

UNIDAD 2 - Diseño detallado (estructura) en sistemas de software orientados a objetos usando UML (2ª. PARTE)

Por. Mónica María Rojas Rincón y Norha Milena Villegas Machado

Contenido

UNIDAD 2 - Diseño detallado (estructura) en sistemas de software orientados a objetos usando UML	1
Principios de diseño	1
Acoplamiento y cohesión (coupling and cohesion)	2
Separación de intereses (Separation of concerns).....	4
Paquetes	5
Reutilización	6
Herencia	6
Polimorfismo:	9
Separación de la interfaz de la implementación (separation of interface and implementation)	10

Principios de diseño

Los principios de diseño de software son nociones clave que proporcionan la base para muchos enfoques y conceptos de diseño de software diferentes (SWEBOK, 2014).

Los principales incluyen:

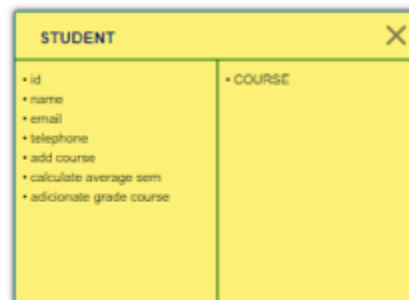
- Abstracción (abstraction)
- Descomposición y modularidad (decomposition and modularization)
- Encapsulamiento/ocultamiento de información (encapsulation/information hiding)
- Acoplamiento y cohesión (coupling and cohesion)
- Separación de intereses (separation of concerns)
- Separación de la interfaz de la implementación (separation of interface and implementation)

En el material anterior, ya mencionamos de que se trata la abstracción, el encapsulamiento y la modularidad. Ahora vamos a ver de qué se tratan los demás y por qué son importantes en nuestros diseños.

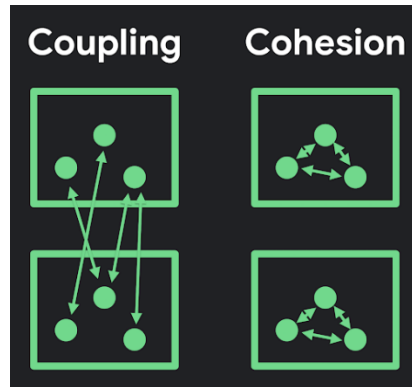
Acoplamiento y cohesión (coupling and cohesion)

El acoplamiento se define como “una medida de la interdependencia entre módulos en un programa de computadora”, mientras que la cohesión se define como “una medida de la fuerza de asociación de los elementos dentro de un módulo”.

Al realizar diseños de software, se busca alta cohesión y bajo acoplamiento. Cuando determinamos las responsabilidades de una clase, y colocamos sus características y métodos, lo que se busca es aumentar la cohesión. Entendiendo por cohesión, como lo estrecha que es la relación entre los componentes de algo. Si hablamos de clases, una clase tendrá una cohesión alta si sus métodos están relacionados entre sí, es decir, tienen una “temática” común.



Mientras que el acoplamiento es la manera que se relacionan varios componentes entre ellos. Si existen muchas relaciones entre los componentes, con muchas dependencias entre ellos, tendremos un grado de acoplamiento alto. En este ejemplo hay acoplamiento entre la clase STUDENT y COURSE, porque para poder desarrollar STUDENT sus responsabilidades, requiere de COURSE. Al aumentar el acoplamiento se puede dificultar el mantenimiento del software, porque si es necesario realizar un cambio, se tendría que hacer cambios en otros componentes con los que se relaciona el componente a modificar.

