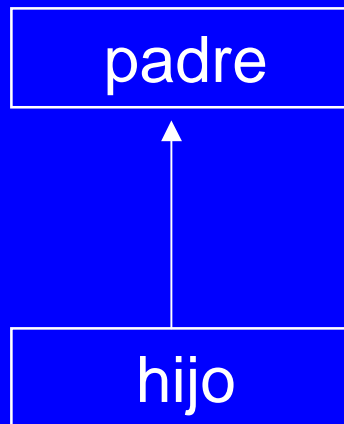


Herencia

Agustín J. González
Versión original de Kip Irvine
ELO320: Seminario II
2do. Sem 2001

Clasificación de Objetos

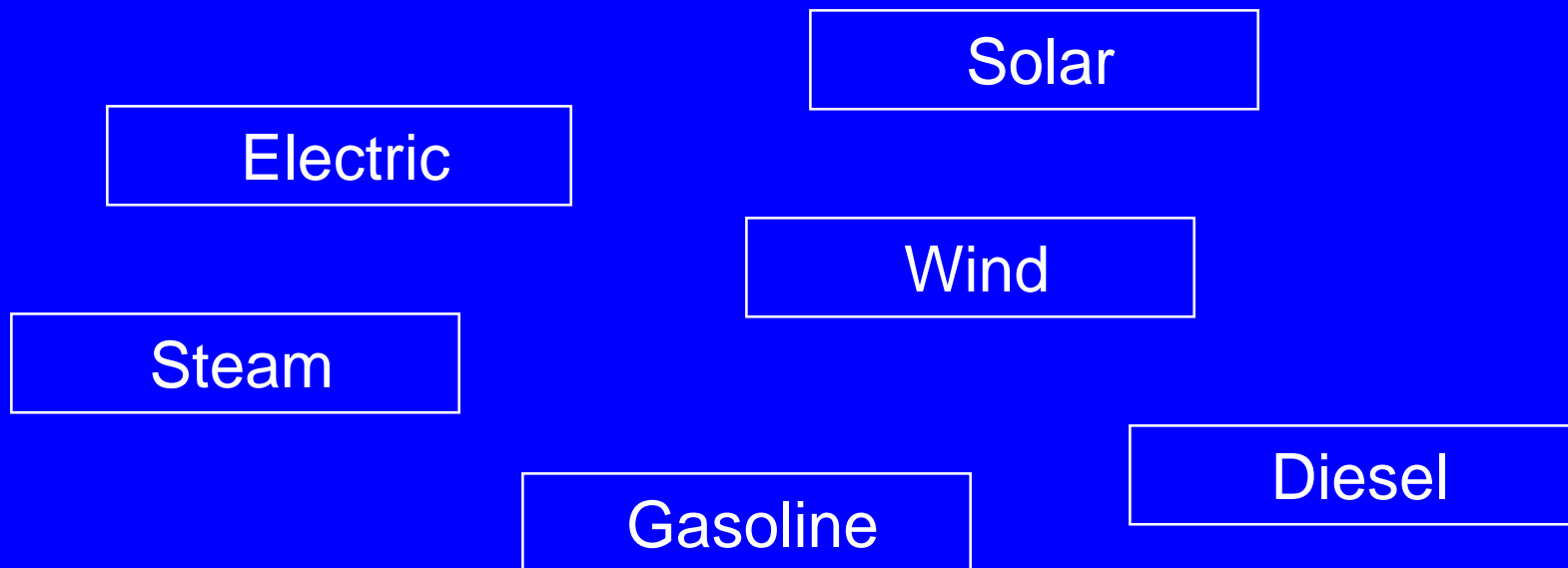
En un intento en modelar el mundo real los diseñadores clasifican los objetos. Herencia implica una relación en la cual ciertas características son heredadas.



Como en biología, los niños heredan características de su padre

Tipos de Motores

Claramente todos estos casos están relacionados.
¿Qué podrían todos estos motores tener en común?



