**Clases**

**Organización De Clases**Una clase en Java debe comenzar con una lista de variables. Primero se declaran las constantes estáticas públicas (si las hay), seguidas por las variables estáticas privadas y, finalmente, las variables de instancia privadas. Las variables públicas generalmente no son necesarias.

Después de la lista de variables, se deben definir las funciones públicas. Las funciones auxiliares públicas, que son utilizadas por estas funciones públicas se colocan a continuación. Esta estructura sigue una regla de presentación descendente, facilitando la lectura del código.

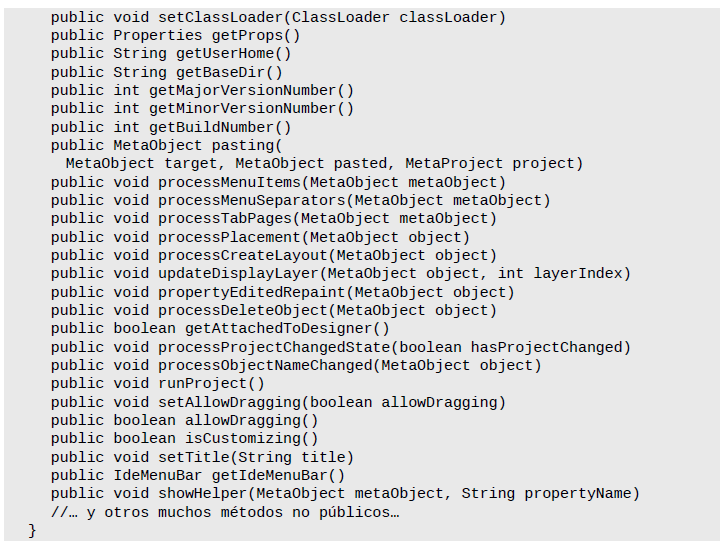
**Encapsulación:**

Idealmente, las variables y funciones deberían ser privadas. Sin embargo, en ocasiones se pueden hacer accesibles para pruebas o en situaciones específicas. Si una función o variable necesita ser accesible desde el mismo paquete, se debe usar el modificador protected o el ámbito de paquete. La relajación de la encapsulación debe ser una medida excepcional.

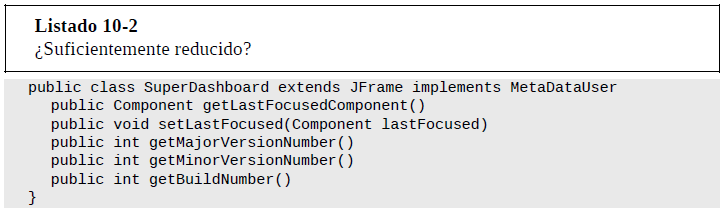
Las clases deben ser de tamaño reducido  
La primera regla de las clases es que deben ser de tamaño reducido. La segunda regla es que deben ser todavía más reducidas.

El Listado 10-1 muestra una clase, SuperDashboard, que muestra 70 métodos públicos. Muchos programadores estarán de acuerdo en que es un tamaño excesivo. Algunos denominarían a SuperDashboard una clase Dios.





¿Y si SuperDashboard sólo incluyera los métodos mostrados en el Listado 10-2?

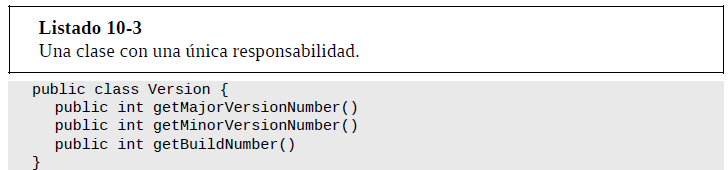


En este caso fueron necesarios los 5 métodos ya que, a pesar del reducido número de métodos, SuperDashboard cuenta con demasiadas responsabilidades.

