6. Relaciones entre clases

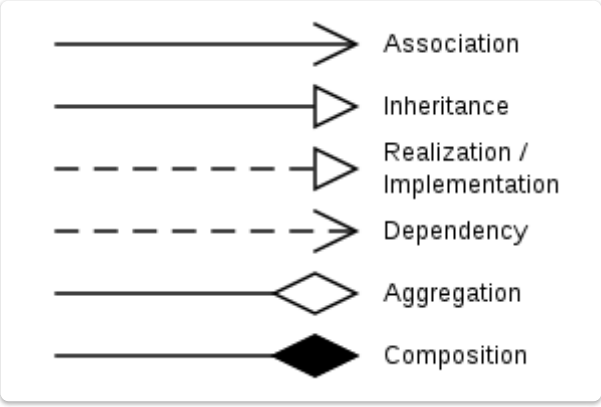
> *Tutor Isaac Palma Medina, CC BY-SA 4.0*
> *Versión 12/07/2023*

> *Los malos programadores se preocupan por el código. Los buenos programadores se preocupan por las estructuras de datos y sus relaciones. - Linus Torvalds*

Cuando se tienen objetos que interactúan entre sí, surgen relaciones entre ellos. Una forma común de representar estas relaciones es mediante diagramas UML, que proporcionan una visión clara de la estructura de una clase.

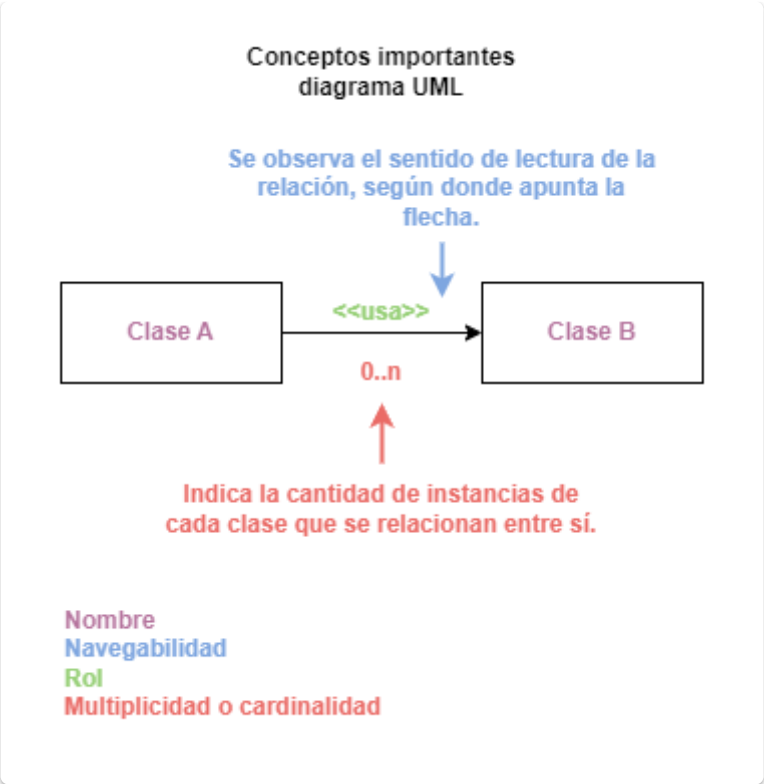
Entre las relaciones estudiadas en UML se incluyen:

- Dependencia.
- Asociación.
- Agregación.
- Composición.
- Herencia.
- Herencia abstracta.



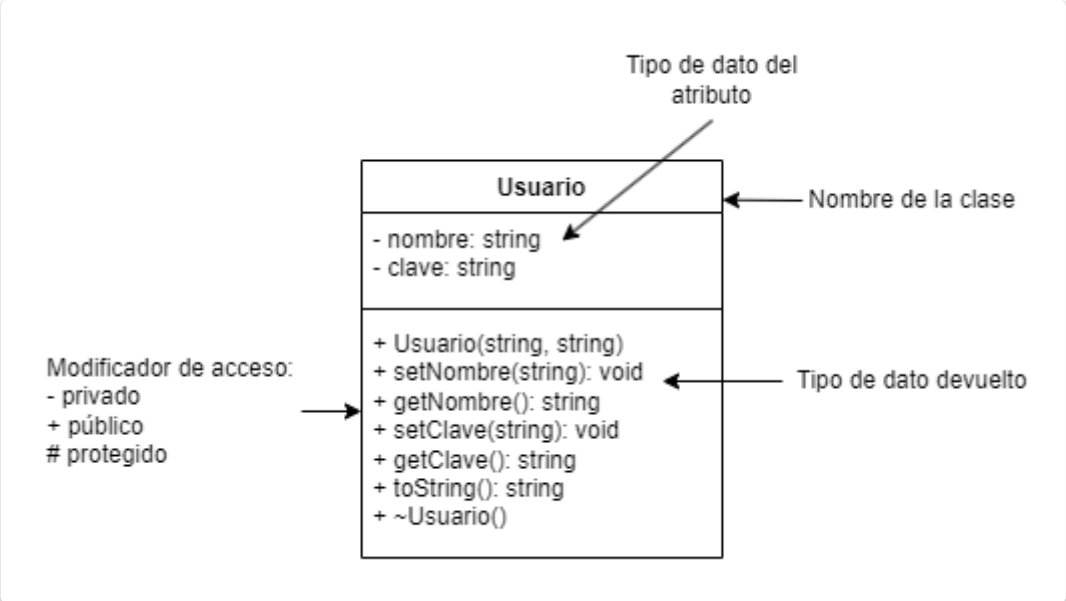
[1]

