CAPITULO 6. OBJETOS CON BUEN CMPORTAMIENTO

6.1 Introducción

La **prueba** es la actividad cuyo objetivo es determinar si una pieza de código (un método, una clase o un programa) produce el comportamiento pretendido.

La **depuración** es el intento de apuntar con precisión y corregir un error en el código. La *depuración* viene a continuación de la prueba. Si las pruebas demostraron que se presentó un error, usamos técnicas de depuración para encontrar exactamente dónde está ese error y corregirlo. Puede haber una cantidad significativa de trabajo entre saber que existe un error y encontrar su causa y solucionarlo.

Probablemente, el punto más fundamental se centra en escribir código con fines de mantenimiento. Se trata de escribir código de tal manera que, en primer término, se eviten los errores, y si aun así aparecen, puedan ser encontrados lo más fácilmente posible. Esto está fuertemente relacionado con el estilo de código y los comentarios.

6.3 Pruebas de unidad

El término **prueba de unidad** se refiere a la prueba de partes individuales de una aplicación en contraposición con el término **prueba de aplicación** que es la prueba de una aplicación en su totalidad.

Las unidades que se prueban pueden ser de tamaños diversos: puede ser un grupo de clases, una sola clase o simplemente un método. Debemos observar que la prueba de unidad puede escribirse mucho antes de que una aplicación esté completa.

La experimentación y prueba temprana conlleva varios beneficios. En primer lugar, nos dan una experiencia valiosa con un sistema que hace posible localizar problemas tempranamente para corregirlos, a un costo mucho menor que si se hubieran encontrado en una etapa más avanzada del desarrollo. En segundo término, podemos comenzar por construir una serie de casos de prueba y resultados que pueden usarse una y otra vez a medida que el sistema crece. Cada vez que hacemos un cambio en un sistema, estas pruebas nos permiten controlar que no hayamos introducido errores inadvertidamente en el resto del sistema como resultado de las modificaciones.

6.3.2 Pruebas positivas y pruebas negativas

En una aplicación, cuando tenemos que decidir qué parte probar, generalmente distinguimos los casos de **pruebas positivas** de los casos de **pruebas negativas**.

Una **prueba positiva** es la prueba de aquellos casos que esperamos que resulten exitosos. Cuando probamos con casos positivos nos tenemos que convencer de que el código realmente funciona como esperábamos.

Una prueba negativa es la prueba de aquellos casos que esperamos que fallen. Cuando probamos con casos negativos esperamos que el programa maneje este error de cierta manera especificada y controlada.

6.4 Pruebas automatizadas

Existen técnicas disponibles que nos permiten automatizar las pruebas repetitivas y así eliminar el trabajo pesado asociado que traen aparejadas.

6.4.1 Prueba de regresión

Cuando se soluciona un error en un lugar determinado se puede, al mismo tiempo, introducir un nuevo error. Es deseable ejecutar **pruebas de regresión** cada vez que se realiza una modificación en el software. Las pruebas de regresión consisten en ejecutar nuevamente las pruebas pasadas previamente para asegurarse de que la nueva versión aún las pasa. Probablemente, estas pruebas son mucho más realizables cuando se las puede automatizar de alguna manera. Una de las formas más fáciles de automatizar las pruebas de regresión es escribir un programa que actúa como un **equipo de pruebas** o una **batería de pruebas**.

6.4.2 Control automático de los resultados de las pruebas

Las clases de prueba son una característica de BlueJ y están diseñadas para implementar pruebas de regresión. Se basan en el marco de trabajo para pruebas JUnit creado.

Las clases de prueba, en cierto sentido, son claramente diferentes de las clases ordinarias. El código de la clase de prueba puede ser escrito por una persona, o fue *generado automáticamente* por BlueJ.

6.4.3 Grabar una prueba

BlueJ posibilita combinar la efectividad de las pruebas manuales con el poder de las pruebas automatizadas habilitándonos para grabar las pruebas manuales y luego ejecutarlas, con el fin de aplicar pruebas de regresión.

Una **aserción** es una expresión que establece una condición que esperamos que resulte verdadera. Si la condición es falsa, decimos que falló esta aserción que indica que hay un error en nuestro programa.

6.4.4 Objetos de prueba

A **fixture** es un conjunto de objetos con un estado definido que sirve como base para las pruebas de unidades. En BlueJ se usa *Test Fixture to Object Bench*, para crear automáticamente los objetos de prueba a partir del código o *Object Bench lo Test Fixture*, para añadir al código los objetos de prueba.