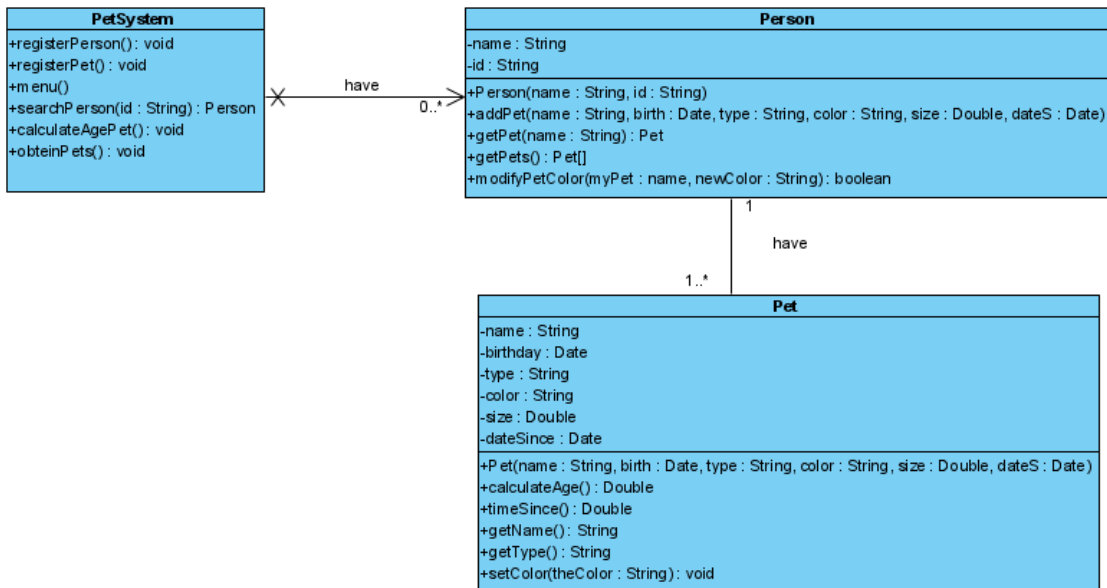


Fuente: <https://ichi.pro/es/comprender-jetpack-compose-parte-1-de-2-222313679392848>

Ejemplo.

Considerando el siguiente diagrama de clases, ¿qué ocurriría si fuera necesario adicionar un atributo a la mascota? por ejemplo si se deseara además de la información actual, registrar el tipo de alimentación (typeFood).

1. ¿En qué clases se debe modificar el código, para implementar este cambio?



Sería necesario modificar, Pet, Person, y PetSystem.

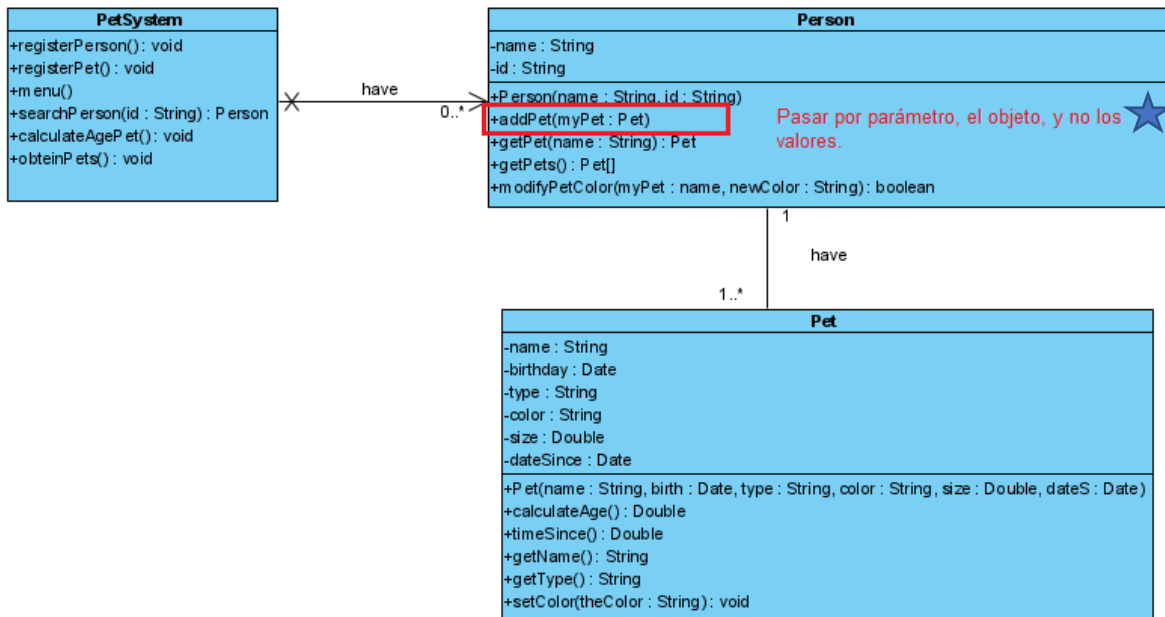
2. Identifique qué cambios de deben realizar en cada clase.



¿Cómo se puede mejorar el diseño anterior para disminuir los efectos de un cambio?