Algunos conceptos importantes al momento de representar relaciones en diagramas, son los siguientes:



^Elaboración propia

Representación UML de una clase



^Elaboración propia

Para aprender más sobre diagramas UML, le recomendamos consultar este enlace:

- UML - Guía rápida

Si desea generar diagramas UML a partir de código C++ de manera sencilla, puede utilizar esta aplicación:

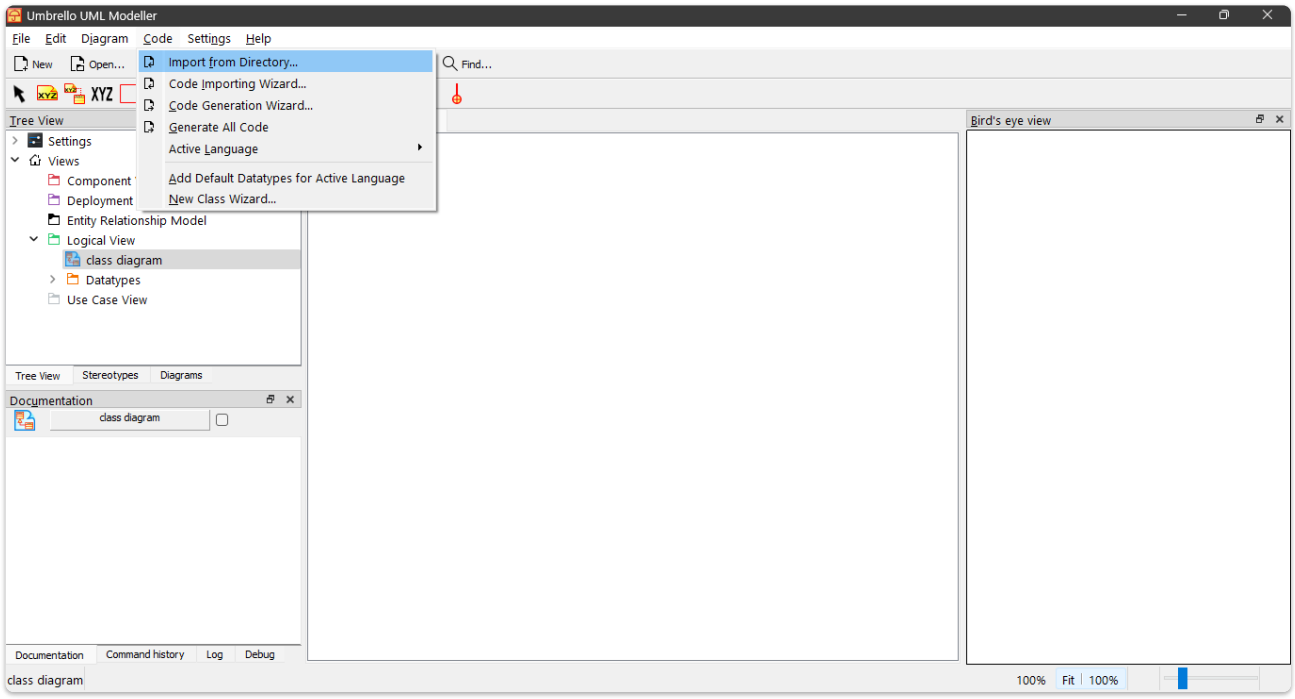
- Enlace de información: Umbrello
- Enlace de descarga directo (Windows).