Motores

La información común puede ser descrita en una clase base llamada **Motor**:

Motor class:

Serial number

Manufacturer

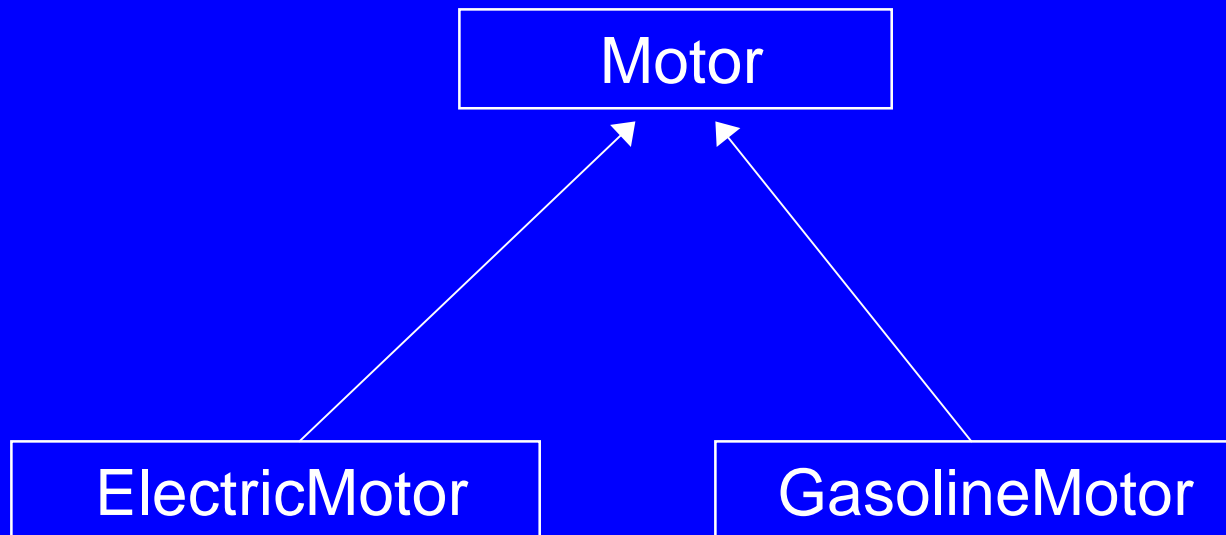
Date of Manufacture

Model Number

etc.

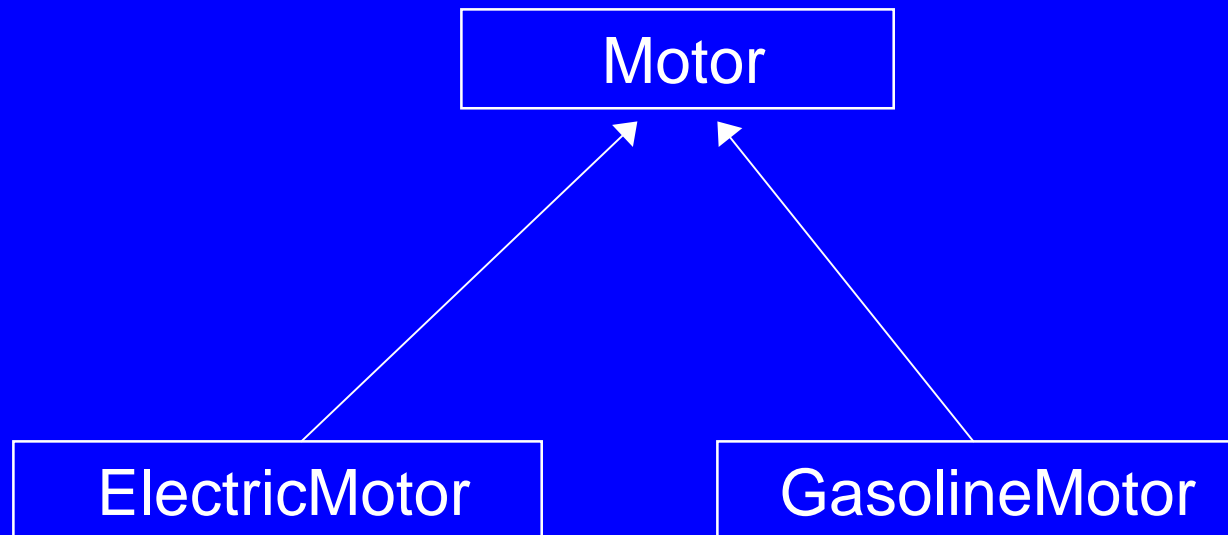
Relación de Herencia

Las flechas muestran que las clases ElectricMotor y GasolineMotor son **clases derivadas** de la clase Motor.



