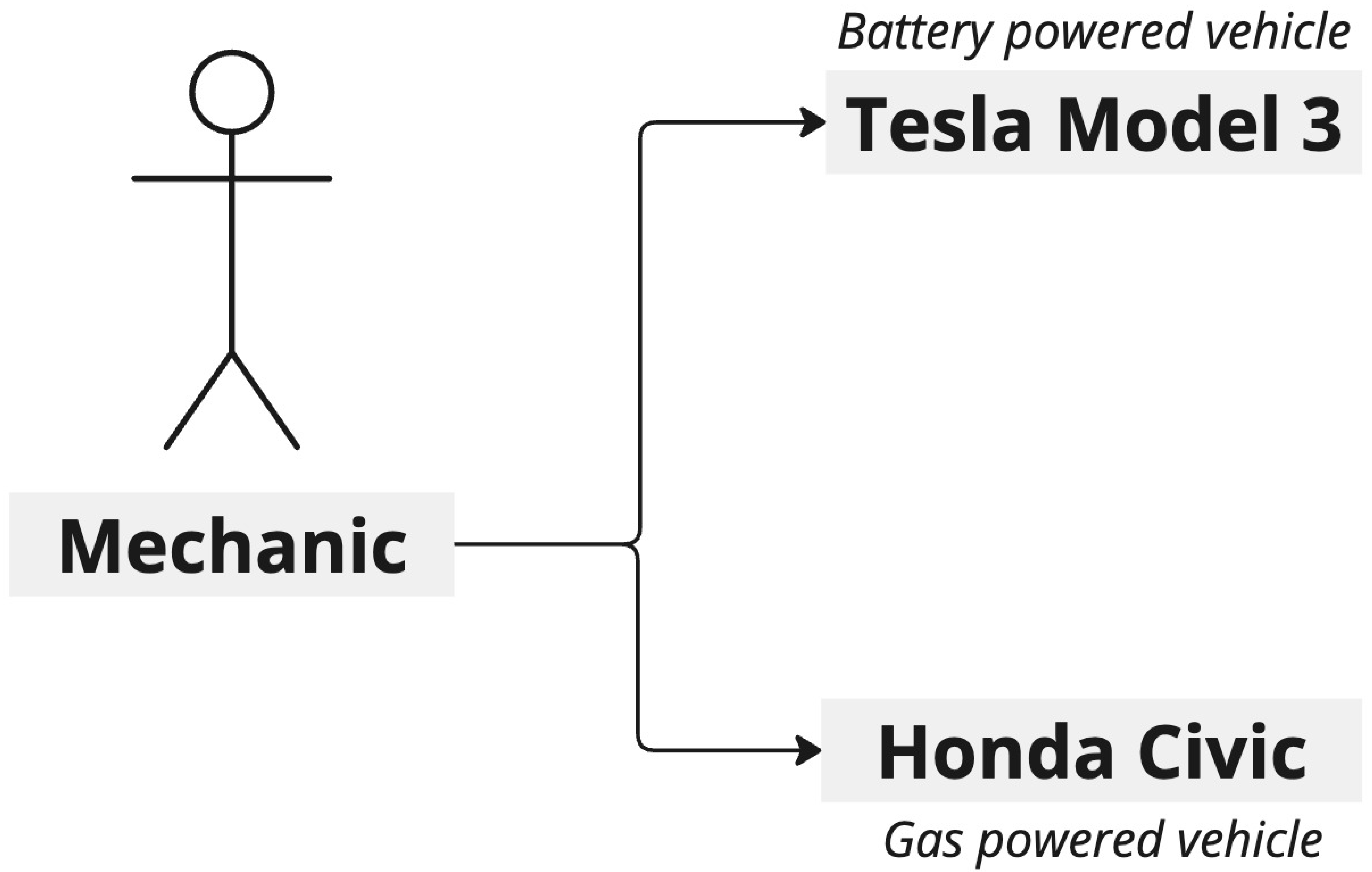
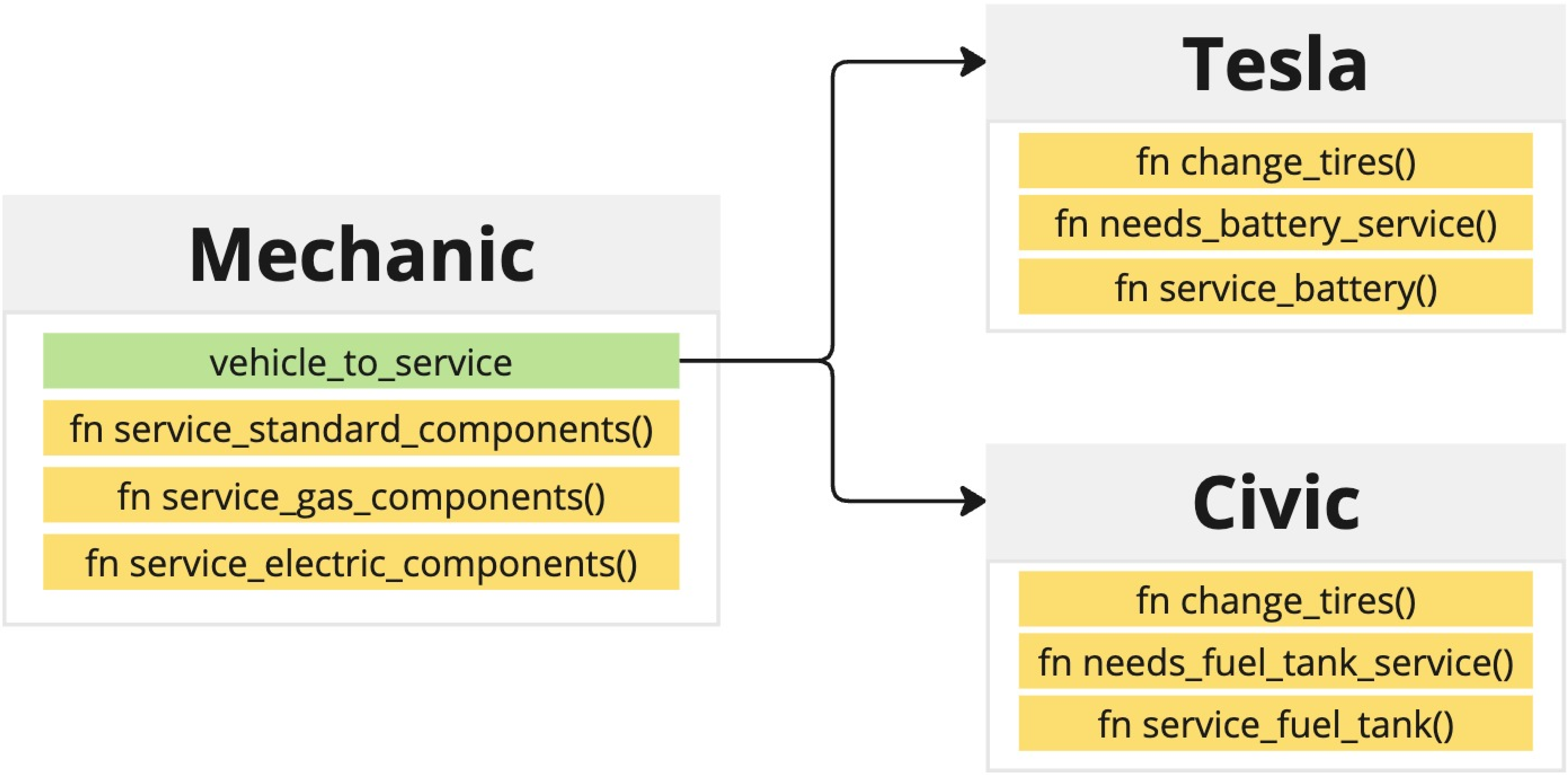
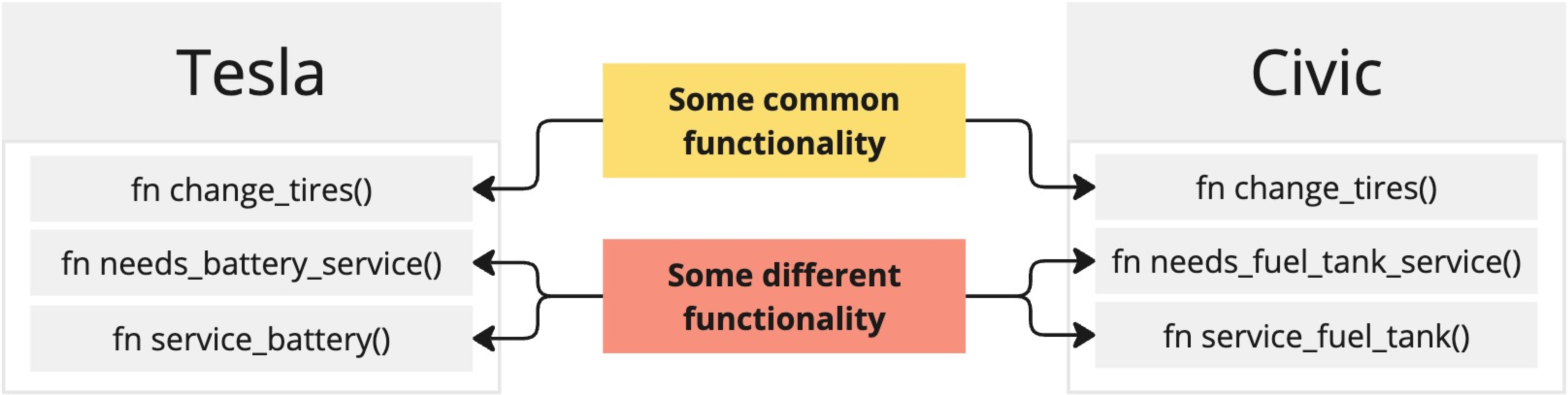