Solución:

Se puede en PetSystem, luego de pedir los datos al usuario, instanciar el objeto de tipo Pet, y se le pasa al método addPet, el objeto en vez de pasarle sus valores. Así solo sería necesario modificar PetSystem donde se piden los datos al usuario y la clase Pet, para adicionar el nuevo atributo.



★ Se estaría aplicando **abstracción por parámetros**, al mandar el objeto, el cual tiene encapsulados los parámetros, y no la lista de parámetros.

Separación de intereses (Separation of concerns)

Separar las preocupaciones por puntos de vista permite a las partes interesadas concentrarse en unas pocas cosas a la vez y ofrece un medio para gestionar la complejidad.

La separación de intereses se logra a través del encapsulamiento y la separación en capas (definición de arquitecturas de varias capas, por ejemplo, cuando se definen múltiples

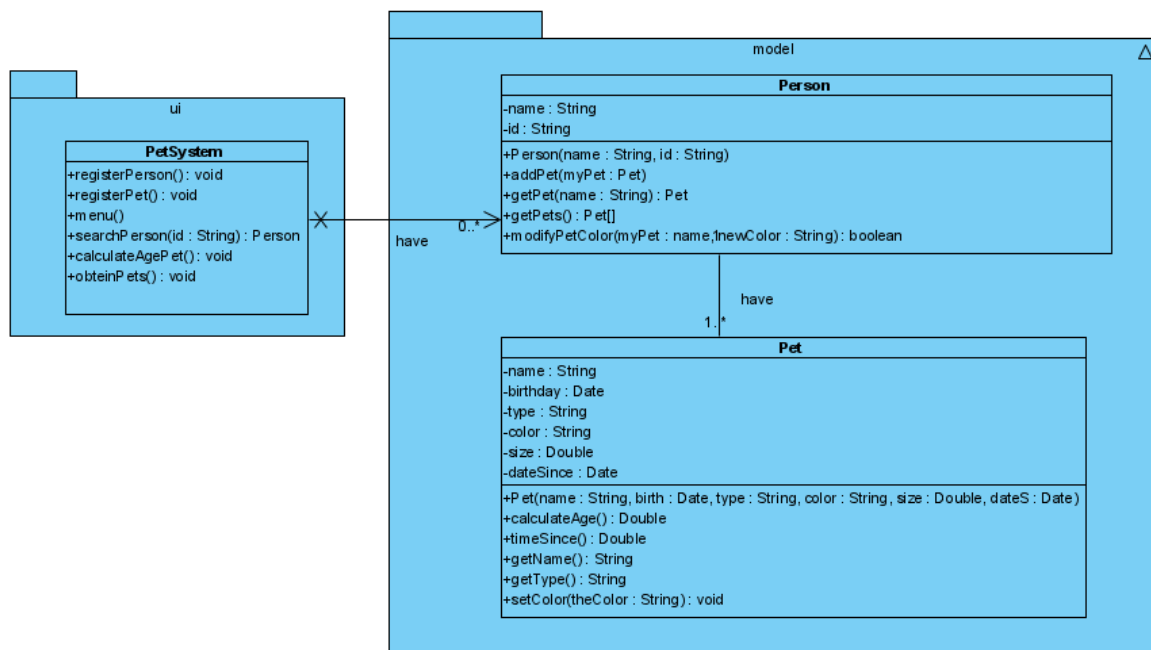
paquetes). La separación de intereses favorece la reutilización y facilita el mantenimiento del software.

Cuando se divide un programa en paquetes, por ejemplo, el paquete de la interaz (ui) se separa del modelo (model) se favorece el mantenimiento de software, porque conservando el mismo código del modelo, se podría implementar una interfaz de usuario diferente, logrando desacoplar la vista del modelo.

Paquetes

Los paquetes, son una forma de organizar grupos de clases. Un paquete contiene un conjunto de clases relacionadas bien por finalidad, por ámbito o por herencia. Las clases dentro de un paquete son únicas, esto permite también ocultar clases y encapsular funcionalidades con cierto grado de independencia.

Ejemplo:



```

Package ui;

Public class PetSystem{

    ...

}
    
```

Reutilización

Uno de los principales objetivos de la programación orientada a objetos es la reutilización, no se debe confundir con clonación de código, lo cual es un antipatrón y debe ser evitado. Algunas estrategias para llevar a cabo la reutilización, son: la herencia y el polimorfismo.

Hay otras situaciones, en las que hay otros mecanismos mejores que la herencia para la reutilización, pero serán vistos más adelante.