> *Los diagramas generados por esta aplicación, son **propuestas**, y no necesariamente están totalmente correctos. Úsese con precaución y honestidad.*

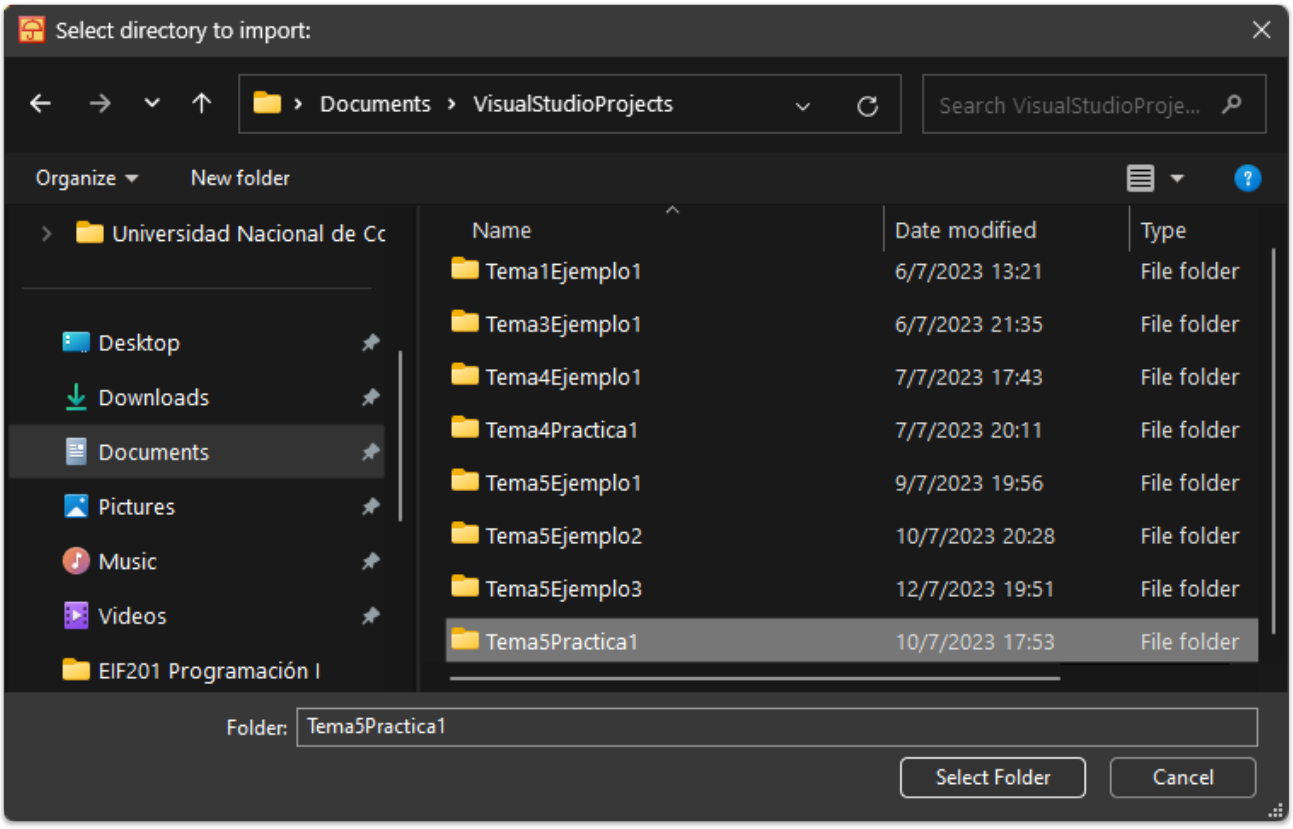
¿Cómo generar un diagrama?

En este caso se usará la práctica 1 de la sección pasada.