El nombre de una clase debe reflejar claramente sus responsabilidades. Un nombre claro y conciso puede ayudar a determinar si la clase tiene un tamaño adecuado. Si es difícil encontrar un nombre específico para una clase, es probable que esté manejando demasiadas responsabilidades. Nombres como Processor, Manager o Super suelen indicar que la clase tiene demasiadas funciones o responsabilidades.

Principio de Responsabilidad Única (SRP):

El Principio de Responsabilidad Única establece que una clase o módulo debe tener solo una razón para cambiar, es decir, debe tener una única responsabilidad. Esto ayuda a definir claramente el propósito de una clase y a determinar su tamaño adecuado. Por ejemplo, la clase SuperDashboard en el Listado 10-2 tiene dos responsabilidades: gestionar la información de versión y manejar componentes de Java Swing. Estos dos motivos de cambio son independientes, lo que sugiere que la clase está sobrecargada. Para mejorar el diseño, se puede extraer la gestión de versiones a una clase separada, como Version, permitiendo así reutilizar esta clase en otras aplicaciones y clarificar las responsabilidades.



Importancia y Aplicación del Principio de Responsabilidad Única (SRP):

El SRP es fundamental en el diseño orientado a objetos y, aunque es fácil de entender, a menudo se aplica incorrectamente. Muchas veces, las clases terminan haciendo demasiadas cosas porque el enfoque inicial suele estar en que el software funcione, no en su organización y limpieza. A menudo, los programadores solo se concentran en hacer que el código funcione y pasan por alto la necesidad de refactorizar y dividir las clases en unidades más manejables con una sola responsabilidad.

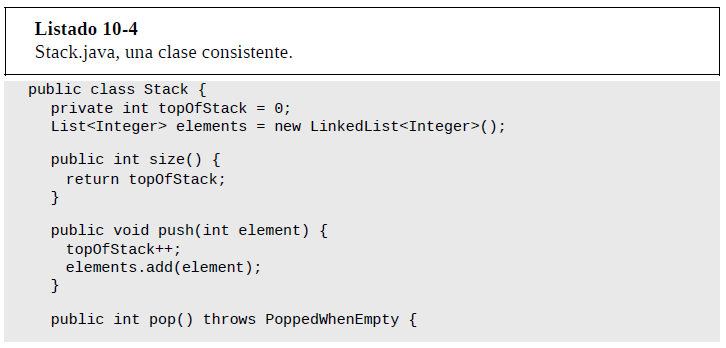
Además, algunos temen que dividir el código en muchas clases pequeñas pueda hacer que el sistema sea más difícil de entender, ya que tendrían que navegar entre varias clases para comprender cómo funciona un aspecto específico del sistema. Sin embargo, un sistema con muchas clases pequeñas no es más complicado que uno con pocas clases grandes; la clave es la organización. Tener muchas clases pequeñas y bien definidas permite una mejor gestión de la complejidad, facilitando a los programadores encontrar y entender lo que necesitan sin tener que lidiar con una clase monolítica que mezcla múltiples responsabilidades.

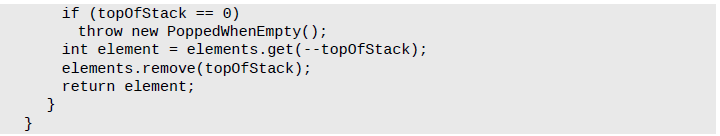
Es mejor diseñar sistemas con muchas clases pequeñas, cada una con una única responsabilidad, en lugar de pocas clases grandes que intentan hacer demasiado. Esto ayuda a organizar la complejidad y facilita la comprensión del sistema.

Cohesión

Las clases deben tener un número limitado de variables de instancia, y los métodos de la clase deben operar sobre estas variables. Cuantas más variables maneje un método, mayor será la cohesión de la clase. La cohesión es máxima cuando cada método utiliza todas las variables de la clase.

