gador web, al igual que hace cualquier programa ordinario. Aquí, la fuerza de Java no es sólo su portabilidad, sino también su programabilidad (facilidad de programación). Como se verá a lo largo del presente libro, Java tiene muchos aspectos que permiten la creación de programas robustos en un período de tiempo menor al que requerían los lenguajes de programación anteriores.

Uno debe ser consciente de que esta bendición no lo es del todo. El precio a pagar por todas estas mejoras es una velocidad de ejecución menor (aunque se está haciendo bastante trabajo en este área —JDK 1.3, en particular, presenta las mejoras de rendimiento denominadas "*hotspot*"). Como cualquier lenguaje, Java tiene limitaciones intrínsecas que podrían hacerlo inadecuado para resolver cierto tipo de problemas de programación. Java es un lenguaje que evoluciona rápidamente, no obstante, y cada vez que aparece una nueva versión, se presenta más y más atractivo de cara a la solución de conjuntos mayores de problemas.

Análisis y diseño

El paradigma de la orientación a objetos es una nueva manera de enfocar la programación. Son muchos los que tienen problemas a primera vista para enfrentarse a un proyecto de POO. Dado que se supone que todo es un objeto, y a medida que se aprende a pensar de forma orientada a objetos, es posible empezar a crear "buenos" diseños y sacar ventaja de todos los beneficios que la POO puede ofrecer.

Una *metodología* es un conjunto de procesos y heurísticas utilizadas para descomponer la complejidad de un problema de programación. Se han formulado muchos métodos de POO desde que enunció la programación orientada a objetos. Esta sección presenta una idea de lo que se trata de lograr al utilizar un método.

Especialmente en la POO, la metodología es un área de intensa experimentación, por lo que es importante entender qué problema está intentando resolver el método antes de considerar la adopción de uno de ellos. Esto es particularmente cierto con Java, donde el lenguaje de programación se ha desarrollado para reducir la complejidad (en comparación con C) necesaria para expresar un programa. Esto puede, de hecho, aliviar la necesidad de metodologías cada vez más complejas. En vez de esto, puede que las metodologías simples sean suficientes en Java para conjuntos de problemas mucho mayores que los que se podrían manipular utilizando metodologías simples con lenguajes procedimentales.

También es importante darse cuenta de que el término "metodología" es a menudo muy general y promete demasiado. Se haga lo que se haga al diseñar y escribir un programa, se sigue un método. Puede que sea el método propio de uno, e incluso puede que uno no sea consciente de utilizarlo, pero es un proceso que se sigue al crear un programa. Si el proceso es efectivo, puede que simplemente sea necesario afinarlo ligeramente para poder trabajar con Java. Si no se está satisfecho con el nivel de productividad y la manera en que se comportan los programas, puede ser buena idea considerar la adopción de un método formal, o la selección de fragmentos de entre los muchos métodos formales existentes.

Mientras se está en el propio proceso de desarrollo, el aspecto más importante es no perderse, aunque puede resultar fácil. Muchos de los métodos de análisis y desarrollo fueron concebidos para re-

solver los problemas más grandes. Hay que recordar que la mayoría de proyectos no encajan en esta categoría, siendo posible muchas veces lograr un análisis y un diseño con éxito con sólo un pequeño subconjunto de los que el método recomienda[10]. Pero algunos tipos de procesos, sin importar lo limitados que puedan ser, le permitirán encontrar el camino de manera más sencilla que si simplemente se empieza a codificar.

También es fácil desesperarse, caer en "parálisis de análisis", cuando se siente que no se puede avanzar porque no se han cubierto todos los detalles en la etapa actual. Debe recordarse que, independientemente de cuánto análisis lleve a cabo, hay cosas de un sistema que no aparecerán hasta la fase de diseño, y otras no aflorarán incluso hasta la fase de codificación o en un extremo, hasta que el programa esté acabado y en ejecución. Debido a esto, es crucial moverse lo suficientemente rápido a través de las etapas de análisis y diseño e implementar un prototipo del sistema propuesto.

Debe prestarse un especial énfasis a este punto. Dado que ya se conoce la misma historia con los lenguajes *procedimentales*, es recomendable que el equipo proceda de manera cuidadosa y comprenda cada detalle antes de pasar del diseño a la implementación. Ciertamente, al crear un SGBD, esto pasa por comprender completamente la necesidad del cliente. Pero un SGBD es la clase de problema bien formulada y bien entendida; en muchos programas, *es* la estructura de la base de datos la que debe ser desmenuzada. La clase de problema de programación examinada en el presente capítulo es un "juego de azar"[11], en el que la solución no es simplemente la formulación de una solución bien conocida, sino que involucra además a uno o más "factores de azar" —elementos para los que no existe una solución previa bien entendida, y para los cuales es necesario algún tipo de proceso de investigación[12]. Intentar analizar completamente un problema al azar antes de pasar al diseño e implementación conduce a una parálisis en el análisis, al no tener suficiente información para resolver este tipo de problemas durante la fase de análisis. Resolver un problema así, requiere iterar todo el ciclo, y precisa de un comportamiento que asuma riesgos (lo cual tiene sentido, pues está intentando hacer algo nuevo y las recompensas potenciales crecen). Puede parecer que el riesgo se agrava al precipitarse hacia una implementación preliminar, pero ésta puede reducir el riesgo en los problemas al azar porque se está averiguando muy pronto si un enfoque particular al problema es o no viable. El desarrollo de productos conlleva una gestión del riesgo.