La automatización de pruebas es un concepto poderoso porque hace más probable que las pruebas se escriban en primer lugar y más probable que se ejecuten y reejecuten a medida que el programa se desarrolle. Podría formarse el hábito de comenzar por escribir pruebas de unidad tempranamente en el desarrollo de un proyecto y mantenerlas actualizadas a medida que el proyecto avance.

6.5 Modularización e interfaces

La modularización es crucial en cualquier proyecto en el que diferentes personas implementen varios componentes del mismo. Sin embargo, no alcanza sólo con dividir una tarea en varias clases, además, debe haber directivas claras para las diferentes implementaciones que indiquen qué deben hacer y cómo encajan todos los componentes en la aplicación final.

Cuando varios componentes de un software colaboran para completar una misma tarea decimos que la *interfaz* entre ellos debe ser clara y bien definida. Por interfaz entendemos aquellas partes de una clase que se conocen y que se utilizan en otras clases.

Cuando dos módulos se desarrollen simultáneamente, con frecuencia será necesario definir la interfaz antes de comenzar a trabajar sobre la implementación de cada uno. Esto puede hacerse, generalmente, mediante las signaturas de los métodos porque proporcionan suficiente información de una clase sobre cómo interactuar con otra sin necesidad de saber cómo están implementados dichos métodos. Este es un concepto importante. Tratamos, tanto como sea posible, de separar las interfaces de las clases de los detalles de implementación.

La interfaz representa una especie simple de contrato entre una clase y otras partes del programa que la usarán. La interfaz describe un conjunto mínimo de métodos que serán implementados en el componente lógico y para cada método están completamente definidos su tipo de retorno y sus parámetros. Observe que la interfaz no brinda detalles sobre lo que hará su implementación internamente; estos detalles quedan en manos de sus implementadores. Además, la implementación de la clase podría contener otros métodos que no aparecen en este listado.

6.7 Comentarios y estilo

Buenas costumbres en el uso de comentarios pueden ser:

Comentar la clase con un comentario multilínea en la parte superior indicando el propósito de la misma. Incluir anotaciones indicando el autor y el número de versión.

Cada método de la interfaz debe tener un comentario que indica su propósito, sus parámetros y su tipo de retorno. Ciertamente, estos comentarios facilitarán la generación de documentación para la interfaz.

El esquema de la clase debe ser consistente, con cantidades adecuadas de espacios en blanco para la indentación que usa para distinguir los niveles de los bloques anidados y las estructuras de control.

Las variables deben tener nombres significativos y los nombres de los métodos deben ser bien elegidos.

Pese a que estas convenciones parecen consumir demasiado tiempo durante la implementación, pueden redundar en un beneficio enorme para ayudar a otro a comprender el código o en ayudar a recordar qué hace una clase si dejamos de trabajar un tiempo en ella.

6.8 Seguimiento manual

Un **seguimiento manual** o **prueba de escritorio** es la actividad en la que trabajamos sobre un segmento de código línea por línea mientras se observan los cambios de estado y otros comportamientos de la aplicación.

Un seguimiento manual involucra tanto la lectura de clases como el seguimiento del control del flujo entre las clases y los objetos. Ayuda a la comprensión tanto de las maneras en que interactúan los objetos unos con otros como de la forma en que se comportan internamente. En efecto, un seguimiento manual es una simulación en papel y lápiz de lo que ocurre dentro de la computadora cuando se ejecuta un programa. En la práctica, es la mejor forma de concentrarse en una porción pequeña de la aplicación, tal como un grupo lógico de acciones o la llamada a un método.

Uno de los requerimientos cruciales de la depuración exitosa es ser capaz de disparar fácilmente el error que estamos buscando y reproducirlo, por este camino es mucho más fácil de evaluar el efecto de un intento de corrección.

6.8.2 Controlar el estado mediante el seguimiento

Los objetos completamente pasivos no inician ninguna actividad por sí mismos, sino que simplemente responden a invocaciones externas de métodos. Este es el estilo de comportamiento típico de un servidor. Con frecuencia, los objetos servidores descansan fuertemente sobre su propio estado para determinar cómo deben responder a las llamadas de métodos. Por lo que, una parte importante al conducir el seguimiento es estar seguro de que siempre disponemos de una representación exacta de su estado. Una forma de hacer esto en papel y lápiz es construyendo una tabla de los campos del objeto y sus valores.