Battery powered vehicle
Tesla Model 3
Mechanic
Honda Civic
Gas powered vehicle


Mechanic
vehicle_to_service
fn service_standard_components()
fn service_gas_components()
fn service_electric_components()
Tesla
fn change_tires()
fn needs_battery_service()
fn service_battery()
Civic
fn change_tires()
fn needs_fuel_tank_service()
fn service_fuel_tank()



Tesla
fn change_tires()
fn needs_battery_service()
fn service_battery()
Some common functionality
Some different functionality
Civic
fn change_tires()
fn needs_fuel_tank_service()
fn service_fuel_tank()

Option #1

Use an enum to represent the different kinds of cars

Downside:

Every function has to see if the operation is appropriate for the car variant

```rust
pub enum Vehicle {
    Tesla,
    Civic,
}

impl Vehicle {
    fn change_tires(&self) {
        match self {
            Vehicle::Tesla => println!("Replacing tesla tires"),
            Vehicle::Civic => println!("Replacing civic tires"),
        }
    }

    fn needs_battery_service(&self) -> Result<bool, String> {
        match self {
            Vehicle::Tesla => Ok(true),
            Vehicle::Civic => {
                Err(String::from("cant service the battery of a civic!"))
            },
        }
    }
}
```

```rust
fn service_car(car: ?????) {
    car.change_tires();
}

fn main() {
    let tesla = Tesla{};
    let civic = Civic{};

    service_car(tesla);
    service_car(civic);
}
```