A menudo, se propone "construir uno para desecharlo". Con POO es posible tirar *parte*, pero dado que el código está encapsulado en clases, durante la primera pasada siempre se producirá algún diseño de clases útil y se desarrollarán ideas que merezcan la pena para el diseño del sistema de las que no habrá que deshacerse. Por tanto, la primera pasada rápida por un problema no sólo suministra información crítica para las ulteriores pasadas por análisis, diseño e implementación, sino que también crea la base del código.

---

[10] Un ejemplo excelente de esto es *UML Distilled*, 2.ª edición, de Martin Fowler (Addison-Wesley 2000), que reduce el proceso, en ocasiones aplastante, a un subconjunto manejable (existe versión española con el título *UML gota a gota*).

[11] N. del traductor: Término *wild-card*, acuñado por el autor original.

[12] Regla del pulgar —acuñada por el autor— para estimar este tipo de proyectos: si hay más de un factor al azar, ni siquiera debe intentarse planificar la duración o el coste del proyecto hasta que no se ha creado un prototipo que funcione. Existen demasiados grados de libertad.

Dicho esto, si se está buscando una metodología que contenga un nivel de detalle tremendo, y sugiera muchos pasos y documentos, puede seguir siendo difícil saber cuándo parar. Debe recomendarse lo que se está intentando descubrir.

1. ¿Cuáles son los objetos? (¿Cómo se descompone su proyecto en sus componentes?)

2. ¿Cuáles son las interfaces? (¿Qué mensajes es necesario enviar a cada objeto?)

Si se delimitan los objetos y sus interfaces, ya es posible escribir un programa. Por diversas razones, puede que sean necesarias más descripciones y documentos que éste, pero no es posible avanzar con menos.

El proceso puede descomponerse en cinco fases, y la Fase 0 no es más que la adopción de un compromiso para utilizar algún tipo de estructura.

Fase 0: Elaborar un plan

En primer lugar, debe decidirse qué pasos debe haber en un determinado proceso. Suena bastante simple (de hecho, *todo* esto suena simple) y la gente no obstante, suele seguir sin tomar esta decisión antes de empezar a codificar. Si el plan consiste en "empecemos codificando", entonces, perfecto (en ocasiones, esto es apropiado, si uno se está enfrentando a un problema que conoce perfectamente). Al menos, hay que estar de acuerdo en que eso también es tener un plan.

También podría decidirse en esta fase que es necesaria alguna estructura adicional de proceso, pero no toda una metodología completa. Para que nos entendamos, a algunos programadores les gusta trabajar en "modo vacación", en el que no se imponga ninguna estructura en el proceso de desarrollar de su trabajo; "se hará cuando se haga". Esto puede resultar atractivo a primera vista, pero a medida que se tiene algo de experiencia uno se da cuenta de que es mejor ordenar y distribuir el esfuerzo en distintas etapas en vez de lanzarse directamente a "finalizar el proyecto". Además, de esta manera se divide el proyecto en fragmentos más asequibles, y se resta miedo a la tarea de enfrentarse al mismo (además, las distintas fases o hitos proporcionan más motivos de celebración).

Cuando empecé a estudiar la estructura de la historia (con el propósito de acabar escribiendo algún día una novela), inicialmente, la idea que más me disgustaba era la de la estructura, pues parecía que uno escribe mejor si simplemente se dedica a rellenar páginas. Pero más tarde descubrí que al escribir sobre computadores, tenía la estructura tan clara que no había que pensar demasiado en ella. Pero aún así, el trabajo se estructuraba, aunque sólo fuera semiconscientemente en mi cabeza. Incluso cuando uno piensa que el plan consiste simplemente en empezar a codificar, todavía se atraviesan algunas fases al plantear y contestar ciertas preguntas.

El enunciado de la misión

Cualquier sistema que uno construya, independientemente de lo complicado que sea, tiene un propósito fundamental: el negocio intrínseco en el mismo, la necesidad básica que cumple. Si uno puede mirar a través de la interfaz de usuario, a los detalles específicos del hardware o del sistema, los algoritmos de codificación y los problemas de eficiencia, entonces se encuentra el centro de su existencia —simple y directo. Como el denominado alto concepto (*high concept*) en las películas de

Hollywood, uno puede describir el propósito de un programa en dos o tres fases. Esta descripción, pura, es el punto de partida.