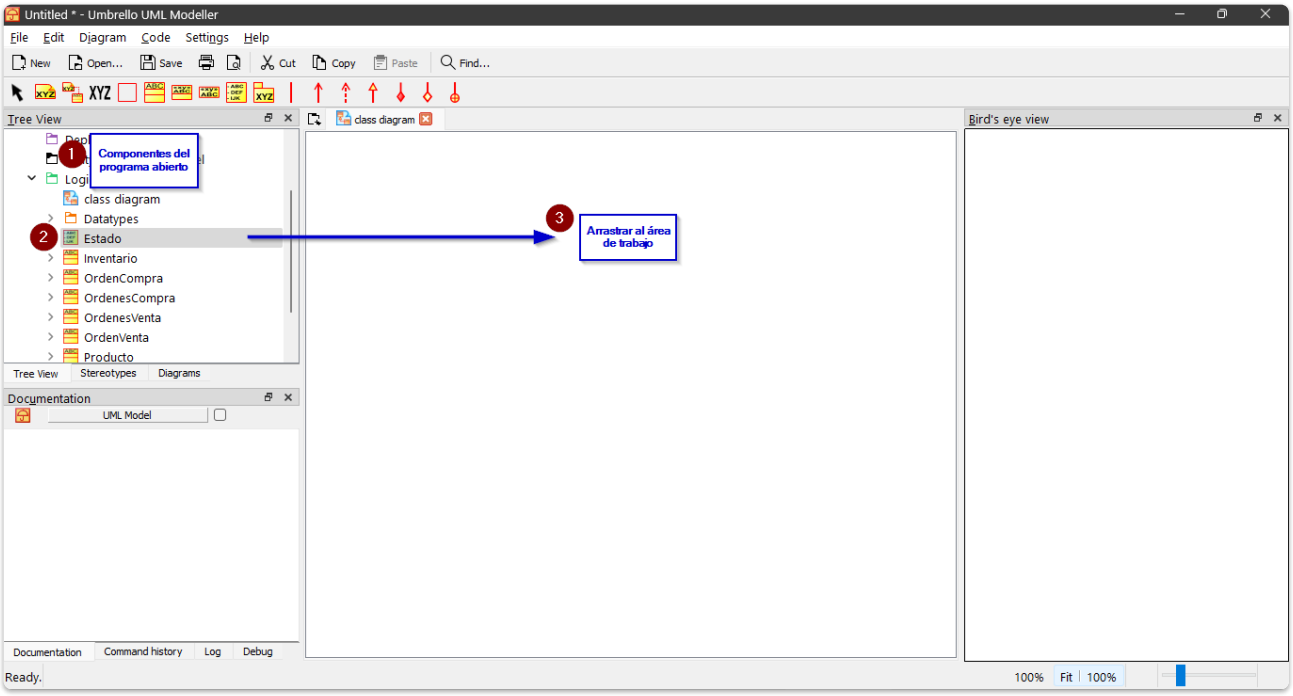
1. Programar la solución de la cual se desea generar el diagrama.
2. Instalar y abrir Umbrello.
3. En la pestaña Code, seleccionar la opción Import from directory.



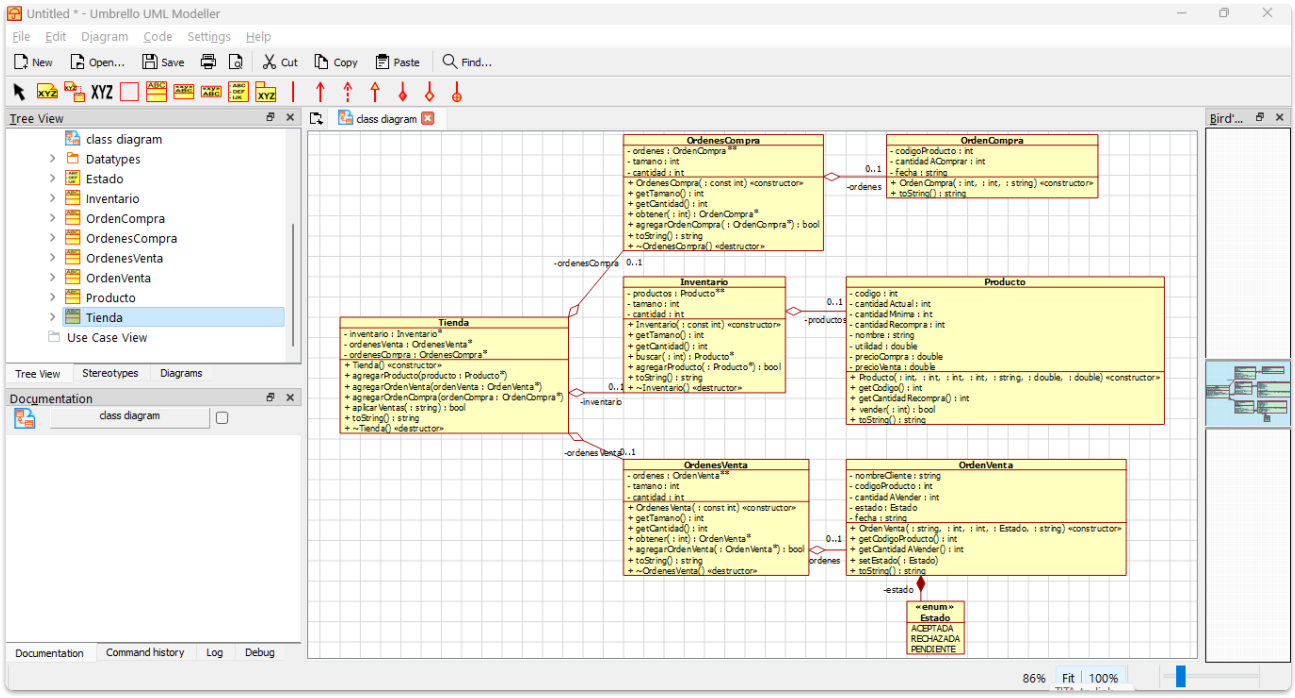
4. Seleccionar la carpeta del proyecto.