Goal

Be able to call this 'service_car' function with either a Tesla or a Civic

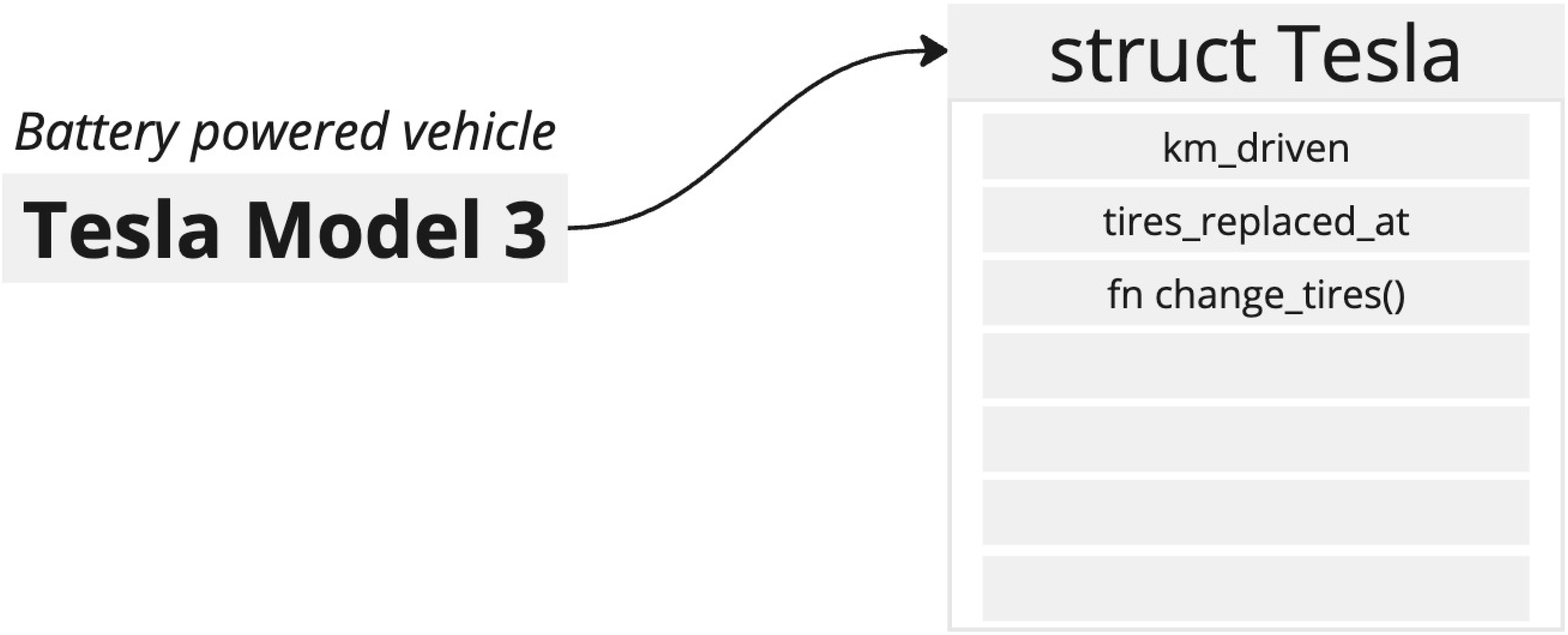
Battery powered vehicle