El alto concepto es bastante importante porque establece el tono del proyecto; es el enunciado de su misión. Uno no tiene por qué acertar necesariamente a la primera (puede ser que uno esté en una fase posterior del problema cuando se le ocurra el enunciado completamente correcto) pero hay que seguir intentándolo hasta tener la certeza de que está bien. Por ejemplo, en un sistema de control de tráfico aéreo, uno puede comenzar con un alto concepto centrado en el sistema que se está construyendo: "El programa de la torre hace un seguimiento del avión". Pero considérese qué ocurre cuando se introduce el sistema en un pequeño aeródromo; quizás sólo hay un controlador humano, o incluso ninguno. Un modelo más usual no abordará la solución que se está creando como describe el problema: "Los aviones llegan, descargan, son mantenidos y recargan, a continuación, salen".

Fase 1: ¿Qué estamos construyendo?

En la generación previa del diseño del programa (denominada *diseño procedural*) a esta fase se le denominaba "creación del *análisis de requisitos* y *especificación del sistema*". Éstas, por supuesto, eran fases en las que uno se perdía; documentos con nombres intimidadores que podían de por sí convertirse en grandes proyectos. Sin embargo, su intención era buena. El análisis de requisitos dice: "Construya una lista de directrices que se utilizarán para saber cuándo se ha acabado el trabajo y cuándo el cliente está satisfecho". La especificación del sistema dice: "He aquí una descripción de lo *que* el programa hará (pero no *cómo*) para satisfacer los requisitos hallados". El análisis de requisitos es verdaderamente un contrato entre usted y el cliente (incluso si el cliente trabaja en la misma compañía o es cualquier otro objeto o sistema). La especificación del sistema es una exploración de alto nivel en el problema, y en cierta medida, un descubrimiento de si puede hacerse y cuánto tiempo llevará. Dado que ambos requieren de consenso entre la gente (y dado que generalmente variarán a lo largo del tiempo) lo mejor es mantenerlos lo más desnudos posible —idealmente, tratará de listas y diagramas básicos para ahorrar tiempo. Se podría tener otras limitaciones que exijan expandirlos en documentos de mayor tamaño, pero si se mantiene que el documento inicial sea pequeño y conciso, es posible crearlo en unas pocas sesiones de tormenta de ideas (*brainstorming*) en grupo, con un líder que dinámicamente va creando la descripción. Este proceso no sólo exige que todos aporten sus ideas sino que fomenta el que todos los miembros del equipo lleguen a un acuerdo inicial. Quizás lo más importante es que puede incluso ayudar a que se acometa el proyecto con una gran dosis de entusiasmo.

Es necesario mantenerse centrado en el corazón de lo que se está intentando acometer en esta fase: determinar qué es lo que se supone que debe hacer el sistema. La herramienta más valiosa para esto es una colección de lo que se denomina "casos de uso". Los casos de uso identifican los aspectos claves del sistema, que acabarán por revelar las clases fundamentales que se usarán en éste. De hecho, los casos de uso son esencialmente soluciones descriptivas a preguntas como[13]:

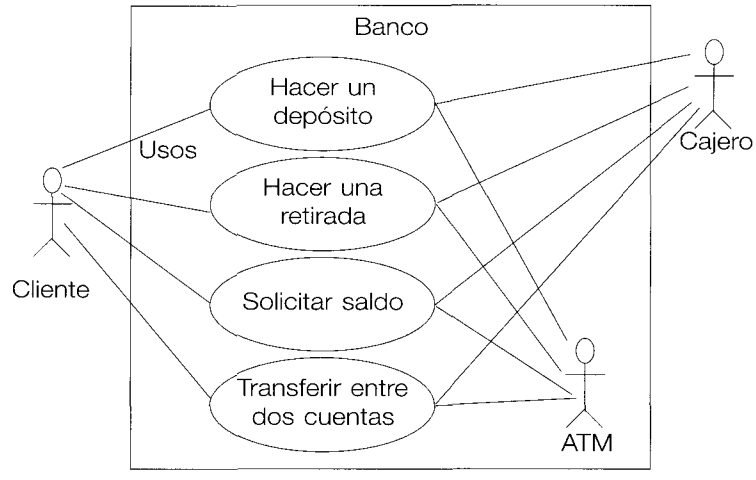
- "¿Quién usará el sistema?"
- "¿Qué pueden hacer esos actores con el sistema?"

[13] Agradecemos la ayuda de James H. Jarrett.

- ¿Cómo se las ingenia *cada actor* para *hacer* eso con este sistema?"
- "¿De qué otra forma podría funcionar esto si alguien más lo estuviera haciendo, o si el mismo actor tuviera un objetivo distinto?" (Para encontrar posibles variaciones.)
- "¿Qué problemas podrían surgir mientras se hace esto con el sistema?" (Para localizar posibles excepciones.)