Aunque no siempre es posible o recomendable alcanzar una cohesión perfecta, se debe buscar una alta cohesión en las clases. Esto significa que los métodos y variables de la clase están estrechamente relacionados y trabajan juntos de manera coherente como un todo lógico.





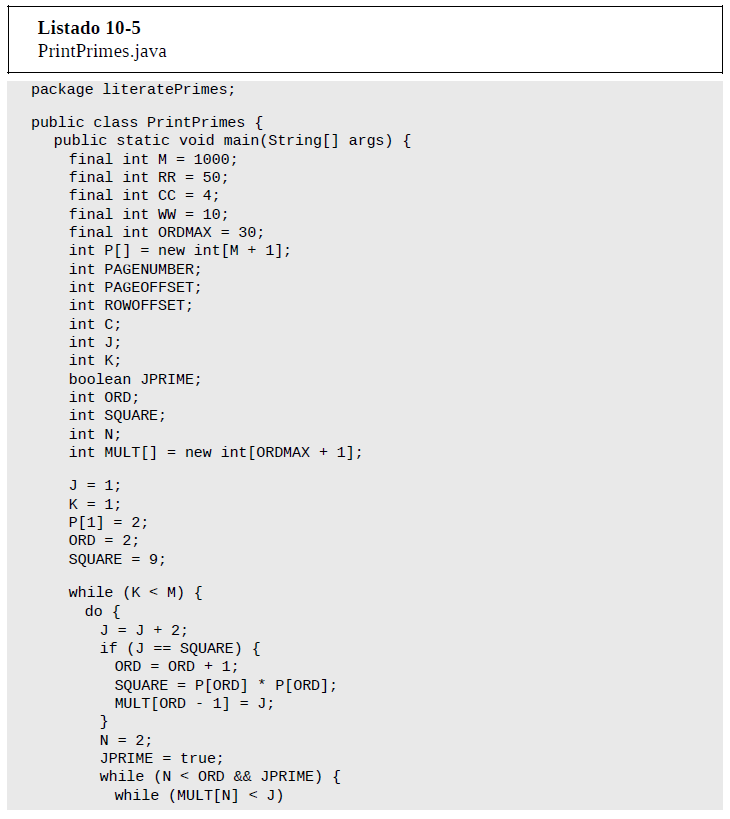
Reducir el tamaño de las funciones y el número de parámetros puede llevar a que algunas variables de instancia sean usadas solo por un subconjunto de métodos. Este problema indica que la clase original puede estar demasiado cargada. Para mejorar la organización, se debe considerar dividir la clase en dos o más clases más pequeñas, cada una con una mayor coherencia interna. Esta separación ayuda a que cada clase sea más consistente y manejable.

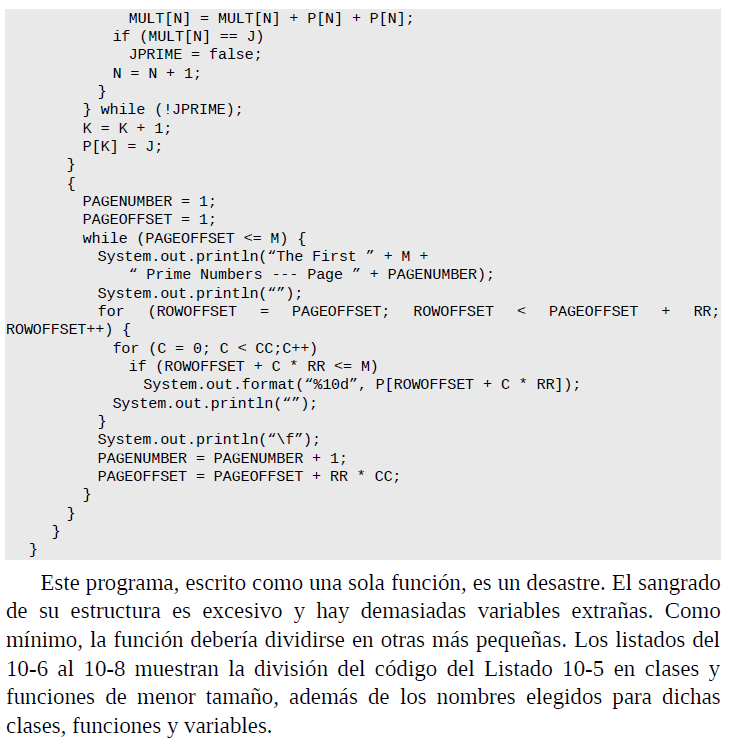
Dividir grandes funciones en funciones más pequeñas puede llevar a un aumento en el número de clases si se usan muchas variables. En lugar de pasar numerosas variables como argumentos a una nueva función, es más eficiente convertirlas en variables de instancia de la clase. Esto facilita la extracción del código sin necesidad de pasar todas las variables.