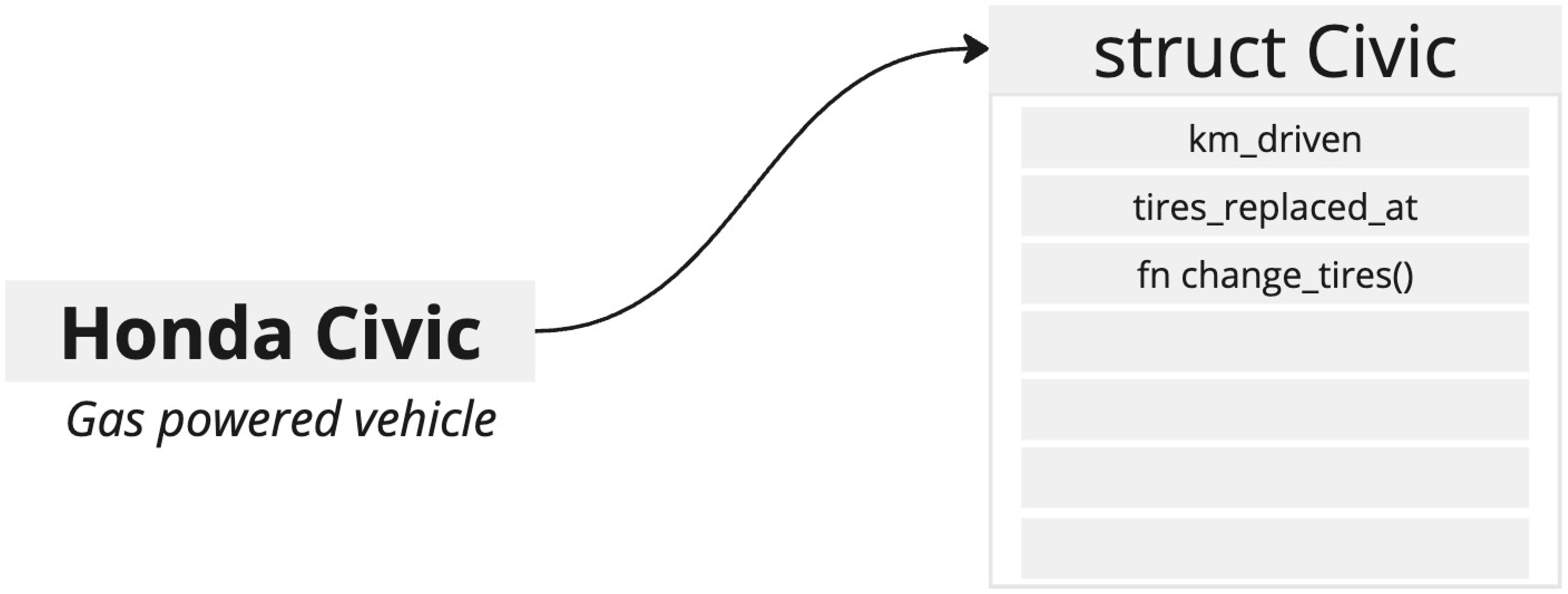
Tesla Model 3

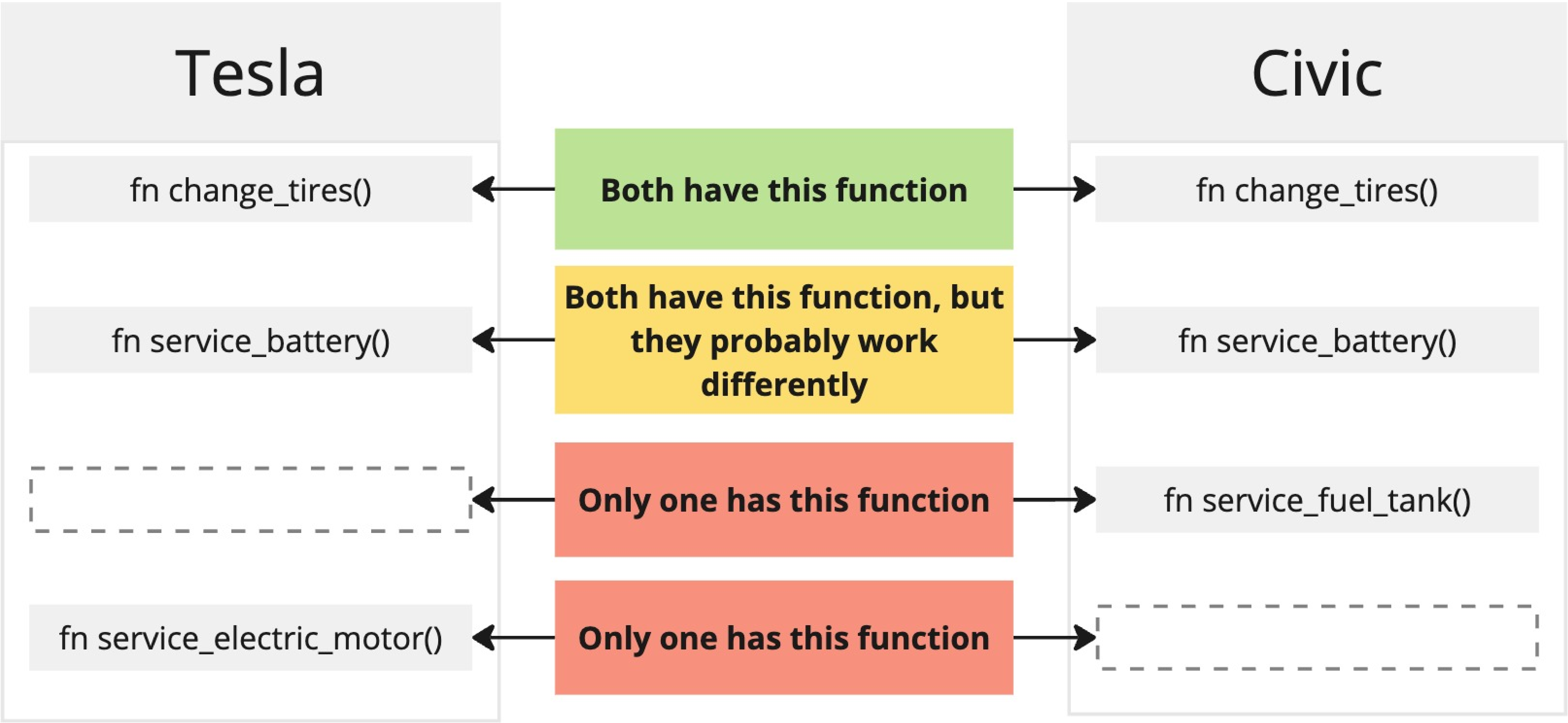
Goal: Model maintenance for each vehicle