Si se está diseñando, por ejemplo, un cajero automático, el caso de uso para un aspecto particular de la funcionalidad del sistema debe ser capaz de describir qué hace el cajero en cada situación posible. Cada una de estas "situaciones" se denomina un *escenario*, y un caso de uso puede considerarse como una colección de escenarios. Uno puede pensar que un escenario es como una pregunta que empieza por: "¿Qué hace el sistema si...?". Por ejemplo: "¿Qué hace el cajero si un cliente acaba de depositar durante las últimas 24 horas un cheque y no hay dinero suficiente en la cuenta, sin haber procesado el cheque, para proporcionarle la retirada el efectivo que ha solicitado?"

Deben utilizarse diagramas de caso de uso intencionadamente simples para evitar ahogarse prematuramente en detalles de implementación del sistema :

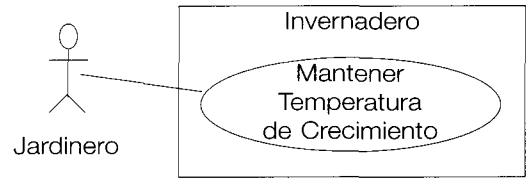


Cada uno de los monigotes representa a un "actor", que suele ser generalmente un humano o cualquier otro tipo de agente (por ejemplo, otro sistema de computación, como "ATM")[14]. La caja representa los límites de nuestro sistema. Las elipses representan los casos de uso, que son descripciones del trabajo útil que puede hacerse dentro del sistema. Las líneas entre los actores y los casos de uso representan las interacciones.

De hecho no importa cómo esté el sistema implementado, siempre y cuando tenga una apariencia como ésta para el usuario.

[14] ATM, siglas en inglés de cajero automático. (N. del T.)

Un caso de uso no tiene por qué ser terriblemente complejo, aunque el sistema subyacente sea complejo. Solamente se pretende que muestre el sistema tal y como éste se muestra al usuario. Por ejemplo:



Los casos de uso proporcionan las especificaciones de requisitos determinando todas las interacciones que el usuario podría tener con el sistema. Se trata de descubrir un conjunto completo de casos de uso para su sistema, y una vez hecho esto, se tiene el núcleo de lo que el sistema se supone que hará. Lo mejor de centrarse en los casos de uso es que siempre permiten volver a la esencia manteniéndose alejado de aspectos que no son críticos para lograr culminar el trabajo. Es decir, si se tiene un conjunto completo de casos de uso, es posible describir el sistema y pasar a la siguiente fase. Posiblemente no se podrá configurar todo a la primera, pero no pasa nada. Todo irá surgiendo a su tiempo, y si se demanda una especificación perfecta del sistema en este punto, uno se quedará parado.

Cuando uno se quede bloqueado, es posible comenzar esta fase utilizando una extensa herramienta de aproximación: describir el sistema en unos pocos párrafos y después localizar los sustantivos y los verbos. Los sustantivos pueden sugerir quiénes son los actores, el contexto del caso de uso (por ejemplo, "corredor"), o artefactos manipulados en el caso de uso. Los verbos pueden sugerir interacciones entre los actores y los casos de uso, y especificar los pasos dentro del caso de uso. También será posible descubrir que los sustantivos y los verbos producen objetos y mensajes durante la fase de diseño (y debe tenerse en cuenta que los casos de uso describen interacciones entre subsistemas, de forma que la técnica de "el sustantivo y el verbo" puede usarse sólo como una herramienta de tormenta de ideas, pues no genera casos de uso)[15].

La frontera entre un caso de uso y un actor puede señalar la existencia de una interfaz de usuario, pero no lo define. Para ver el proceso de cómo definir y crear interfaces de usuario, véase *Software for Use* de Larry Constantine y Lucy Lockwood, (Addison-Wesley Longman, 1999) o ir a *http://www.ForUse.com.*

Aunque parezca magia negra, en este punto es necesario algún tipo de planificación. Ahora se tiene una visión de lo que se está construyendo, por lo que probablemente se pueda tener una idea de cuánto tiempo le llevará. En este momento intervienen muchos factores. Si se estima una planificación larga, la compañía puede decidir no construirlo (y por consiguiente usar sus recursos en algo más razonable —esto es *bueno*). Pero un director podría tener decidido de antemano cuánto tiempo debería llevar el proyecto y podría tratar de influir en la estimación. Pero lo mejor es tener una estimación honesta desde el principio y tratar las decisiones duras al principio. Ha habido muchos in-

[15] Puede encontrarse más información sobre casos de uso en *Applying Use Cases*, de Schneider & Winters (Addison-Weley 1998) y *Use Case Driven Object modeling with UML* de Rosenberg (Addison-Welsey 1999).

tentos de desarrollar técnicas de planificación exactas (muy parecidas a las técnicas de predicción del mercado de valores), pero probablemente el mejor enfoque es confiar en la experiencia e intuición. Debería empezarse por una primera estimación del tiempo que llevaría, para posteriormente multiplicarla por dos y añadirle el 10 por ciento. La estimación inicial puede que sea correcta; a lo mejor *se puede* hacer que algo funcione en ese tiempo. Al "doblarlo" resultará que se consigue algo decente, y en el 10 por ciento añadido se puede acabar de pulir y tratar los detalles finales[16]. Sin embargo, es necesario explicarlo, y dejando de lado las manipulaciones y quejas que surgen al presentar una planificación de este tipo, normalmente funcionará.