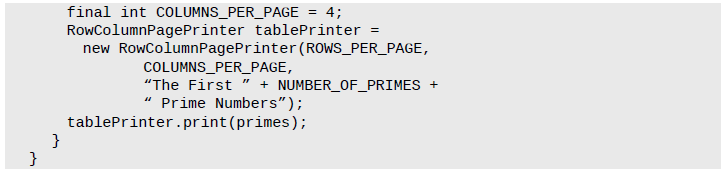
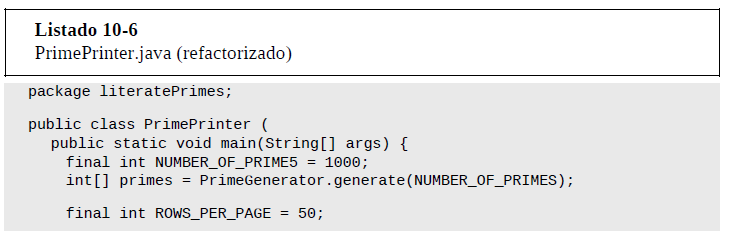
Sin embargo, esto puede reducir la cohesión de las clases, ya que acumularán más variables de instancia solo para ser compartidas entre funciones. Si una clase termina teniendo muchas variables compartidas sin una clara cohesión, podría ser una señal de que debería ser dividida en clases más pequeñas.

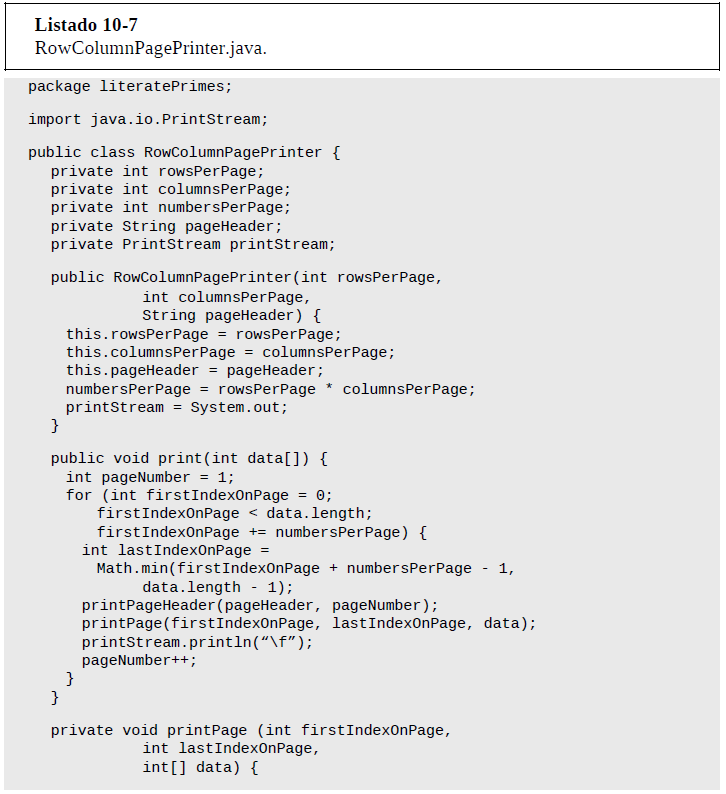
Al dividir funciones grandes en fragmentos más pequeños, también se puede llevar a dividir clases grandes en clases más reducidas. Esto mejora la organización y hace que la estructura del programa sea más clara.