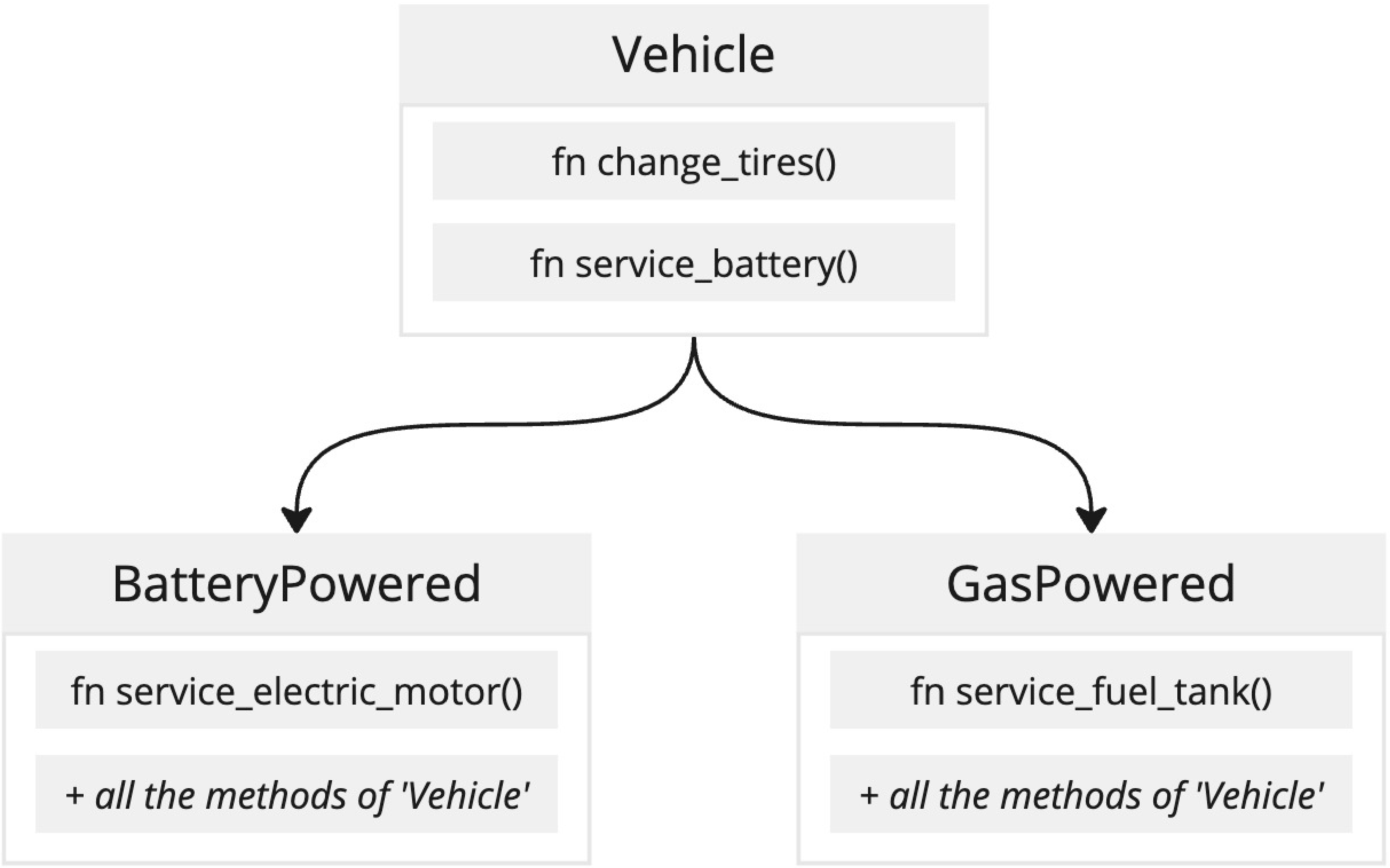
Honda Civic

Gas powered vehicle


Battery powered vehicle
Tesla Model 3
struct Tesla
km_driven
tires_replaced_at
fn change_tires()


Honda Civic
Gas powered vehicle
struct Civic
km_driven
tires_replaced_at
fn change_tires()


Tesla
Civic
fn change_tires()
Both have this function
fn change_tires()
fn service_battery()
Both have this function, but they probably work differently
fn service_battery()
Only one has this function
fn service_fuel_tank()
fn service_electric_motor()
Only one has this function



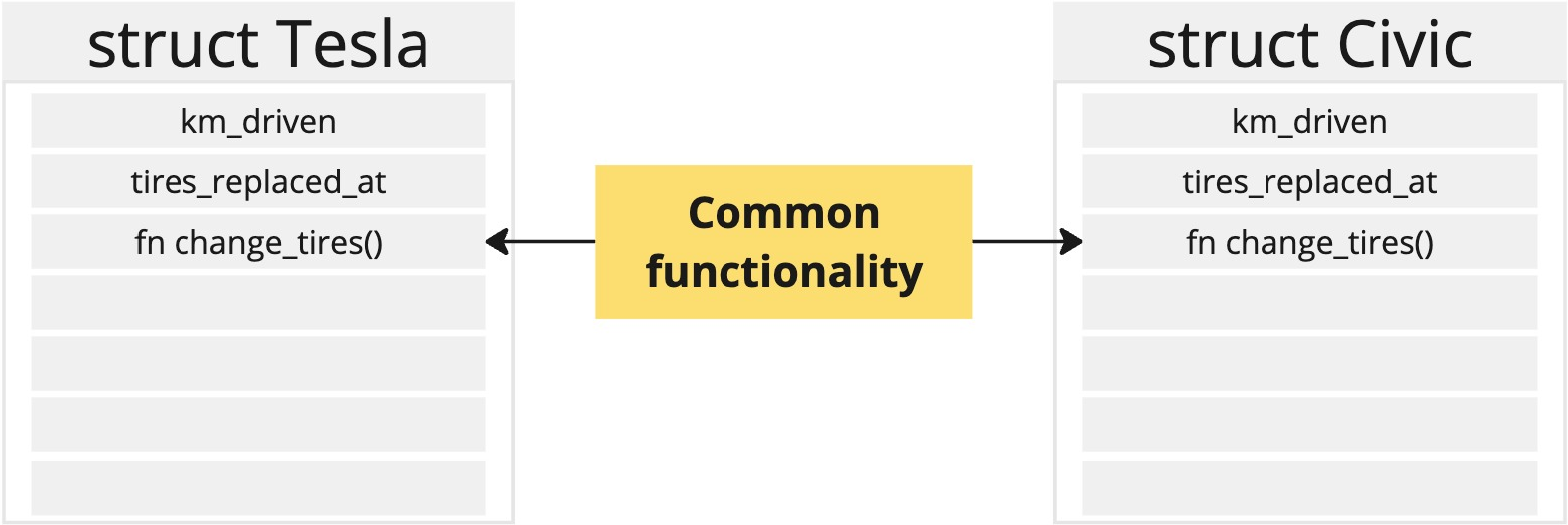
Vehicle
fn change_tires()
fn service_battery()
BatteryPowered
fn service_electric_motor()
+ all the methods of 'Vehicle'
GasPowered
fn service_fuel_tank()
+ all the methods of 'Vehicle'

