Pincipios de programación orientada a objetos

Introducción

La programación orientada a objetos entiende los programas como un conjunto organizado de objetos cooperativos. Cada objeto representa una instancia de una clase. Muchas de estas clases se relacionan mediante herencia o composición. Podemos decir que se focaliza la programación en el tratamiento de la información más que el la algorítmica y la información es parte privada de cada objeto. Solo un objeto sabe como operar con su propia información, lo cual protege la información de cambios indeseados.

Los principios que vamos a tratar son:

- Abstraction
- Encapsulation
- Inheritance
- Composition
- Modularity
- Polymophism

Abstraction

La abstracción denota las características esenciales de un objeto, que lo distinguen de otros objetos. Se centra en las características y operaciones esenciales, haciendo una interpretación codificada del problema que queremos resolver.

Por ejemplo, un coche puede ser visto como un conjunto que funciona. Su abstracción nos llevara a sus características (niveles de gasolina, neumáticos, velocidad, frenos) y a un conjunto de operaciones (acelerar, frenar, repostar). Las operaciones pueden afectar a características (acelerar

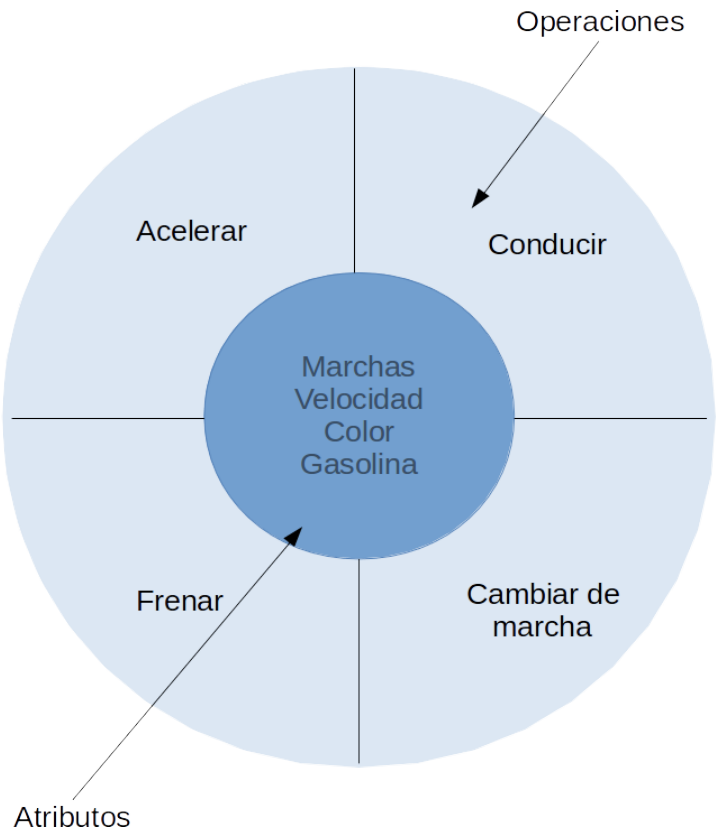
afectara al nivel de gasolina, a los neumáticos, la velocidad…).

Encapsulation

Consiste en separar la parte contractual de una abstracción y su implementación. Es decir, el proceso de compartimentar los elementos de una abstracción que constituye su estructura y comportamiento.

En la encapsulación, los elementos extraídos de la abstracción son compartimentados de tal manera que no todos serán accesibles des de fuera. Estos elementos pueden ser operaciones o atributos y estos son enlazados de tal manera que ciertos elementos no son accesibles des de fuera (normalmente atributos). De esta manera en vez de dar acceso a atributos directamente se hace des de operaciones donde se controla el acceso.

Cada objeto tiene su propio funcionamiento. Pero este no es visible a los usuarios. Tampoco es necesario y así evitamos problemas (posibles accesos a información sensible). Esto produce que los usuarios no entren en contacto con la implementación, esto permite modificarla sin ningún problema siempre y cuando se mantengan las operaciones intactas.

