lugar, un objeto *'mock'* observa cómo otros objetos interactúan con sus métodos y compara el comportamiento efectivo con las expectativas

prefijadas. Cuando se da una discrepancia, un objeto *'mock'* puede interrumpir la prueba e informar de la anomalía. Si la discrepancia no puede ser descubierta durante la prueba, un método de verificación invocado por el realizador de la misma asegura que todas las expectativas se han cumplido o se ha informado de los fallos." - de *http://MockObjects.com* [9]

Implementándolos cumpliendo completamente con la agenda *mock-object* , los objetos *mock* se usan en toda la aplicación, no sólo en la interfaz externa. La idea clave del movimiento *mock-object* es el cumplimiento del protocolo especificado a nivel de clases individuales y objetos. Tomo prestada su palabra *"mock"* como la mejor descripción breve de "sustituto en memoria de un actor secundario externo".

El patrón *Loopback* es un patrón explícito para crear una sustitución interna de un dispositivo externo.

Pedestal

En *"Patterns for Generating a Layered Architecture"* , Barry Rubel describe un patrón para crear un eje de simetría en software de control que es muy similar a *Ports and Adapters* . El patrón *"Pedestal"* propugna implementar un objeto que represente cada dispositivo hardware de un sistema, y enlazar dichos objetos entre sí en una capa de control. El patrón *"Pedestal"* se puede usar para describir cualquiera de los dos lados de la arquitectura hexagonal, pero no redunda en la similitud entre adaptadores. Además, al haber sido escrito para un entorno de control mecánico, no es tan fácil ver cómo aplicar el patrón a aplicaciones de IT [10].

Checks

El lenguaje del patrón de Ward Cunningham para detectar y gestionar errores de usuario en la entrada de datos, es adecuado para la gestión de errores en la frontera del hexágono interno.

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) establece que:

9 Enlace roto. El enlace actual (Junio 2018) es el siguiente: http://www.mockobjects.com

10 Siglas en inglés de "Information Technology" (Tecnología de la Información).

"Los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones."

El patrón de "Inyección de Dependencias" de Martin Fowler aporta algunas implementaciones, las cuales muestran cómo crear adaptadores intercambiables para los actores secundarios. El código se puede escribir directamente, como se hace en el código de ejemplo del artículo, o usando archivos de configuración y dejando que el *framework* SPRING genere el código equivalente.

Agradecimientos

Gracias a Gyan Sharma, de *"Intermountain Health Care"* , por facilitar el código de ejemplo utilizado aquí. Gracias a Rebecca Wirfs-Brock por su libro *"Object Design"* , que leído junto al patrón "Adaptador" del libro *"Design Patterns"* , me ayudó a comprender de qué trataba el hexágono. Gracias también a las personas que en la wiki de Ward aportaron comentarios sobre este patrón a lo largo de los años (por ejemplo, particularmente el de Kevin Rutherford: http://silkandspinach.net/blog/2004/07/hexagonal_soup.html).

Referencias y Lecturas relacionadas

* ***FIT, A Framework for Integrating Testing*** : Cunningham, W., online en http://fit.c2.com , y Mugridge, R. y Cunningham, W., ***"Fit for Developing Software"*** , Prentice-Hall PTR, 2005.

* ***The "Adapter" pattern*** : en Gamma, E., Helm, R., Johnson, R., Vlissides, J., ***"Design Patterns"*** , Addison-Wesley, 1995, pp. 139-150.

* ***The "Pedestal" pattern*** : en Rubel, B., ***"Patterns for Generating a Layered Architecture"*** , en Coplien, J., Schmidt, D., ***"PatternLanguages of Program Design"*** , Addison-Wesley, 1995, pp. 119-150.

* ***The "Checks" pattern*** : de Cunningham, W., online en http://c2.com/ppr/checks.html

* The "Dependency Inversion Principle" ***"Agile Software Development Principles Patterns and Practices"***, Prentice Hall, 2003, ***Chapter 11: "The Dependency-Inversion Principle"***, y online en *http://www.objectmentor.com/resources/articles/dip.pdf* [11]

* ***The "Dependency Injection" pattern*** : Fowler, M., online en http://www.martinfowler.com/articles/injection.html

* ***The "Mock Object" pattern*** : Freeman, S. online en *http://MockObjects.com* [12]

* ***The "Loopback" pattern*** : Cockburn, A., online en http://c2.com/cgi/wiki?LoopBack

* "Use cases:" Cockburn, A., ***"Writing Effective Use Cases"*** , Addison-Wesley, 2001, y Cockburn, A., ***"Structuring Use Cases with Goals"*** , online en *http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm* [13]

11 Enlace roto.

12 Enlace roto. El enlace actual (Junio 2018) es el siguiente: http://www.mockobjects.com

13 Enlace roto. Actualmente (Junio 2018) se desconoce si será migrado.

ANEXO: NOTAS SOBRE LA TRADUCCIÓN

(*) El patrón "Ports and Adapters" es obra del **Dr. Alistair Cockburn**, coautor del "Agile Manifesto", que allá por el año 2005 escribió un articulo donde definía el patrón. Ésta es la traducción de dicho artículo. Mi agradecimiento al Sr. Cockburn por permitirme traducirlo.

(*) La URL del artículo original en inglés era http://alistair.cockburn.us/Hexagonal+architecture . Actualmente (Junio 2018) el sitio web está caído y se desconoce si será migrado a otro sitio.

(*) Hay ciertos términos que deliberadamente no han sido traducidos, bien por ser considerados nombres propios (como por ejemplo los nombres de libros), bien por ser términos informáticos que en español se usan habitualmente tal cual (como por ejemplo batch, framework, mock, etc).

(*) El artículo original, en su apartado "Intent", contiene la frase:

"As events arrive from the outside world at a port, a technology-specific adapter converts it into a usable procedure call or message and passes it to the application"

La cual puede dar lugar a confusión, porque podría entenderse que el dibujo de la arquitectura fuese:

Actor Primario → Puerto Primario → Adaptador Primario → Hexágono

Cuando el dibujo correcto es:

Actor Primario → Adaptador Primario → Puerto Primario (API del Hexágono)

Tras preguntarle sobre ello, el Sr. Cockburn me dió otra versión de la citada frase. La reescribió de la siguiente forma:

"As an event arrives from the outside world, it gets converted by a technology-specific adapter to a usable procedure call or message at the port"

La cual encaja con la arquitectura. Es por tanto esta frase la que he traducido, y queda así:

"Cuando llega un evento del mundo exterior, un adaptador dependiente de la tecnología lo convierte en una llamada a un procedimiento utilizable, o en un mensaje, de un puerto"

Autor de la Traducción: Juan Manuel Garrido de Paz

Ingeniero en Informática

Fecha de la Traducción: 24 de Junio de 2018