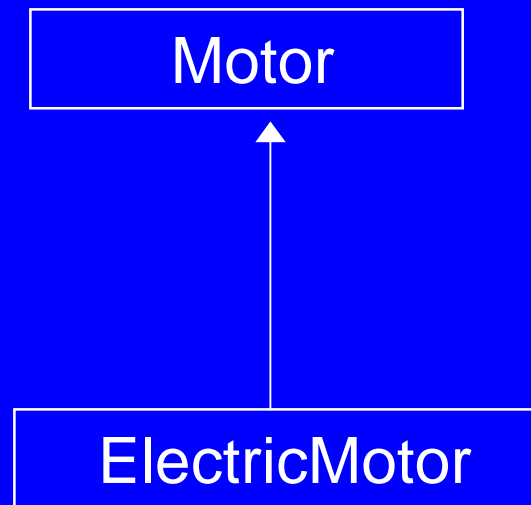
Relación de Herencia...

La clase Motor es llamada la **clase base**, y las otras dos son llamadas **clases derivadas**.



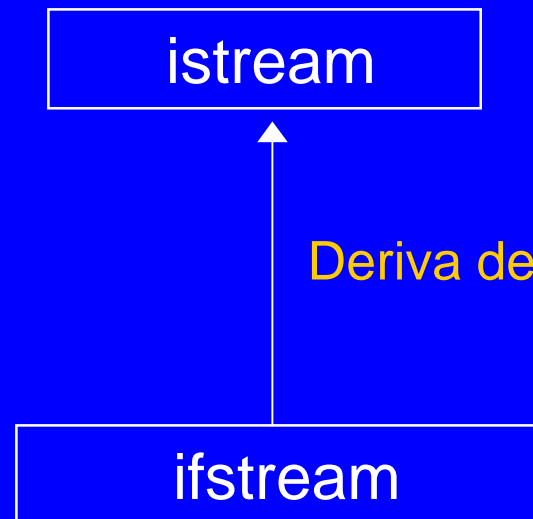
Herencia (Es-un)

Herencia es frecuentemente llamada relación *es-un*. En otras palabras, "un ElectricMotor *es un* Motor."



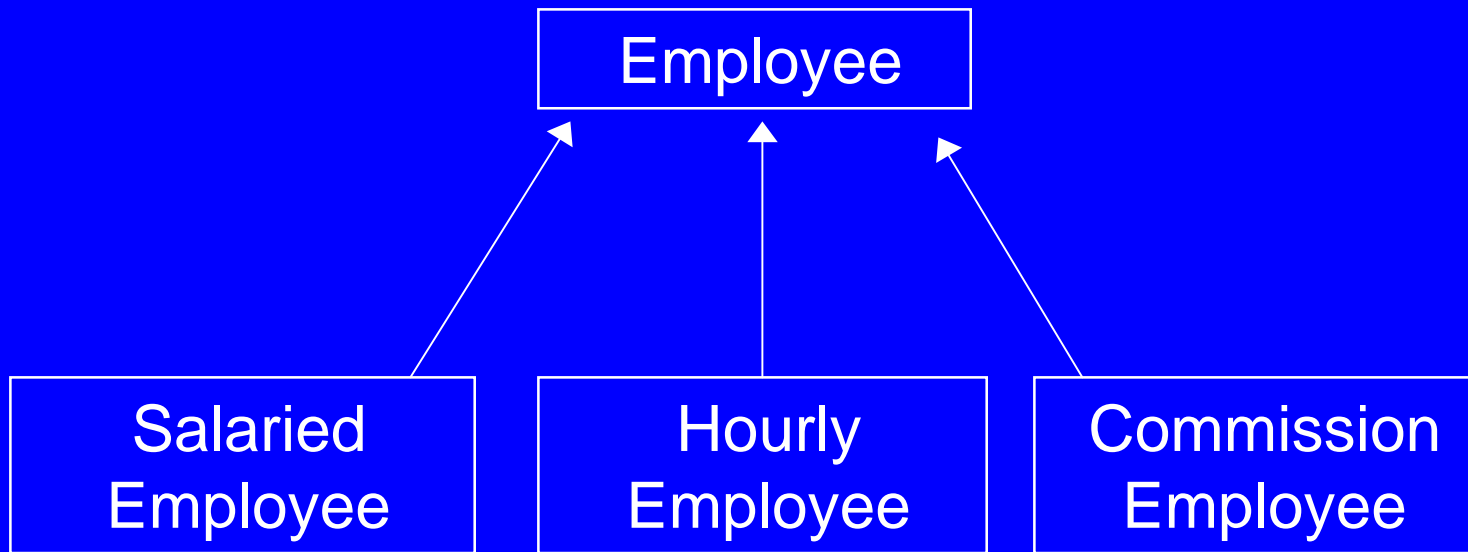
Relación de Herencia...