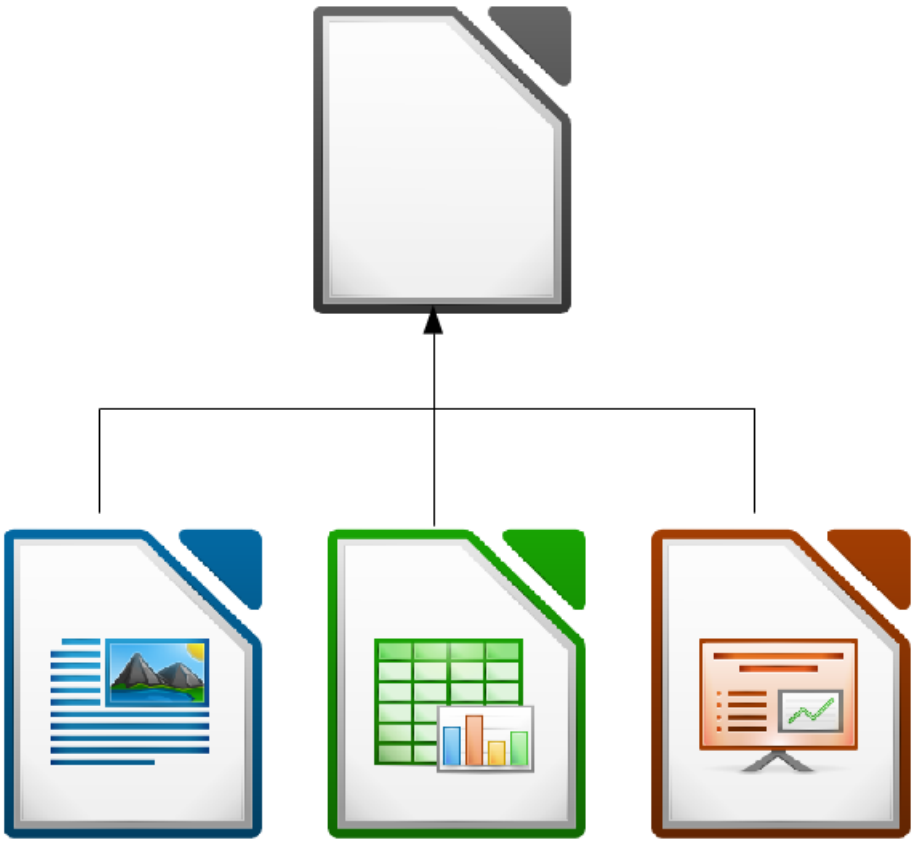


Inheritance

La herencia es la propiedad a través de la cual un objeto «hereda» las funcionalidades y estructura de otro.

Ciertos objetos tiene una misma estructura. En estos casos en vez de implementar esta misma estructura repetidamente, se puede representar en un solo objetos y los demás objetos puede «heredarlo» . Esto es posible cuando se ha creado una relación entre clases. Cuando se crean objetos de la clase el objeto hijo hereda las característica y funcionalidades del objeto padre.

En el siguiente esquema tenemos una clase documento. La clase documento contendrá la cabecera de los documentos y diferentes propiedades comunes de texto. Los documentos de texto, presentación y de hoja de calculo heredarán estas propiedades. De esta manera evitaremos repetir código inútilmente y lo reutilizaremos. Este tipo de relación de herencia se conoce como «is a».



Composition

La composición es una propiedad que permite a un objeto ser compuesto dentro de otro.

La composición facilita la reutilización de código. Se logra mediante la composición de un objeto dentro de otro. Por ejemplo, un objeto que representa un coche va a se compuesto por un objeto motor. Esta relación da lugar a objetos complejos y es vital para aplicaciones de gran tamaño.

Este tipo de relación se conoce como «has a» o «part of». Y existen dos tipos:

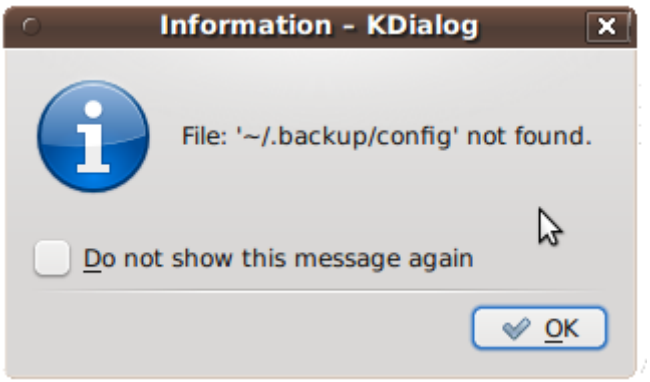
1. Aggregation

Se refiere a una relación débil entre el objeto exterior y el objeto interior. Donde este no depende de la vida útil del objeto exterior. En cuanto a código se refiere esto implica que el objeto exterior debe tener una referencia o puntero al