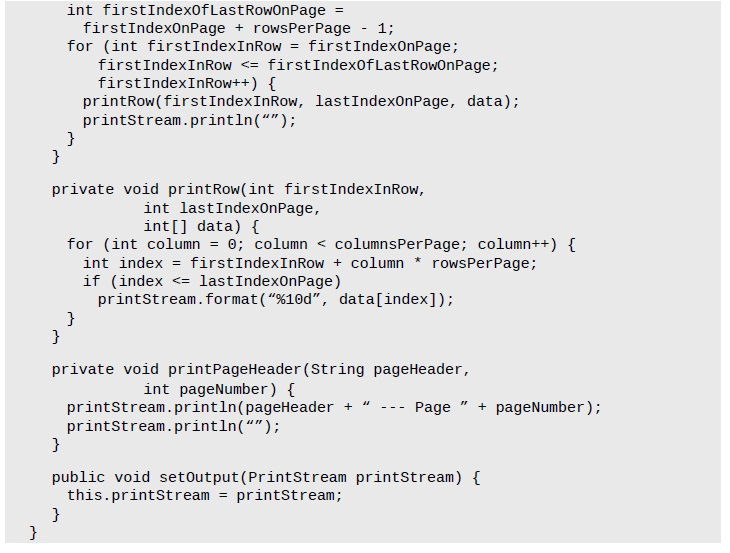
Ejemplo obtenido del libro de Knuth LiterateProgramming