Traits define common functionality between different types

1	Find one or more types that need to have common functionality
2	Identify which methods those types will have in common
3	Define the trait
4	Implement the trait for each type in step #1
5	Use the trait as bounds for a generic function/struct/enum/vector/etc

1 Find one or more types that need to have common functionality

2 Identify which methods those types will have in common



3

Define the trait

```
trait Vehicle {
    pub fn change_tires(&mut self) {
        /* code to change tires */
    }
}
```

4 Implement the trait for each type in step #1

```rust
trait Vehicle {
    pub fn change_tires(&mut self) {
        /* code to change tires */
    }
}
```

```rust
struct Tesla {}

impl Vehicle for Tesla {

}
```

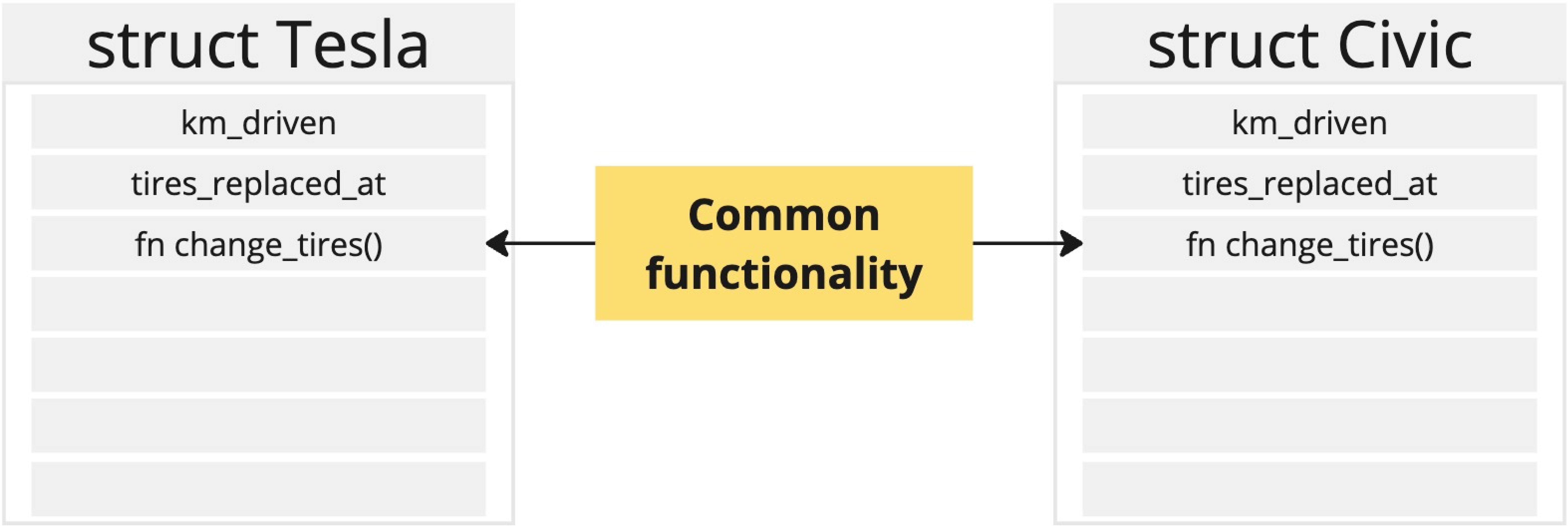
```rust
struct Civic {}

impl Vehicle for Civic {

}
```

5 Use the trait as bounds for a generic function/struct/enum/vector/etc

```rust
fn service_car(car: impl Vehicle) {
    car.change_tires();
}

fn main() {
    let tesla = Tesla{};
    let civic = Civic{};

    service_car(tesla);
    service_car(civic);
}
```

'service_car' expects to be called with any value that implements the 'Vehicle' trait

Both Tesla and Civic implement the 'Vehicle' trait, so they can be passed into 'service_car'


struct Tesla
km_driven
tires_replaced_at
fn change_tires()
Common functionality
struct Civic
km_driven
tires_replaced_at
fn change_tires()

```rust
trait Vehicle {
    fn change_tires(&mut self) {
        println!("Changing tires!");
    }
}
```