Fase 2: ¿Cómo construirlo?

En esta fase debe lograrse un diseño que describe cómo son las clases y cómo interactuarán. Una técnica excelente para determinar las clases e interacciones es la tarjeta *Clase-Responsabilidad-Colaboración* (CRC)[17]. Parte del valor de esta herramienta se basa en que es de muy baja tecnología: se comienza con un conjunto de tarjetas de 3 x 5, y se escribe en ellas. Cada tarjeta representa una única clase, y en ella se escribe:

1. El nombre de la clase. Es importante que este nombre capture la esencia de lo que hace la clase, de manera que tenga sentido a primera vista.
2. Las "responsabilidades" de la clase: qué debería hacer. Esto puede resumirse típicamente escribiendo simplemente los nombres de las funciones miembros (dado que esas funciones deberían ser descriptivas en un buen diseño), pero no excluye otras anotaciones. Si se necesita ayuda, basta con mirar el problema desde el punto de vista de un programador holgazán: ¿qué objetos te gustaría que apareciesen por arte de magia para resolver el problema?
3. Las "colaboraciones" de la clase: ¿con qué otras clases interactúa? "Interactuar" es un término amplio intencionadamente; vendría a significar agregación, o simplemente que cualquier otro objeto existente ejecutara servicios para un objeto de la clase. Las colaboraciones deberían considerar también la audiencia de esa clase. Por ejemplo, si se crea una clase **Petardo**, ¿quién la va a observar, un **Químico** o un **Observador**? En el primer caso estamos hablando de punto de vista del químico que va a construirlo, mientras que en el segundo se hace referencia a los colores y las formas que libere al explotar.

Uno puede pensar que las tarjetas deberían ser más grandes para que cupiera en ellas toda la información que se deseara escribir, pero son pequeñas a propósito, no sólo para mantener pequeño el tamaño de las clases, sino también para evitar que se caiga en demasiado nivel de detalle muy pronto. Si uno no puede encajar todo lo que necesita saber de una clase en una pequeña tarjeta, la clase es demasiado compleja (o se está entrando en demasiado nivel de detalle, o se debería crear más de una clase). La clase ideal debería ser comprensible a primera vista. La idea de las tarjetas CRC es ayudar a obtener un primer diseño de manera que se tenga un dibujo a grandes rasgos que pueda ser después refinado.

[16] Mi opinión en este sentido ha cambiado últimamente. Al doblar y añadir el 10 por ciento se obtiene una estimación razonablemente exacta (asumiendo que no hay demasiados factores al azar) pero todavía hay que trabajar con bastante diligencia para finalizar en ese tiempo. Si se desea tener tiempo suficiente para lograr un producto verdaderamente elegante y disfrutar durante el proceso, el multiplicador correcto, en mi opinión, puede ser por tres o por cuatro.

[17] En inglés, *Class-Responsibility-Collaboration*. (N. del R.T.)

Una de las mayores ventajas de las tarjetas CRC se logra en la comunicación. Cuando mejor se hace es en tiempo real, en grupo y sin computadores. Cada persona se considera responsable de varias clases (que al principio no tienen ni nombres ni otras informaciones). Se ejecuta una simulación en directo resolviendo cada vez un escenario, decidiendo qué mensajes se mandan a los distintos objetos para satisfacer cada escenario. A medida que se averiguan las responsabilidades y colaboraciones de cada una, se van rellenando las tarjetas correspondientes. Cuando se han recorrido todos los casos de uso, se debería tener un diseño bastante completo.

Antes de empezar a usar tarjetas CRC, tuve una experiencia de consultoría de gran éxito, que me permitió presentar un diseño inicial a todo el equipo, que jamás había participado en un proyecto de POO, y que consistió en ir dibujando objetos en una pizarra, después de hablar sobre cómo se deberían comunicar los objetos entre sí, y tras borrar algunos y reemplazar otros. Efectivamente, estaban haciendo uso de "tarjetas CRC" en la propia pizarra. El equipo (que sabía que el proyecto se iba a hacer) creó, de hecho, el diseño; ellos eran los "propietarios" del diseño, más que recibirlo hecho directamente. Todo lo que yo hacía era guiar el proceso haciendo en cada momento las preguntas adecuadas, poniendo a prueba los distintos supuestos, y tomando la realimentación del equipo para ir modificando los supuestos. La verdadera belleza del proyecto es que el equipo aprendió cómo hacer diseño orientado a objetos no repasando ejemplos o resúmenes de ejemplos, sino trabajando en el diseño que les pareció más interesante en ese momento: el de ellos mismos.