Muchas otras clases tienen esta relación, tales como las clases istream (flujo de entrada) y ifstream (flujo de entrada de archivo).



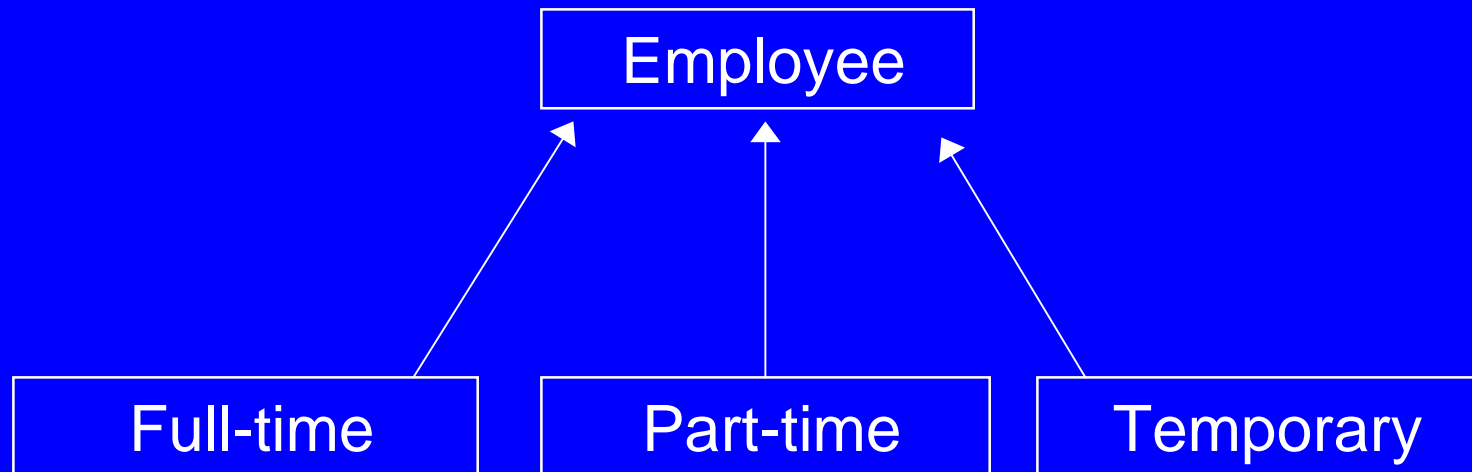
Ejemplo: Empleados

Podríamos clasificar empleados basados en cómo se les paga ...