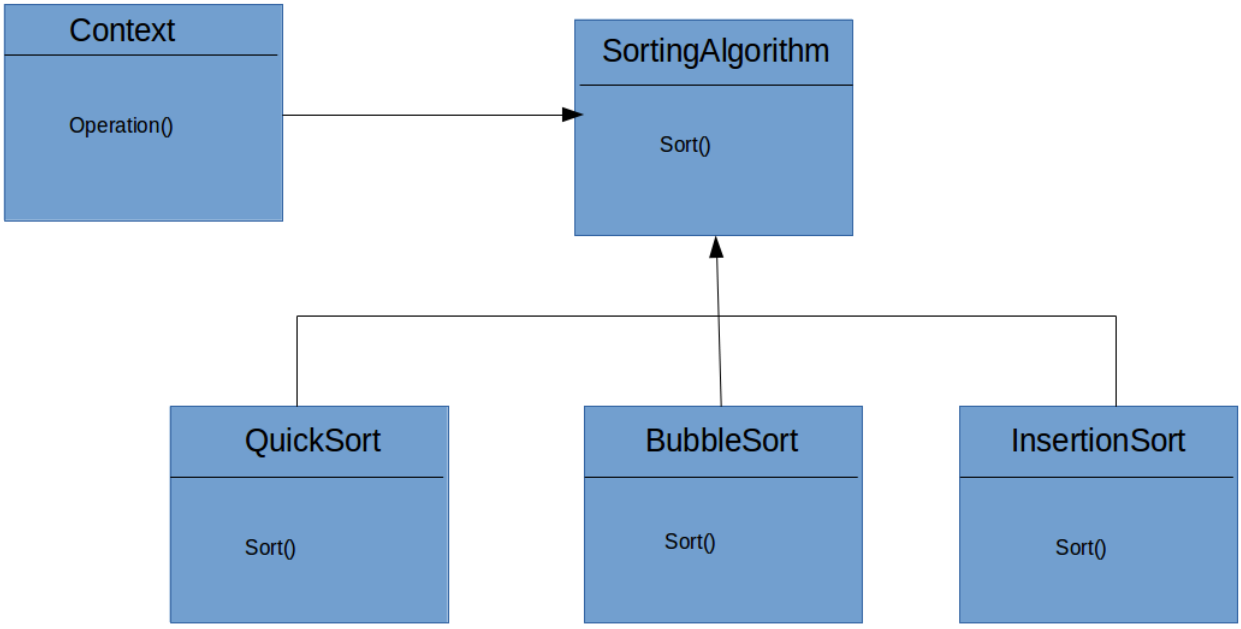
objeto interior. Por ejemplo, una ventana puede utilizar el cursor, pero cuando la ventana se destruye el cursor no.

1. Containment

Se refiere a una relación fuerte entre el objeto exterior y el objeto interior. En este tipo de relación (por ejemplo, en C++ una clase contiene el objeto de otra), la destrucción del objeto exterior implica la destrucción del objeto interior. Por ejemplo, cuando se destruye la ventana todos los controles de que dispone son destruidos (botones, label, edit boxes…)



A diferencia de la herencia, la composición es una relación dinámica porque se crea en tiempo de ejecución. Esto se consigue mediante punteros y referencias. En la figura siguiente, es posible para la clase *Context* utilizar cualquier tipo de algoritmo de ordenación en tiempo de ejecución.



Dada la naturaleza dinámica de la composición es mas flexible que la herencia. Las relaciones entre objetos se pueden crear o romper en tiempo de ejecución.

Modularity

Normalmente, en especial en grandes aplicaciones, tenemos que trabajar con un gran nombre de clases y otros útiles como

parte del código fuente. Desarrollar el código fuente como una única unidad es una tarea complicada i difícil.

por lo tanto, para reducir la complejidad, el código fuente es dividido en módulos pequeños e independientes. Normalmente estos son agrupados de forma lógica (conjuntos de procedimiento o bien la información con la que trabajan), así mantenemos grupos de código ordenados y estructurados.

El objetivo de la modularidad es descomponer el código en pequeñas unidades que faciliten el diseño, desarrollo y test de módulos individuales. Así podemos modificar un módulo sin tener conocimiento de los demás o bien que estos se vean afectados.

En cuanto a mantenimiento se refiere la modularidad es una gran herramienta. Sabes donde buscar exactamente cuando surge un error puesto que todo esta debidamente ordenado y compartimentado. Y una vez solucionado el problema no es necesario realizar tests de todo el código, solo de la parte afectada.