A trait is a set of methods

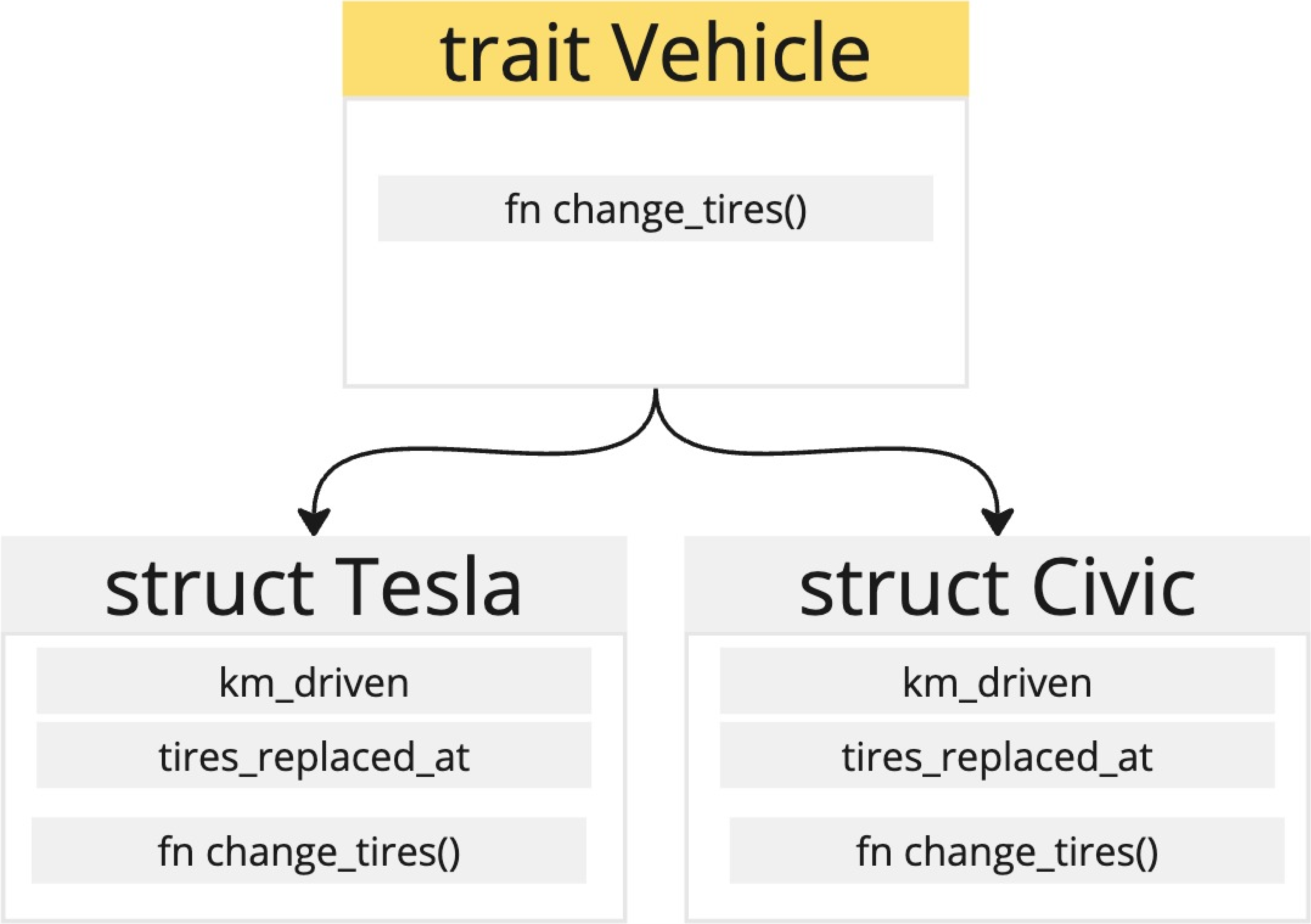
```rust
impl Vehicle for Tesla { }
impl Vehicle for Civic { }
```

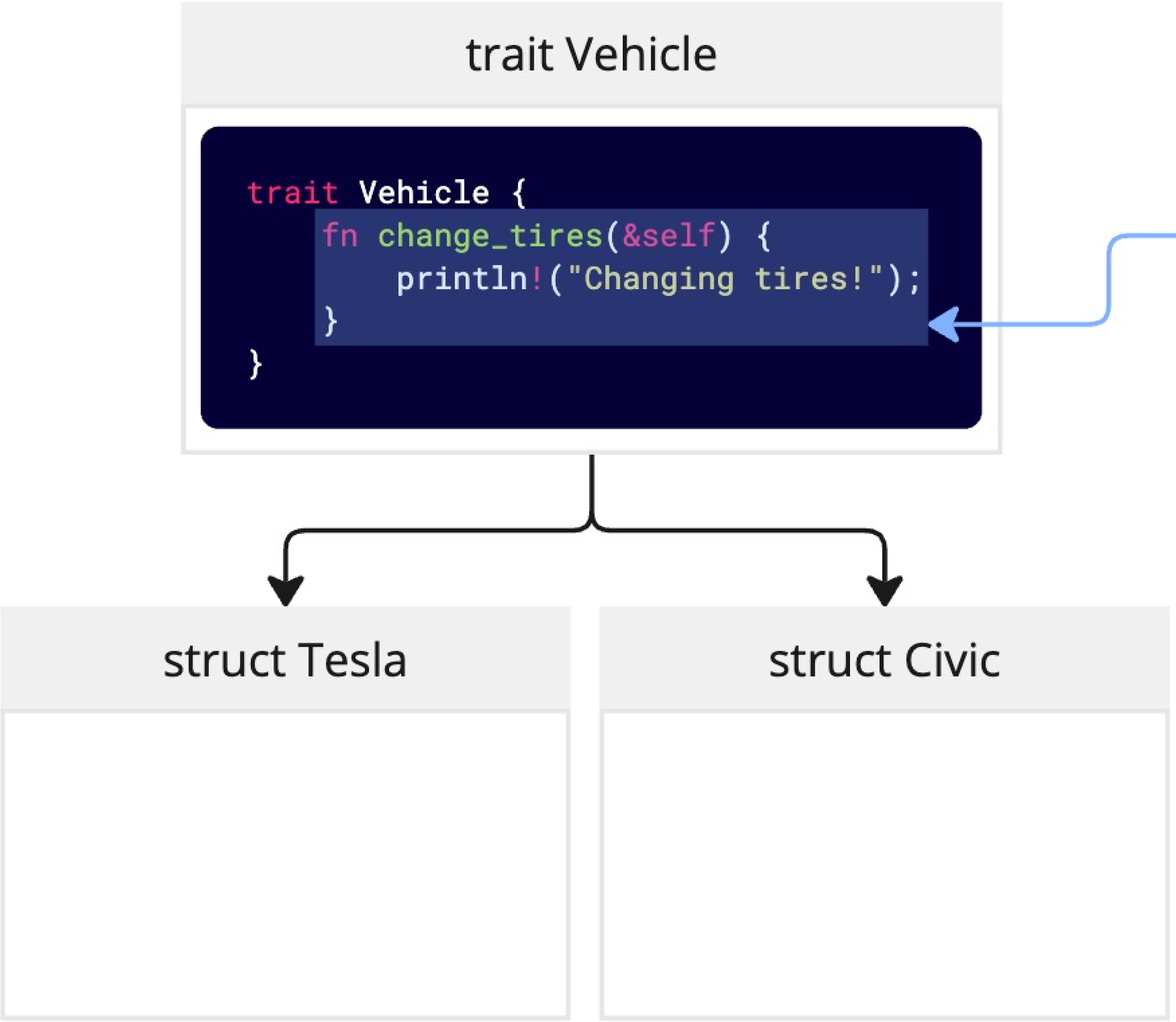
A struct/enum/anything can choose to "implement" that trait.