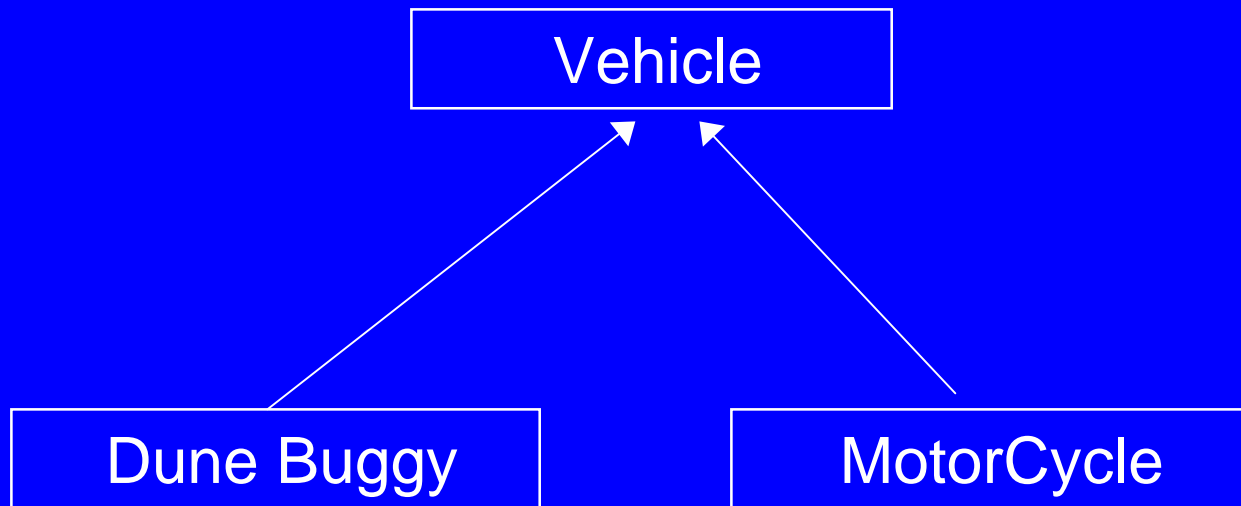
Tipos de Empleados

...o podríamos clasificarlos basados en su estatus:



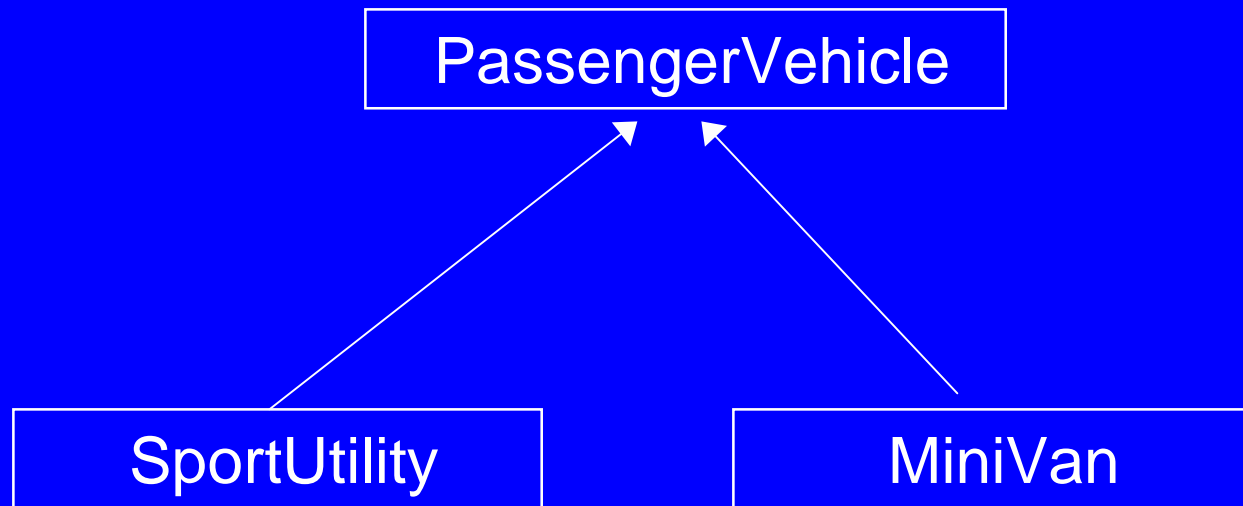
Relaciones de Herencia...

Algunas relaciones son un tanto remotas....