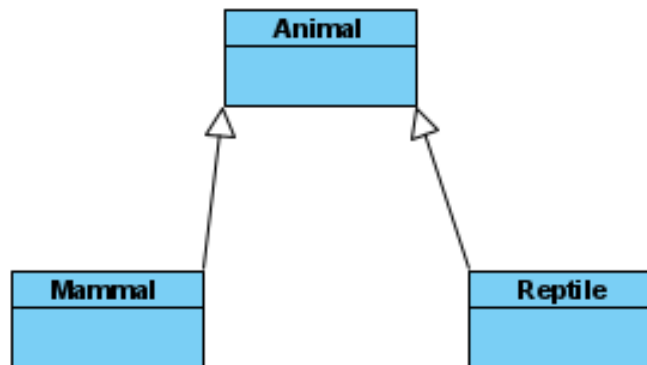
Herencia

Es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase.

La herencia permite que se puedan definir nuevas clases basadas de unas ya existentes a fin de reutilizar el código, generando así una jerarquía de clases dentro de una aplicación. Si una clase deriva de otra, esta hereda sus atributos y métodos y puede añadir nuevos atributos, métodos o redefinir los heredados.

Ejemplo:

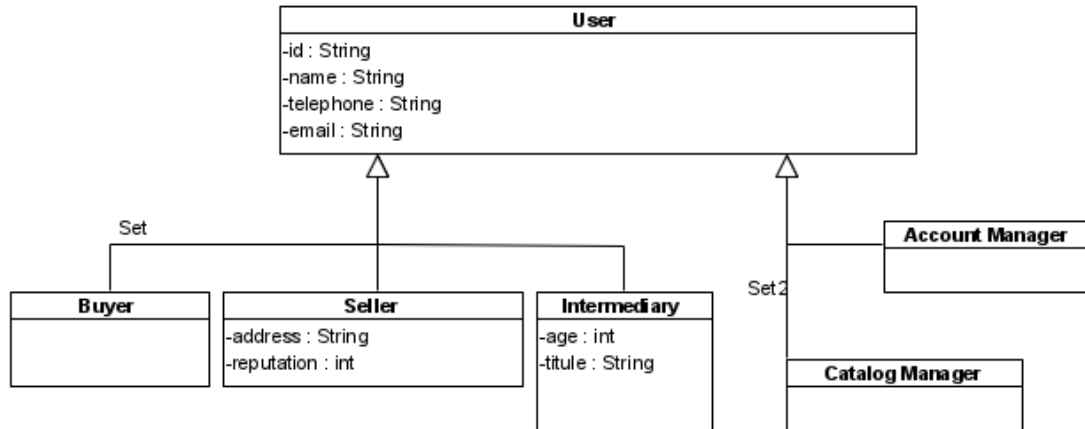
En este ejemplo, tenemos una clase padre Animal, y unas clases hijas Mamífero y Reptil. Lo que esto significa es que la implementación que tiene la clase Animal, es heredada por las subclases (clases hijas).



- **¿Tienen iguales atributos?** Los atributos comunes deben estar en la clase padre, y lo que es específico en las clases hijas.
- **¿Tienen igual comportamiento?** Los métodos que son comunes para todas las clases se deben colocar en la clase padre, y los específicos estarían en las subclases. Toda las funcionalidades definidas en la clase padre Animal, estarían disponible para las hijas: Mammal y Reptile.

Jerarquías de herencia: Mammal y Reptile son de familias diferentes, entonces se dejan cada una con flecha.

Ejemplo, se definen que tenemos unos usuarios de tipo administrador mientras otros son de aplicación. Cada flecha, significa una jerarquía de herencia.