Una vez que se tiene un conjunto de tarjetas CRC se desea crear una descripción más formal del diseño haciendo uso de UML[18]. No es necesario utilizar UML, pero puede ser de gran ayuda, especialmente si se desea poner un diagrama en la pared para que todo el mundo pueda ponderarlo, lo cual es una gran idea. Una alternativa a UML es una descripción textual de los objetos y sus interfaces, o, dependiendo del lenguaje de programación, el propio código[19].

UML también proporciona una notación para diagramas que permiten describir el modelo dinámico del sistema. Esto es útil en situaciones en las que las transiciones de estado de un sistema o subsistema son lo suficientemente dominantes como para necesitar sus propios diagramas (como ocurre en un sistema de control). También puede ser necesario describir las estructuras de datos, en sistemas o subsistemas en los que los datos sean un factor dominante (como una base de datos).

Sabemos que la Fase 2 ha acabado cuando se han descrito los objetos y sus interfaces. Bueno, la mayoría —hay generalmente unos pocos que quedan ocultos y que no se dan a conocer hasta la Fase 3. Pero esto es correcto. En lo que a uno respecta, esto es todo lo que se ha podido descubrir de los objetos a manipular. Es bonito descubrirlos en las primeras etapas del proceso pero la POO proporciona una estructura tal, que no presenta problema si se descubren más tarde. De hecho, el diseño de un objeto tiende a darse en cinco etapas, a través del proceso completo de desarrollo de un programa.

---

[18] Para los principiantes, recomiendo *UML Distilled*, 2.ª edición.

[19] Python *(http://www.Python.org)* suele utilizarse como "pseudocódigo ejecutable".

Las cinco etapas del diseño de un objeto

La duración del diseño de un objeto no se limita al tiempo empleado en la escritura del programa, sino que el diseño de un objeto conlleva una serie de etapas. Es útil tener esta perspectiva porque se deja de esperar la perfección; por el contrario, uno comprende lo que hace un objeto y el nombre que debería tener surge con el tiempo. Esta visión también se aplica al diseño de varios tipos de programas; el patrón para un tipo de programa particular emerge al enfrentarse una y otra vez con el problema (esto se encuentra descrito en el libro *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*). Los objetos, también tienen su patrón, que emerge a través de su entendimiento, uso y reutilización.

1. **Descubrimiento de los objetos**. Esta etapa ocurre durante el análisis inicial del programa. Se descubren los objetos al buscar factores externos y limitaciones, elementos duplicados en el sistema, y las unidades conceptuales más pequeñas. Algunos objetos son obvios si ya se tiene un conjunto de bibliotecas de clases. La comunidad entre clases que sugieren clases bases y herencia, puede aparecer también en este momento, o más tarde dentro del proceso de diseño.

2. **Ensamblaje de objetos**. Al construir un objeto se descubre la necesidad de nuevos miembros que no aparecieron durante el descubrimiento. Las necesidades internas del objeto pueden requerir de otras clases que lo soporten.

3. **Construcción del sistema**. De nuevo, pueden aparecer en esta etapa más tardía nuevos requisitos para el objeto. Así se aprende que los objetos van evolucionando. La necesidad de un objeto de comunicarse e interconectarse con otros del sistema puede hacer que las necesidades de las clases existentes cambien, e incluso hacer necesarias nuevas clases. Por ejemplo, se puede descubrir la necesidad de clases que faciliten o ayuden, como una lista enlazada, que contiene poca o ninguna información de estado y simplemente ayuda a la función de otras clases.

4. **Aplicación del sistema**. A medida que se añaden nuevos aspectos al sistema, puede que se descubra que el diseño previo no soporta una ampliación sencilla del sistema. Con esta nueva información, puede ser necesario reestructurar partes del sistema, generalmente añadiendo nuevas clases o nuevas jerarquías de clases.

5. **Reutilización de objetos**. Ésta es la verdadera prueba de diseño para una clase. Si alguien trata de reutilizarla en una situación completamente nueva, puede que descubra pequeños inconvenientes. Al cambiar una clase para adaptarla a más programas nuevos, los principios generales de la clase se mostrarán más claros, hasta tener un tipo verdaderamente reutilizable. Sin embargo, no debe esperarse que la mayoría de objetos en un sistema se diseñen para ser reutilizados —es perfectamente aceptable que un porcentaje alto de los objetos sean específicos del sistema para el que fueron diseñados. Los tipos reutilizables tienden a ser menos comunes, y deben resolver problemas más generales para ser reutilizables.