5. En el panel de la izquierda arrastrar todas las clases al área de trabajo, el programa automáticamente generará las relaciones entre estas. Deberá ir acomodando los componentes hasta lograr un resultado estético.

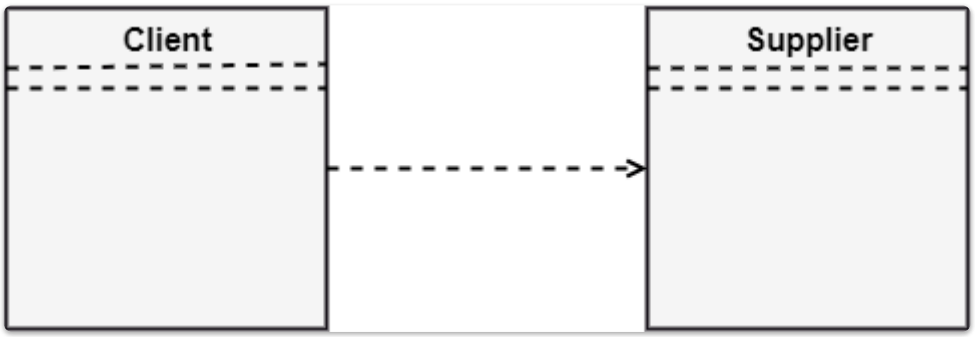


6. Verificar que el diagrama sea correcto, hacer cambios si lo considera necesario. Puede usar otro programa para dar una mejor presentación, como lo es Draw.io.



Relaciones

Dependencia



[2]

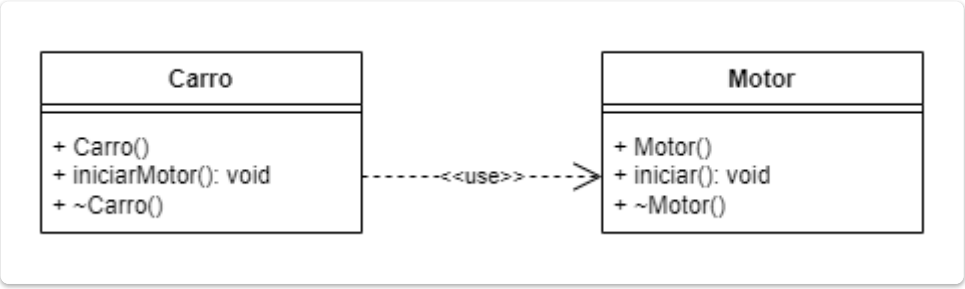
Es una relación en la que un elemento, el cliente, utiliza o depende de otro elemento, el proveedor. La relación de dependencia indica que la clase cliente realiza una de las siguientes funciones[3] :

- Utiliza temporalmente una clase proveedora de ámbito global.
- Utiliza temporalmente una clase proveedora como parámetro de una de sus operaciones.
- Utiliza temporalmente una clase proveedora como variable local para una de sus operaciones.
- Envía un mensaje a una clase proveedora.

Existen tipos que describen la relación de dependencia[3-1] :

Tipo	Descripción
`«abstraction»`, `«derive»`, `«refine»`, o `«trace»`	Relaciona dos clases, indicando que ambas representan el mismo concepto, en diferentes niveles de abstracción.
`«bind»`	Conecta interfaces, para crear clases.
`«realize»`	Indica que el cliente es una implementación del proveedor.
`«substitute»`	Indica que el cliente toma el lugar del proveedor, por medio de alguna interfaz.
`«use»`, `«call»`, `«create»`, `«instantiate»`, o `«send»`	Indica que un elemento del modelo requiere otro elemento del modelo para su plena aplicación o funcionamiento.

Ejemplo



^Elaboración propia

```
class Motor {
public:
        Motor() {}
        void iniciar() {
                cout << "\nRrrr Rrrr Rrrr";
        }
        ~Motor() = default;
};

class Carro {
```

```
public:
        Carro() {}
        void iniciarMotor() {
                Motor* motor = new Motor();
                motor->iniciar();
                delete motor;
        }
        ~Carro() = default;
};
```

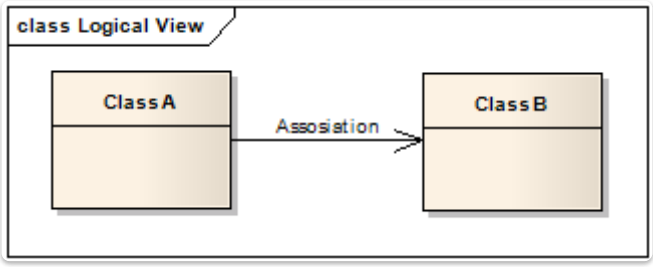
Se muestra una clase `Carro` que depende de la clase `Motor` para poder iniciar su motor. En el método `iniciarMotor()` de la clase `Carro`, se crea una instancia de la clase `Motor` y se llama a su método `iniciar()` para iniciar el motor.