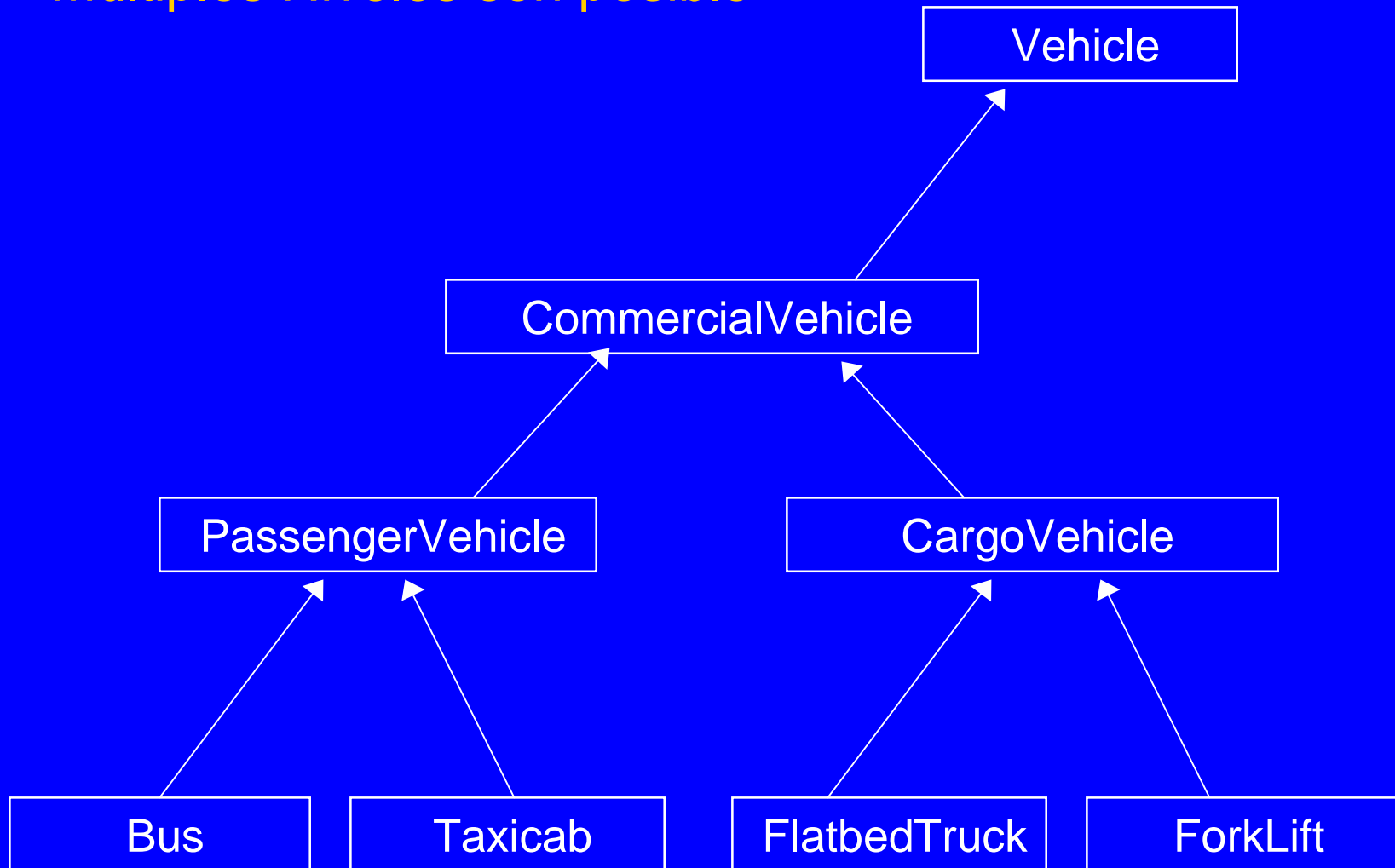


Relaciones de herencia...

Otras son muy específicas:

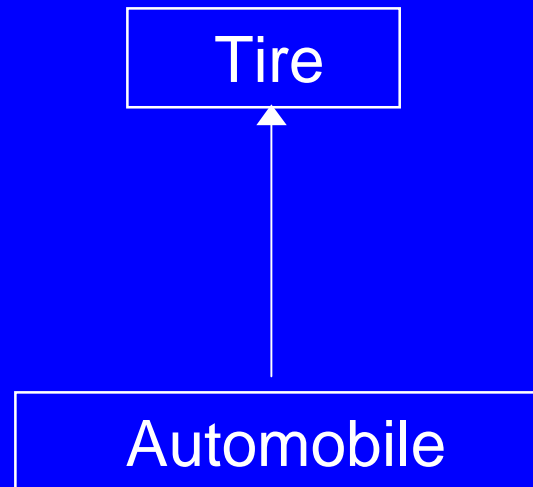


Múltiples Niveles son posible



Otras Relaciones

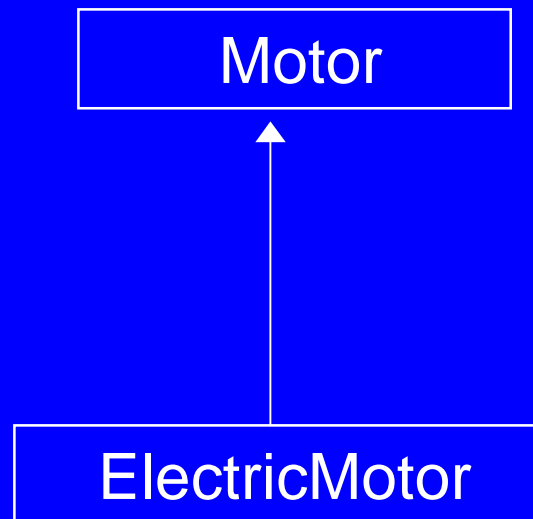
Hay relaciones que **no son** basadas en herencia. Cómo las llamaríamos?



Esta es una relación *tiene-un*, también llamada **agregación**.

Motor y ElectricMotor

Consideremos dos clases que tienen algo en común.



Motor y ElectricMotor

Un objeto ElectricMotor contiene el mismo número de identificación (ID) como un Motor, más el voltaje.

Motor

33333

33333

220.0

ElectricMotor

Clase CMotor

Definición de la clase CMotor:

```
class CMotor {
public:
    CMotor() { }
    CMotor( const string & id );

    string get_ID() const;
    void set_ID(const string & s);

    void Display() const;
    void Input();

private:
    string m_sID;
}; // mas...
```

```
CMotor::CMotor( const string & id )
{ set_ID(id); }

string CMotor::get_ID() const
{ return m_sID; }

void CMotor::set_ID(const string & s)
{ m_sID = s; }

void CMotor::Display() const
{ cout << "[CMotor] ID=" << m_sID; }

void CMotor::Input()
{
    string temp;
    cout << "Enter Motor ID: ";
    cin >> temp;
    set_ID(temp);
}
```

Creación de Clases Derivadas

La clase base debe aparecer primero en las declaraciones. Cuando la clase derivada es declarada, ésta menciona a la clase base.

```
class base {  
    ...  
};  
  
class derived : public base {  
    ...  
};
```

Cualquier número de clases pueden ser derivadas de una misma clase base. Una clase puede derivarse de más de una clase base (ojo no permitido en JAVA)

Class CElectricMotor

```
class CElectricMotor : public CMotor {
public:
    CElectricMotor();
    CElectricMotor(const string & id,
                  double volts);


    void Display() const;
    void Input();
    void set_Voltage(double volts);
    double get_Voltage() const;

private:
    double m_nVoltage;
};
```

Inicializador de Clase Base

Un **inicializador de clase base** llama al constructor de la clase base. En este ejemplo, el número ID del motor es pasado al constructor de CMotor.

```
CElectricMotor::CElectricMotor(const string & id,  
                                double volts) : CMotor(id)  
{  
    m_nVoltage = volts;  
}
```



Llamando a métodos de la clase base

La función **Input** existe en ambas clases CMotor y CElectricMotor. En lugar de duplicar el código ya escrito, se llama al método correspondiente en la clase base:

```
void CElectricMotor::Input()
{
    CMotor::Input(); // call base class first

    double volts;
    cout << "Voltage: ";
    cin >> volts;
    set_Voltage(volts);
}
```

Llamado de Métodos de la clase Base

Esta es la función Input en la clase CMotor:

```
void CMotor::Input()  
{  
    string temp;  
    cout << "Enter Motor ID: ";  
    cin >> temp;  
    set_ID(temp);  
}
```

Función o método de Despliegue

La función Display funciona de la misma manera. Ésta llama a CMotor::Display primero.

```
void CElectricMotor::Display() const
{
    // call base class function first
    CMotor::Display();

    cout << " [CElectricMotor]"
         << " Voltage=" << m_nVoltage << endl;
}
```


Probando las Clases

Un programa de prueba puede crear instancias de ambas clases CMotor and CElectricMotor.

```
CMotor mot("33333");  
mot.Display();  
cout << endl;
```

```
CElectricMotor elec("40000",220.0);  
elec.Display();  
cout << endl;
```