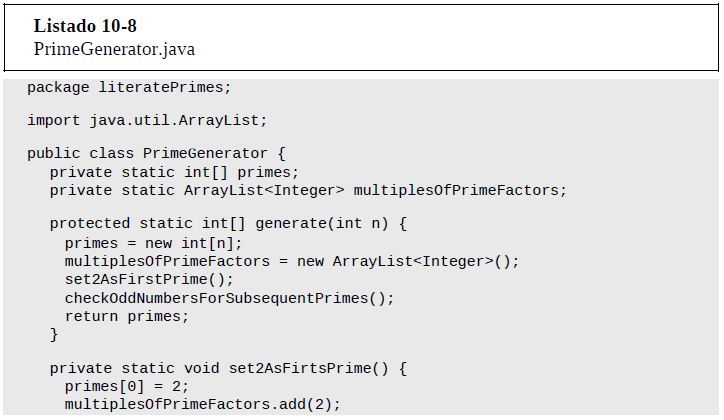








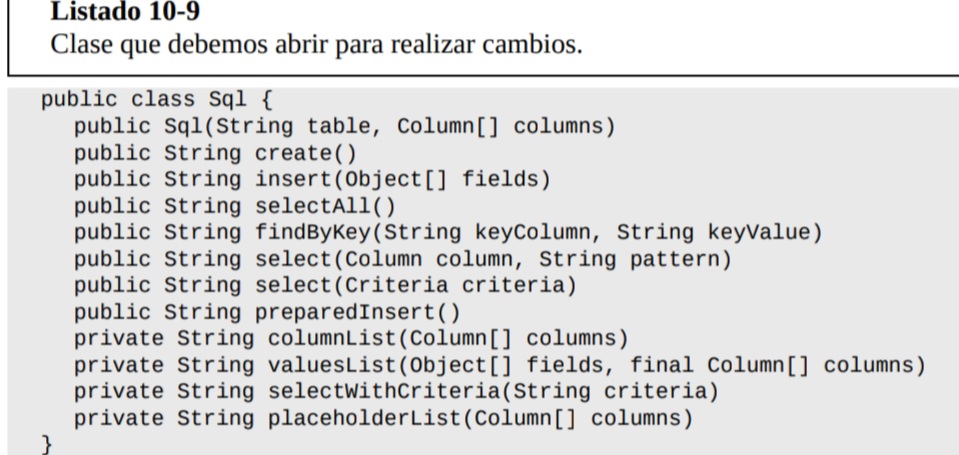






**Organizar los cambios**

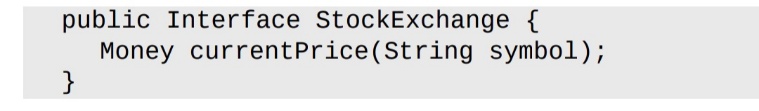
En sistemas que experimentan cambios continuos, cada modificación puede afectar el funcionamiento general del sistema. Para minimizar estos riesgos, es crucial organizar las clases de manera adecuada. Por ejemplo, la clase Sql se encarga de generar cadenas SQL correctamente usando los metadatos correspondientes. No está diseñada para manejar funciones SQL como las instrucciones `update`.



Si en el futuro se necesita que la clase Sql soporte estas instrucciones, se deberán hacer modificaciones en la clase, lo que conlleva riesgos. Estas modificaciones deben probarse exhaustivamente para evitar afectar otras partes del código.

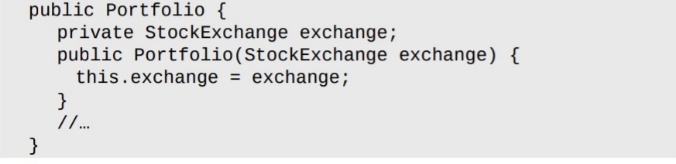
El código de las clases se simplifica, reduciendo el tiempo de comprensión y minimizando el riesgo de que una función afecte a otra. Las pruebas son más fáciles porque las clases están aisladas. Al agregar nuevas instrucciones update, no se modifica el código existente; en su lugar, se crea una subclase llamada `UpdateSql`, manteniendo el resto del sistema inalterado. Esta reestructuración cumple con el Principio de Responsabilidad Única (SRP) y el Principio Abierto/Cerrado, permitiendo que el sistema se amplíe sin modificar el código existente. Así, se minimiza la “suciedad” al agregar nuevas funcionalidades.

**Aislarnos de los cambios**

Las necesidades y el código cambiarán con Yel tiempo. En la programación orientada a objetos, existen clases concretas que contienen la implementación específica y clases abstractas que representan conceptos generales. Si una clase cliente depende de detalles concretos, puede verse afectada por cambios en estos detalles. Para minimizar este impacto, se utilizan interfaces y clases abstractas. Por ejemplo, en lugar de hacer que la clase Portfolio dependa directamente de una API externa como TokyoStockExchange, se crea una interfaz StockExchange.

Que define un método común. Esto facilita las pruebas y la adaptación a cambios en la API.

Creamos Tokyo Stock Exchange para poner en práctica esta interfaz. Además, garantizamos que el constructor de Portfolio acepte una referencia a StockExchange como parámetro.



Descomponer un sistema para pruebas mejora su flexibilidad y reutilización al reducir las conexiones entre sus componentes y con otros cambios, lo que facilita su comprensión. Este enfoque sigue el Principio de Inversión de Dependencias (DIP), que indica que las clases deben depender de abstracciones y no de detalles concretos. En lugar de que la clase Portfolio dependa directamente de la implementación de TokyoStockExchange, se basa en la interfaz StockExchange. Esta interfaz representa el concepto abstracto de obtener el precio de una acción, aislando así la clase Portfolio de los detalles específicos de cómo se obtiene esa información.