La dependencia en este caso significa que la clase `Carro` utiliza la funcionalidad proporcionada por la clase `Motor`, pero no hay una relación permanente o una asociación fuerte entre ambas clases.

La clase `Carro` depende de la existencia y el funcionamiento de la clase `Motor` en un contexto específico, pero no hay una relación de pertenencia o una relación "todo-parte" entre ellas.

> *La flecha apunta desde el elemento que depende hacia el elemento en el que se basa.*

Asociación



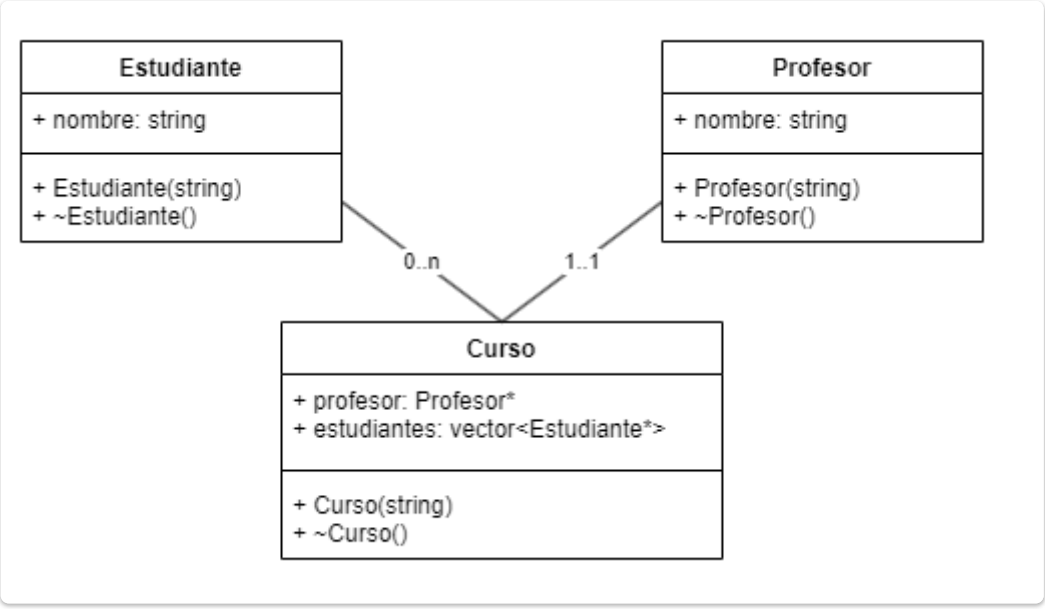
[4]

Una asociación representa una relación estructural que conecta dos clases[5].

En las relaciones entre clases, se pueden usar asociaciones para mostrar las decisiones de diseño que se han tomado acerca de las clases en una aplicación que contiene datos, y para mostrar qué clases necesitan compartir datos[5-1].

Suele ser la manera más abstracta de representar una relación entre clases, por lo que en el fondo, muchas relaciones son de asociación[4-1].

Ejemplo



^Elaboración propia

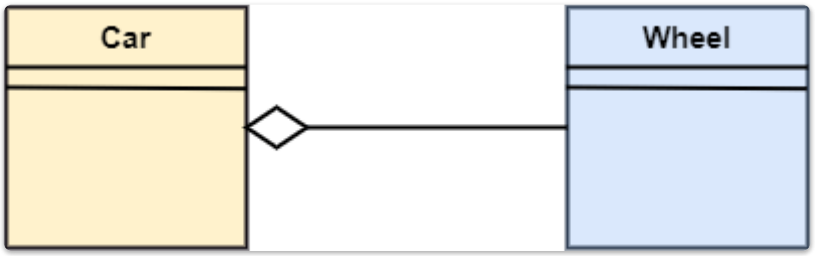
```
class Profesor {
private:
        string nombre;
public:
        Profesor(string nombre) {
                this→nombre = nombre;
        }
        ~Profesor() = default;
};

class Estudiante {
private:
        string nombre;
public:
        Estudiante(string nombre) {
                this→nombre = nombre;
        }
        ~Estudiante() = default;
};

class Curso {
```

```
private:
        Profesor* profesor;
        vector<Estudiante*>* estudiantes;
public:
        Curso(Profesor profesor) {
                this->profesor = profesor;
        }
        ~Curso() = default;
}
```

Se observa una relación más general, donde las instancias de las clases `Profesor` y `Estudiante` pueden existir de forma independiente, y no se destruyen si una instancia de `Curso` se destruye. Ninguna de las clases depende directamente de las demás.