Guías para el desarrollo de objetos

Estas etapas sugieren algunas indicaciones que ayudarán a la hora de pensar en el desarrollo de clases:

1. Debe permitirse que un problema específico genere una clase, y después dejar que la clase crezca y madure durante la solución de otros problemas.

2. Debe recordarse que descubrir las clases (y sus interfaces) que uno necesita es la tarea principal del diseño del sistema. Si ya se disponía de esas clases, el proyecto será fácil.

3. No hay que forzar a nadie a saber todo desde el principio; se aprende sobre la marcha.Y esto ocurrirá poco a poco.

4. Hay que empezar programando; es bueno lograr algo que funcione de manera que se pueda probar la validez o no de un diseño. No hay que tener miedo a acabar con un código de estilo procedimental malo —las clases segmentan el problema y ayudan a controlar la anarquía y la entropía. Las clases malas no estropean las clases buenas.

5. Hay que mantener todo lo más simple posible. Los objetos pequeños y limpios con utilidad obvia son mucho mejores que interfaces grandes y complicadas. Cuando aparecen puntos de diseño puede seguirse el enfoque de una afeitadora de Occam: se consideran las alternativas y se selecciona la más simple, porque las clases simples casi siempre resultan mejor. Hay que empezar con algo pequeño y sencillo, siendo posible ampliar la interfaz de la clase al entenderla mejor. A medida que avance el tiempo será difícil eliminar elementos de una clase.

Fase 3: Construir el núcleo

Ésta es la conversión inicial de diseño pobre en un código compilable y ejecutable que pueda ser probado, y especialmente, que pueda probar la validez o no de la arquitectura diseñada. Este proceso no se puede hacer de una pasada, sino que consistirá más bien en una serie de pasos que permitirán construir el sistema de manera iterativa, como se verá en la Fase 4.

Su objetivo es encontrar el núcleo de la arquitectura del sistema que necesita implementar para generar un sistema ejecutable, sin que importe lo incompleto que pueda estar este sistema en esta fase inicial. Está creando un armazón sobre el que construir en posteriores iteraciones. También se está llevando a cabo la primera de las muchas integraciones y pruebas del sistema, a la vez que proporcionando a los usuarios una realimentación sobre la apariencia que tendrá su sistema, y cómo va progresando. Idealmente, se están además asumiendo algunos riesgos críticos. De hecho, se descubrirán posibles cambios y mejoras que se pueden hacer sobre el diseño original —cosas que no se hubieran descubierto de no haber implementado el sistema.

Una parte de la construcción del sistema es comprobar que realmente se cumple el análisis de requisitos y la especificación del sistema que realmente cumple el analisis de requisitos y la especificación del sistema (independientemente de la forma en que estén planteados). Debe asegurarse que las pruebas verifican los requerimientos y los casos de uso. Cuando el corazón del sistema sea estable, será posible pasar a la siguiente fase y añadir nuevas funcionalidades.

Fase 4: Iterar los casos de uso

Una vez que el núcleo del sistema está en ejecución, cada característica que se añada es en sí misma un pequeño proyecto. Durante cada *iteración*, entendida como un periodo de desarrollo razonablemente pequeño, se añade un conjunto de características.

¿Cúal debe ser la duración de una iteración? Idealmente, cada iteración dura de una a tres semanas (la duración puede variar en función del lenguaje de implementación). Al final de ese periodo, se tiene un sistema integrado y probado con una funcionalidad mayor a la que tenía previamente. Pero lo particularmente interesante es la base de la iteración: un único caso de uso. Cada caso de uso es un paquete de funcionalidad relacionada que se construye en el sistema de un golpe, durante una iteración. Esto no sólo proporciona una mejor idea de lo que debería ser el ámbito de un caso de uso, sino que además proporciona una validación mayor de la idea del mismo, dado que el concepto no queda descartado hasta después del análisis y del diseño, pues éste es una unidad de desarrollo fundamental a lo largo de todo el proceso de construcción de software.

Se deja de iterar al lograr la funcionalidad objetivo, o si llega un plazo y el cliente se encuentra satisfecho con la versión actual (debe recordarse que el software es un negocio de suscripción). Dado que el proceso es iterativo, uno puede tener muchas oportunidades de lanzar un producto, más que tener un único punto final; los proyectos abiertos trabajan exclusivamente en entornos iterativos de gran nivel de realimentación, que es precisamente lo que les permite acabar con éxito.

Un proceso de desarrollo iterativo tiene gran valor por muchas razones. Si uno puede averiguar y resolver pronto los riesgos críticos, los clientes pueden tener muchas oportunidades de cambiar de idea, la satisfacción del programador es mayor, y el proyecto puede guiarse con mayor precisión. Pero otro beneficio adicional importante es la realimentación a los usuarios, que pueden ver a través del estado actual del producto cómo va todo. Así es posible reducir o eliminar la necesidad de reuniones de estado "entumece-mentes" e incrementar la confianza y el soporte de los usuarios.