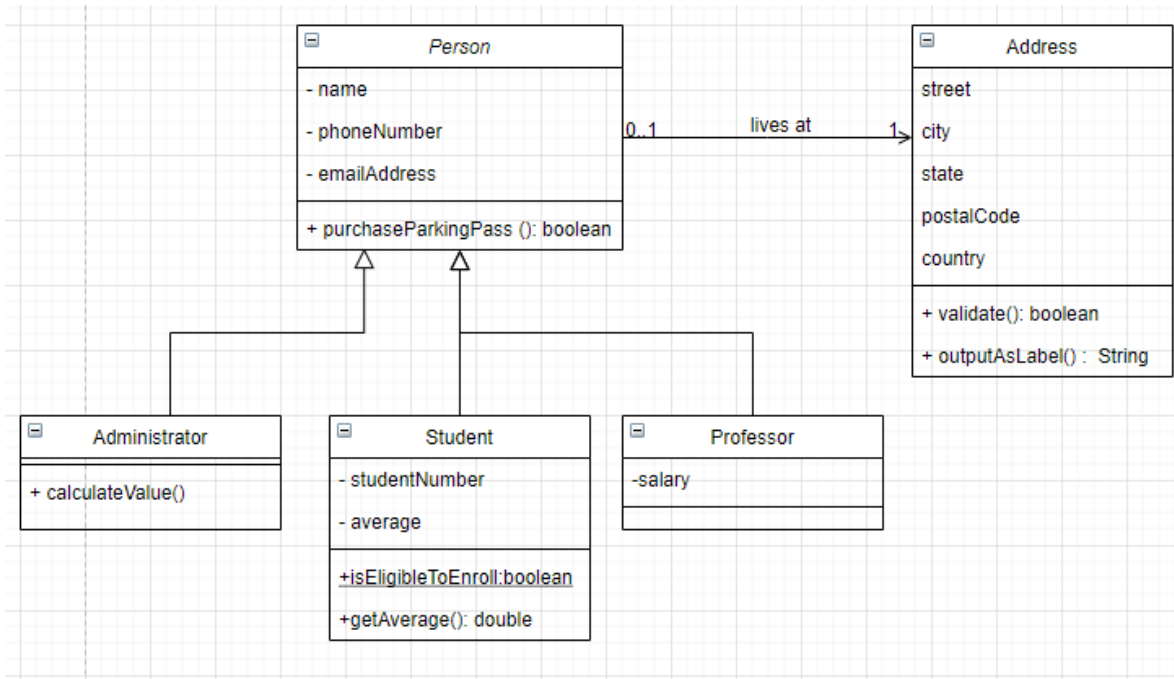
Modificador de acceso: Protegido (Protected) #

Significa que son visible para todas las clases que hay dentro del paquete y para las subclases (así estén en paquetes diferentes).

El modificador de acceso por default en Java es de nivel de paquete.

Modificador/Acceso	Clase	Paquete	Subclase	Todos
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
default	Sí	Sí	No	No
private	Sí	No	No	No

UNIDAD 2 - DISEÑO DETALLADO (ESTRUCTURA) EN SISTEMAS DE SOFTWARE ORIENTADOS A OBJETOS USANDO UML



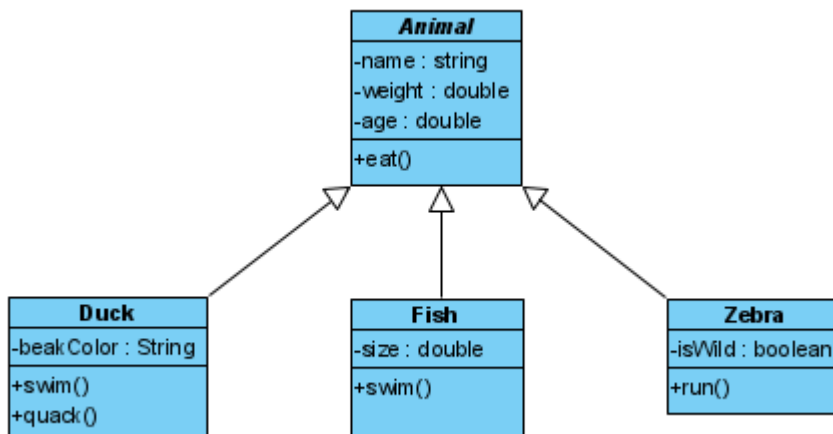
En este ejemplo, tenemos una clase padre **Person**, y tres clases hijas, que están en dos jerarquías de herencia diferente.

Clase abstracta: es una “Clase incompleta”.

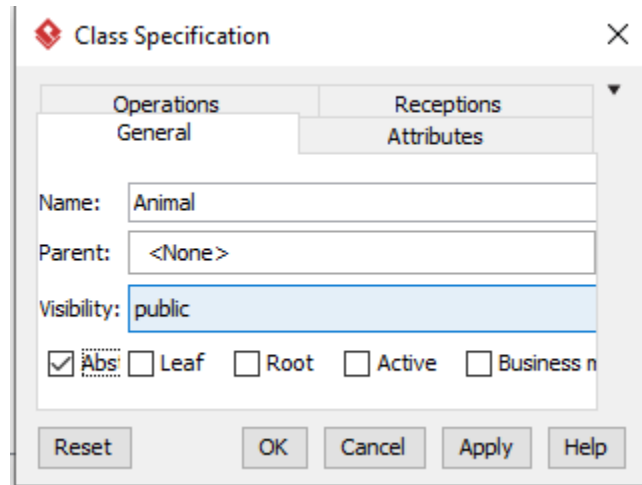
Si se define la clase padre como abstracta, quiere decir que de esa clase no se puede crear instancias, solo de las clases hijas se podría instanciar objetos.