6.8.3 Seguimiento verbal

El **seguimiento verbal** consiste en tratar de explicar a otra persona lo que hace una clase o un método. Esta forma funciona de dos maneras completamente diferentes:

La persona a la que le explica el código podría encontrar el error por usted.

Encontrará con frecuencia que el simple hecho de tratar de poner en palabras lo que debiera hacer una pieza de código es suficiente para activar en su mente una comprensión del por qué no lo hace.

6.9 Sentencias de impresión

La técnica más común usada para comprender y depurar un programa, aun por programadores experimentados, es agregar en los métodos sentencias de impresión temporalmente. Las sentencias de impresión son populares porque existen en la mayoría de los lenguajes, están disponibles para todos y son muy fáciles de agregar mediante un editor.

Cuando se ejecuta un programa, estas sentencias de impresión adicionales proveen al usuario de información tal como:

- qué métodos se han invocado;
- los valores de los parámetros;
- el orden en que se han invocado los métodos;
- los valores de las variables locales y de los campos en lugares estratégicos.

Las sentencias de impresión pueden ser muy efectivas para ayudarnos a comprender los programas o para ubicar errores, pero existen algunas desventajas:

- Generalmente, no es muy práctico agregar sentencias de impresión a cada método de una clase. Por lo que sólo son completamente efectivas si se agregan en los métodos correctos.

- Agregar demasiadas sentencias de impresión puede llevarnos a perder de vista información. En una cantidad muy grande de información de salida es muy difícil identificar lo que necesitamos ver. En particular, las sentencias de impresión dentro de los ciclos traen aparejados estos problemas.

- Una vez que cumplieron con su propósito, puede resultar tedioso eliminarlas.

- Existe también la posibilidad de que habiéndolas eliminado, resulten nuevamente necesarias. ¡Puede ser muy frustrante tener que agregarlas nuevamente!

6.9.1 Activar o desactivar la información de depuración

Si una clase todavía se encontraba en desarrollo cuando se le agregaron sentencias de impresión, generalmente no queremos ver esta salida cada vez que se use la clase. Es mejor que podamos encontrar una manera de activar la impresión o desactivarla, según necesitemos. La forma más común de llevar esto a cabo es agregar un campo lógico (boolean) a la clase y luego hacer que la impresión dependa del valor de este campo.

Una variante más económica de este tema consiste en reemplazar las llamadas directas a sentencias de impresión por invocaciones a los métodos de impresión agregados a la c1ase. El método de impresión sólo imprimirá si el campo depuracion es verdadero (true).

6.10 Elegir una estrategia de prueba

En la práctica, podríamos usar estrategias diferentes en momentos diferentes. Los seguimientos, las sentencias de impresión y las pruebas interactivas son útiles para la prueba inicial de clases recién escritas o para investigar cómo funciona un segmento de un programa. Su ventaja es que estas técnicas son rápidas y fáciles de usar, funcionan bien en cualquier lenguaje de programación y son independientes del entorno (excepto las pruebas interactivas). La principal desventaja es que estas pruebas no se pueden repetir fácilmente más adelante para realizar pruebas de regresión.

El uso de clases de pruebas de unidad tiene la ventaja, una vez que se las construyó, de que las pruebas se pueden ejecutar cualquier número de veces.

6.11 Depuradores

El depurador es esencialmente una herramienta de software que proporciona apoyo para realizar un seguimiento de un segmento de código. Típicamente fijamos puntos de interrupción en las sentencias en donde queremos comenzar nuestro seguimiento y luego usamos las funciones *Step* y *Step lnto* para llevarlo a cabo.

Una de las ventajas es que el depurador automáticamente tiene el cuidado de mantener el trazo del estado de cada objeto y al hacer esto, es más rápido y produce menos errores que cuando hacemos lo mismo manualmente. Una desventaja de los depuradores es que no mantienen registro permanente de los cambios de estado por lo que resulta difícil volver atrás y controlar el estado en que estaba unas cuantas sentencias antes.

Un depurador, típicamente, ofrece información sobre la *secuencia de llamadas* (o pila de llamadas o *stack)* en cada momento. La secuencia de llamadas contiene un registro de todos los métodos activos y aún no terminados.