> *Por facilidad, la direccionalidad de la flecha puede omitirse. Sin embargo, en casos especiales, como una asociación unidireccional (solo un elemento puede acceder a otro), se puede agregar una flecha que apunte hacia el elemento que puede navegar en la relación (hacia al que accede). Si no hay flechas, la asociación se considera bidireccional.*

Agregación



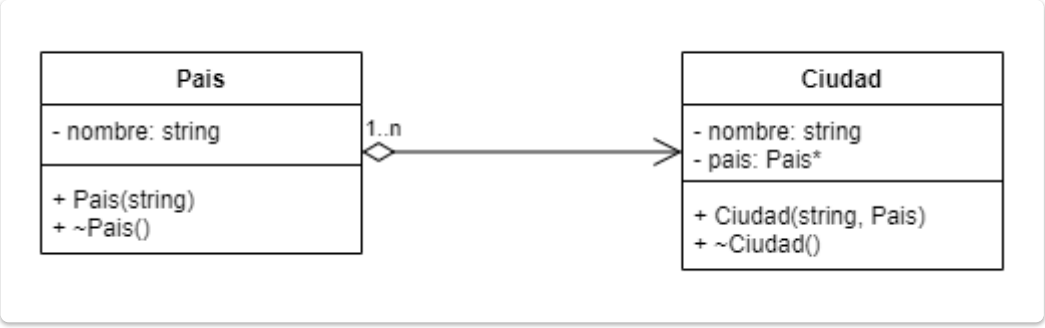
[6]

En una relación de "todo-parte", las partes pueden pertenecer a más de un todo al mismo tiempo y pueden existir de manera independiente al todo. Si se destruye el todo, las partes no necesariamente se destruyen.

Una agregación es un tipo especial de asociación en la cual los objetos se ensamblan o configuran juntos para crear un objeto más complejo. La agregación describe un grupo de objetos y cómo interactúan entre sí[7].

Es por esto, que el ejemplo anterior (profesores, estudiantes y cursos) no es una relación de agregación, ya que no hay una relación "todo-parte" tan estricta y la relación es más general.

Ejemplo

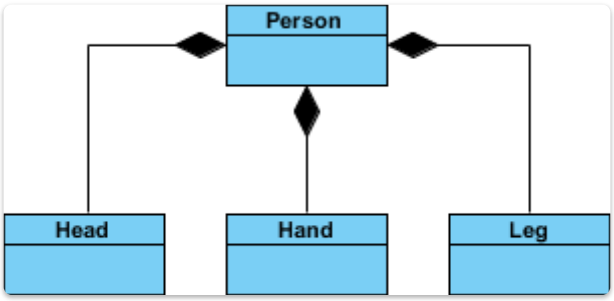


^Elaboración propia

```
class Pais {
private:
        string nombre;
public:
        Pais(string nombre) {
                this→nombre = nombre;
        }
        ~Pais() = default;
};

class Ciudad {
private:
        string nombre;
        Pais* pais;
public:
        Ciudad(string nombre, Pais* pais) {
                this→nombre = nombre;
                this→pais = pais;
        }
        ~Ciudad() = default;
};
```

> *En una agregación, la flecha apunta desde el elemento que contiene hacia el elemento que está contenido. Por lo tanto, la flecha apunta desde el contenedor (todo) hacia el contenido (parte).*

Composición



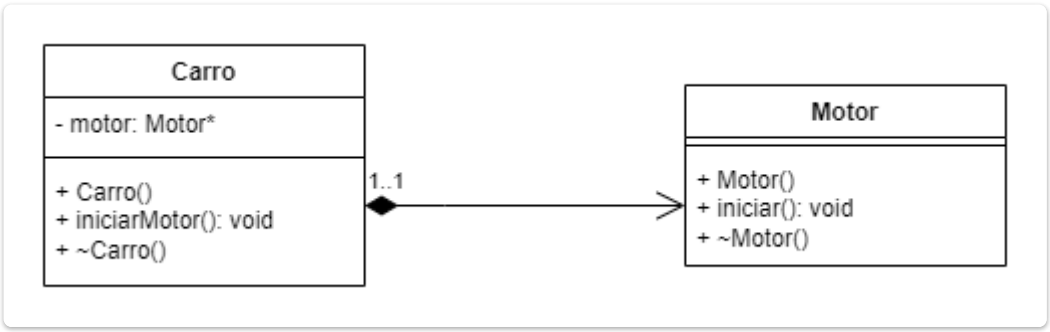
[8]

Una relación de composición representa una relación de todo-parte y es una forma de agregación. Esta relación especifica que la duración de la parte depende de la duración del todo[9].