Fase 5: Evolución

Éste es el punto del ciclo de desarrollo que se ha denominado tradicionalmente "mantenimiento", un término global que quiere decir cualquier cosa, desde "hacer que funcione de la manera que se suponía que lo haría en primer lugar", hasta "añadir aspectos varios que el cliente olvidó mencionar", pasando por el tradicional "arreglar los errores que puedan aparecer" o "la adición de nuevas características a medida que aparecen nuevas necesidades". Por ello, al término "mantenimiento" se le han aplicado numerosos conceptos erróneos, lo que ha ocasionado un descenso progresivo de su calidad, en parte porque sugiere que se construyó una primera versión del programa en la cual hay que ir cambiando partes, además de engrasarlo para evitar que se oxide. Quizás haya un término mejor para describir lo que está pasando.

Prefiero el término *evolución*[20]. De esta forma, "uno no acierta a la primera, por lo que debe concederse la libertad de aprender y volver a hacer nuevos cambios". Podríamos necesitar muchos cambios a medida que vamos aprendiendo y comprendiendo con más detenimiento el problema. A corto y largo plazo, será el propio programa el que se verá beneficiado de este proceso continuo de evolución. De hecho, ésta permitirá que el programa pase de bueno a genial, haciendo que se aclaren aquellos aspectos que no fueron verdaderamente entendidos en la primera pasada. También es

[20] El libro de Martin Fowler *Refactoring: improving the design of existing code* (Addison-Wesley, 1999) cubre al menos un aspecto de la evolución, utilizando exclusivamente ejemplos en Java.

en este proceso en el que las clases se convierten en recursos reutilizables, en vez de clases diseñadas para su uso en un solo proyecto.

"Hacer el proyecto bien" no sólo implica que el programa funcione de acuerdo con los requisitos y casos de uso. También quiere decir que la estructura interna del código tenga sentido, y que parezca que encaja bien, sin aparentar tener una sintaxis extraña, objetos de tamaño excesivo o con fragmentos inútiles de código. Además, uno debe tener la sensación de que la estructura del programa sobrevivirá a los cambios que inevitablemente irá sufriendo a lo largo de su vida, y de que esos cambios se podrán hacer de forma sencilla y limpia. Esto no es trivial. Uno no sólo debe entender qué es lo que está construyendo, sino también cómo evolucionará el programa (lo que yo denomino el *vector del cambio*). Afortunadamente, los lenguajes de programación orientada a objetos son especialmente propicios para soportar este tipo de modificación continua —los límites creados por los objetos son los que tienden a lograr una estructura sólida. También permiten hacer cambios —que en un programa procedural parecerían drásticos— sin causar terremotos a lo largo del código. De hecho, el soporte a la evolución podría ser el beneficio más importante de la POO.

Con la evolución, se crea algo que al menos se aproxima a lo que se piensa que se está construyendo, se compara con los requisitos, y se ve dónde se ha quedado corto. Después, se puede volver y ajustarlo diseñando y volviendo a implementar las porciones del programa que no funcionaron correctamente[21]. De hecho, es posible que se necesite resolver un problema, o determinado aspecto de un problema, varias veces antes de dar con la solución correcta (suele ser bastante útil estudiar en este momento *el Diseño de Patrones)*. También es posible encontrar información en *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*).

La evolución también se da al construir un sistema, ver que éste se corresponda con los requisitos, y descubrir después que no era, de hecho, lo que se pretendía. Al ver un sistema en funcionamiento, se puede descubrir que verdaderamente se pretendía que solucionase otro problema. Si uno espera que se dé este tipo de evolución, entonces se debe construir la primera versión lo más rápidamente posible con el propósito de averiguar sin lugar a dudas qué es exactamente lo que se desea.

Quizás lo más importante que se ha de recordar es que por defecto, si se modifica una clase, sus súper y subclases seguirán funcionando. Uno no debe tener miedo a la modificación (especialmente si se dispone de un conjunto de pruebas, o alguna prueba individual que permita verificar la corrección de las modificaciones). Los cambios no tienen por qué estropear el programa, sino que cualquiera de las consecuencias de un cambio se limitarán a las subclases y/o colaboradores específicos de la clase que se modifica.

Los planes merecen la pena

Por supuesto, uno jamás construiría una casa sin unos planos cuidadosamente elaborados. Si construyéramos un hangar o la casa de un perro, los planes no tendrían tanto nivel de detalle, pero pro-

[21] Esto es semejante a la elaboración de "prototipos rápidos", donde se supone que uno construye una versión "rápida y sucia" que permite comprender mejor el sistema, pero que es después desechada para construirlo correctamente. El problema con el prototipado rápido es que los equipos de desarrollo no suelen desechar completamente el prototipo, sino que lo utilizan como base sobre la que construir. Si se combina, en la programación procedural, con la falta de estructura, se generan sistemas totalmente complicados, y difíciles de mantener.