Probando Clases

Cuando usamos instancias de clases derivadas de otra, podemos llamar a funciones de la clase base y la derivada.

```
CElectricMotor elec;           // CElectricMotor  
  
elec.set_ID("40000");          // CMotor  
  
elec.set_Voltage(220.0);      // CElectricMotor
```

Probando Clases

Cuando el mismo nombre de función existe en ambas clases, C++ automáticamente llama la función implementada para la clase derivada. Éste es el caso con las funciones Input y Display:

```
elec.Input();           // CElectricMotor  
elec.Display();        // CElectricMotor
```

Asignación de Objetos

Podemos asignar objetos de clases derivadas a un objeto de la clase base. Esto es llamado *rebanado de objeto (object slicing)*.

```
CMotor mot;
CElectricMotor elec;

mot = elec;          // sliced down to a motor
elec.get_Voltage(); // ok
mot.get_Voltage();  // error
```

Asignación de Objetos

Pero no podemos asignar una instancia de una clase base a una instancia de una clase derivada. Ello permitiría referencias a miembros no existentes.

```
CMotor mot;  
CElectricMotor elec;  
  
elec = mot;           // error  
  
elec.set_Voltage( 220 ); // ???
```

Acceso a miembros Protected (Protegidos)

Miembros de una clase designados como **protected** son visibles a ambos la clase actual y las clases derivadas. (*y a clases amigas -friend- pero a nadie más*)

Clases con miembros Protected

Aquí hay un ejemplo que usa el calificador **protected** para limitar la visibilidad de `get_ID` y `set_ID`:

```
class CMotor {  
public:  
    CMotor() { }  
    CMotor( const string & id );  
  
protected:  
    string get_ID() const;  
    void set_ID(const string & s);  
  
    //...
```

Acceso a miembros Protected

El programa principal no puede llamar set_ID y get_ID porque ellos son protegidos:

```
CMotor M;  
  
M.set_ID("12345");           // error  
  
M.get_ID();                   // error
```


Acceso a miembros Protected

Pero funciones en CElectricMotor sí pueden acceder set_ID:

```
CElectricMotor::CElectricMotor(  
    const string & id, double volts)  
{  
    m_nVoltage = volts;  
    set_ID(id);  
}
```

Herencia Protegida

Supongamos por un momento que Cmotor usa miembros públicos para todas sus funciones:

```
class CMotor {  
public:  
    CMotor() { }  
    CMotor( const string & id );  
  
    string get_ID() const;  
    void set_ID(const string & s);  
  
    //...
```

Herencia Protegida

Podemos usar el calificador **protected** cuando creamos una clase derivada.

Todas las funciones públicas en la clase base pasan a ser protegidas en la clase derivada.

```
class CElectricMotor : protected CMotor {  
  
    //...  
  
};
```

Herencia Protegida

Por ejemplo, el programa principal no puede llamar `set_ID` y `get_ID` en un motor eléctrico porque las funciones no son públicas en esta clase:

```
CElectricMotor EM;  
  
EM.set_ID("12345");           // error  
  
EM.get_ID();                 // error
```

Herencia Protegida

Puede ser que el autor de la clase CElectricMotor no quiera dar a conocer el número ID del motor.

Funciones en CElectricMotor sí pueden acceder funciones públicas en CMotor. Un ejemplo:

```
CElectricMotor::CElectricMotor(  
    const string & id, double volts)  
{  
    m_nVoltage = volts;  
    set_ID(id);  
}
```

Herencia Privada

La herencia privada causa que todas las funciones declaradas en la clase base sean privadas en la clase derivada.

Pareciera que no hay diferencia con herencia protegida: Funciones en CElectricMotor pueden acceder funciones miembros en CMotor...

```
class CElectricMotor : private CMotor {  
  
    //...  
  
};
```

Herencia Privada

Pero cuando derivamos una nueva clase (CPumpMotor) de CElectricMotor, la diferencia se hace notar: funciones en CPumpMotor no pueden acceder miembros públicos de CMotor.

```
class CPumpMotor : public CElectricMotor {
public:
    void Display() {
        CMotor::Display();           // not accessible!
        CElectricMotor::Display();   // this is OK
    }
};
```

The image features a solid blue background. In the bottom right corner, there is a teal-colored shape that tapers from the bottom edge towards the center. The word "Fin" is written in a bold, yellow, sans-serif font, positioned in the upper-middle part of the image.

Fin