Por ejemplo, una relación de asociación de composición conecta una clase de `Estudiante` con una clase de `Horario`, lo que significa que si se elimina al estudiante, también se elimina el horario.

Esto puede verse aplicado de manera más clara en el ejemplo del carro usado inicialmente.

Ejemplo

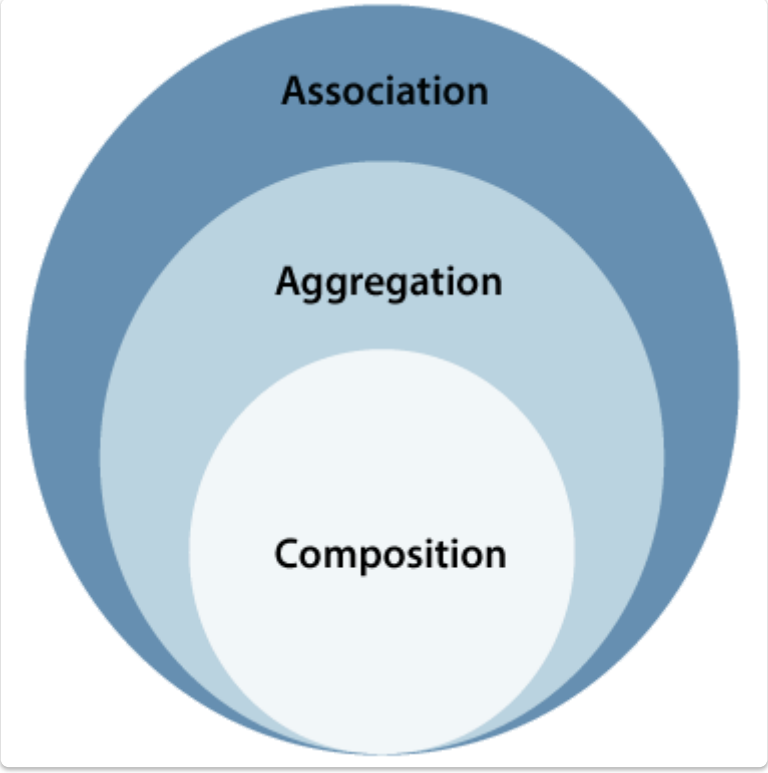


^Elaboración propia

```
class Motor {
public:
        Motor() {}
        void iniciar() {
                cout << "\nRrrr Rrrr Rrrr";
        }
        ~Motor() = default;
};
```

```
class Carro {
private:
        Motor* motor;
public:
        Carro() {
                this->motor = new Motor();
        }
        void iniciarMotor() {
                motor->iniciar();
        }
        ~Carro() {
                delete motor;
        }
};
```

Composición vs. agregación vs. asociación



[6-1]

Referencias

1. Contributors to Wikimedia projects. (2023). *Class diagram.* Recuperado de https://en.wikipedia.org/w/index.php?title=Class_diagram&oldid=1137357004
2. Javatpoint. (2023). *UML Dependency.* Recuperado de https://www.javatpoint.com/uml-dependency
3. IBM Documentation. (2021). *Dependency relationships.* Recuperado de https://www.ibm.com/docs/en/rsar/9.5?topic=diagrams-dependency-relationships
4. Ezra, A. (2023). *UML Class Diagram: Association, Aggregation and Composition.* Recuperado de http://aviadezra.blogspot.com/2009/05/uml-association-aggregation-composition.html
5. IBM Documentation. (2021). *Association relationships.* Recuperado de https://www.ibm.com/docs/en/rsm/7.5.0?topic=ricd-association-relationships
6. Javatpoint. (2023). *UML Association vs. Aggregation vs. Composition.* Recuperado de https://www.javatpoint.com/uml-association-vs-aggregation-vs-composition
7. IBM Documentation. (2021). *Aggregation relationships.* Recuperado de https://www.ibm.com/docs/en/rsm/7.5.0?topic=diagrams-aggregation-relationships
8. Visual Paradigm. (2023). *UML Association vs Aggregation vs Composition.* Recuperado de https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition
9. IBM Documentation. (2021). *Composition association relationships.* Recuperado de https://www.ibm.com/docs/en/rsm/7.5.0?topic=diagrams-composition-association-relationships