Polymophism

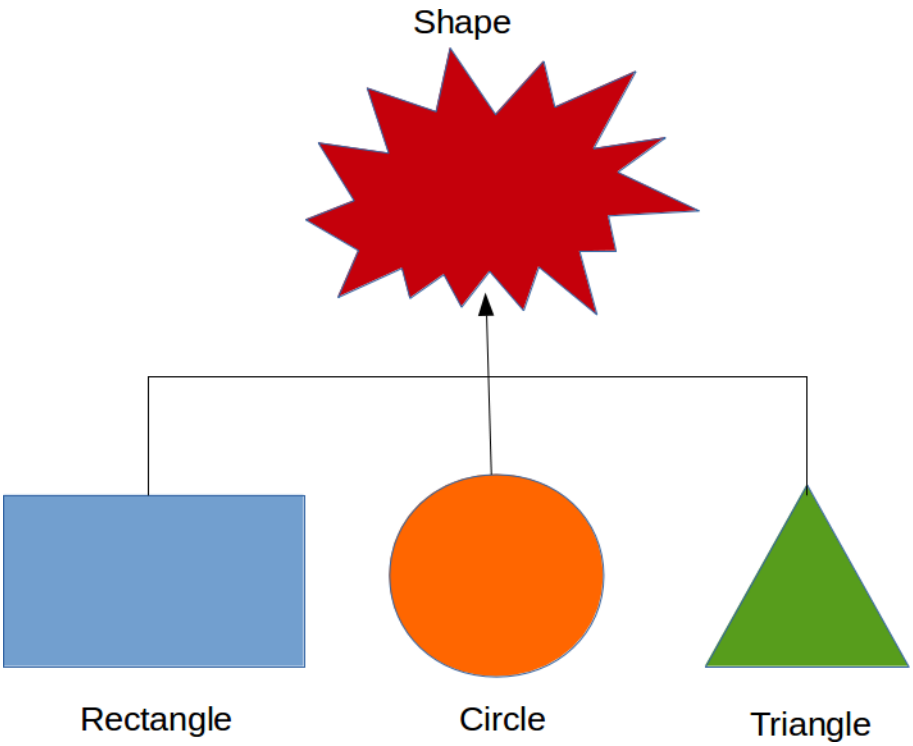
Es una característica de la programación orientada a objetos que denota la posibilidad de asignar un significado diferente para un símbolo particular o «operador» en diferentes contextos.

Un nombre puede referenciar a objetos de diferentes tipos que estn relacionados por características similares. Este nombre debe responder a un conjunto de operaciones y cada operación debe ejecutarse para el tipo adecuado.

En resumen el polimorfismo es la habilidad que tiene diferentes objetos para responder (a su manera) al mismo mensaje

En los lenguajes orientados a objetos el polimorfismo se consigue referenciando (o mediante punteros) al objeto de una subclase por el objeto padre. Las operaciones serán llamadas en la clase base del objeto, pero sern ejecutadas en el objeto correcto automaticamente

En este ejemplo el cliente llama a la función draw() en el objeto padre. La figura correcta se dibuja automáticamente en base a lo que el objeto principal se referencia (podría ser cualquier subclase de Shape)



Hay dos tipos de polimorfismo:

- Static (tiempo de compilación)
- Dynamic (run time)

El **polimorfismo estático** denota que la información requerida esta disponible en tiempo de ejecución. Por lo tanto, las llamadas funciones se pueden resolver en tiempo de compilación. La función exacta a llamar es determinada por el número de parámetros.

El polimorfismo estático se realiza mediante la sobre carga de funciones, operadores de sobrecarga o incluso templates (en c++). Siempre es mas rápido que el dinámico porqué el coste en tiempo de ejecución para resolver que función debe ser llamada se evita.

El **polimorfismo dinámico** denota que la información requerida para llamar la función no se conoce hasta que nos encontramos en tiempo de ejecución. esto se consigue mediante herencia o funciones virtuales.

Si una clase necesita redefinir una función en concreto definido en la clase base, el método es definido como virtual en la clase base y redefinido, para ajustarse a sus necesidades, en la clase derivada. La función virtual en la clase base básicamente define la interficie de la función. Cada clase derivada de ola clase base con las funciones virtuales puede redefinir las funciones con su propia implementación.

Una referencia a una clase base (puntero en C++) se puede usar para apuntar a un objeto de cualquier clase, el método es declarado como virtual en la clase base y redefinido según necesidad. En la clase base se define la interficie de la función. Cada clase derivada de la clase base con métodos virtuales puede redefinir los métodos con su propia implementación.

Dado que la llamada se resuelve en tiempo de ejecución, resultados polimorfismo dinámicos en poco más lenta ejecución del programa.

Observaciones

Espero que este post resulte útil. Es una primera aproximación

a la programación orientada a objetos. En los próximos días introduciremos los principios de diseño de clases. Cabe tener en cuenta que todo esto no es un dogma, mas bien una guía muy útil a la hora de programar.

Ruben.