En Visual Paradigm se identifican porque el nombre queda en cursiva.

Ejemplo: la clase padre, se definió como abstracta porque solo me va a servir para asociar lo que tienen en comun, pero no tiene sentido crear objetos de tipo **Animal**, porque deben ser de algún tipo específico; **Duck**, **Fish** o **Zebra**. Entonces en este caso se debe definir como clase abstracta.



Para hacerlo en Visual Paradigm, se ingresa a la especificación de la clase y se marca como Abstracta.



Polimorfismo:

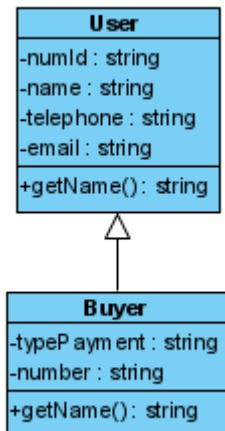
Capacidad que tienen los objetos de comportarse de formas diferentes. Promueve la reutilización.

Una forma de implementarse es a través de la sobreescritura de métodos y del polimorfismo por sobrecarga.

Sobreescritura de métodos:

Una clase hija puede sobrecribir un método de la clase padre. De esta forma se puede reutilizar código pero con algunos cambios, extender su funcionalidad y personalizar el método que ya existía en la superclase.

Por ejemplo, en User está el método getName() en la clase Buyer se puede sobrecribir.



Se puede reescribir getName() en Buyer, para que muestre:

Andrea es un comprador.

Polimorfismo por sobrecarga

Cuando existen métodos con el mismo nombre pero con parámetros diferentes. No puede tener la misma signatura (nombre y parámetros).

Por ejemplo podemos definir el operador suma (+), llamado igual en dos clases independientes, pero en una de ellas suma números y en otra suma caracteres produciendo concatenación.

Ejemplo:

Tenemos tres métodos display, pero cada uno de ellos tiene una firma diferente, es dependiendo de los parámetros se determina cuál se ejecuta.

```

void display(int var1, double var2) { ←
    // code
}

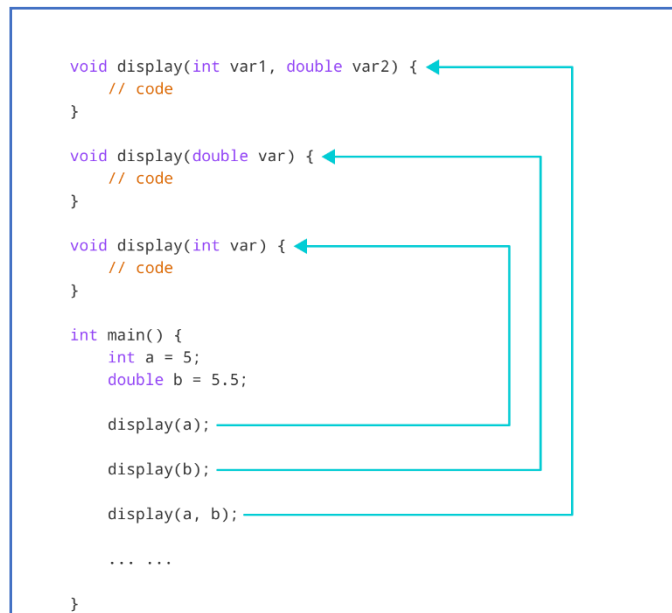
void display(double var) { ←
    // code
}

void display(int var) { ←
    // code
}

int main() {
    int a = 5;
    double b = 5.5;

    display(a);
    display(b);
    display(a, b);

    ... ..
}
    
```



Separación de la interfaz de la implementación (separation of interface and implementation)

Separar la interfaz y la implementación implica definir un componente especificando una interfaz pública (conocida por los clientes) que está separada de los detalles de cómo se realiza el componente (ver encapsulación y ocultamiento de información arriba).

Referencias:

SWEBOK. Pag. 52