This struct/enum/anything is called the 'implementor'

```rust
fn main() {
    let car1 = Civic::new(99999, 0);
    let car2 = Tesla::new(0, 0);

    car1.change_tires();
    car2.change_tires();
}
```

The implementor gets access to the methods defined in the trait


trait Vehicle
fn change_tires()
struct Tesla
km_driven
tires_replaced_at
fn change_tires()
struct Civic
km_driven
tires_replaced_at
fn change_tires()

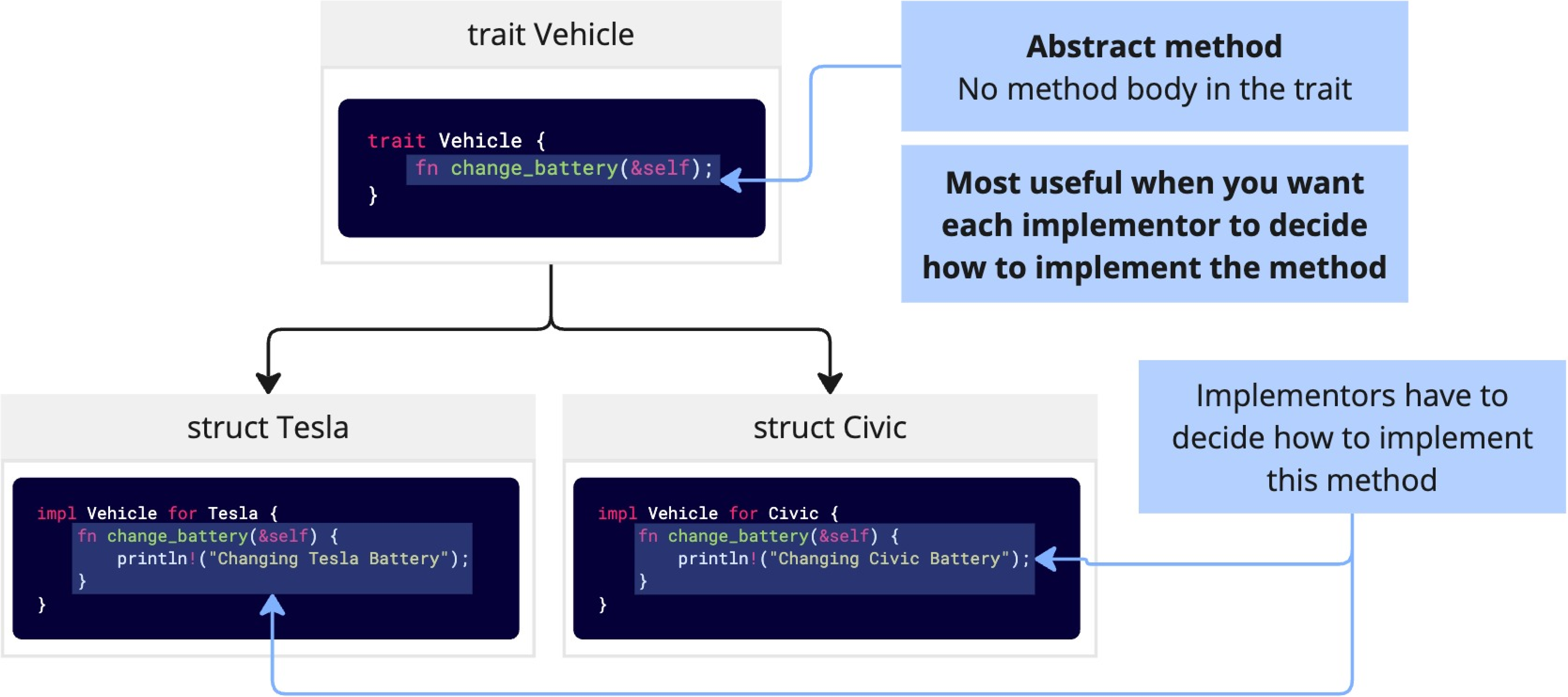


Concrete methods

Method body defined in the trait

Also known as 'default methods'

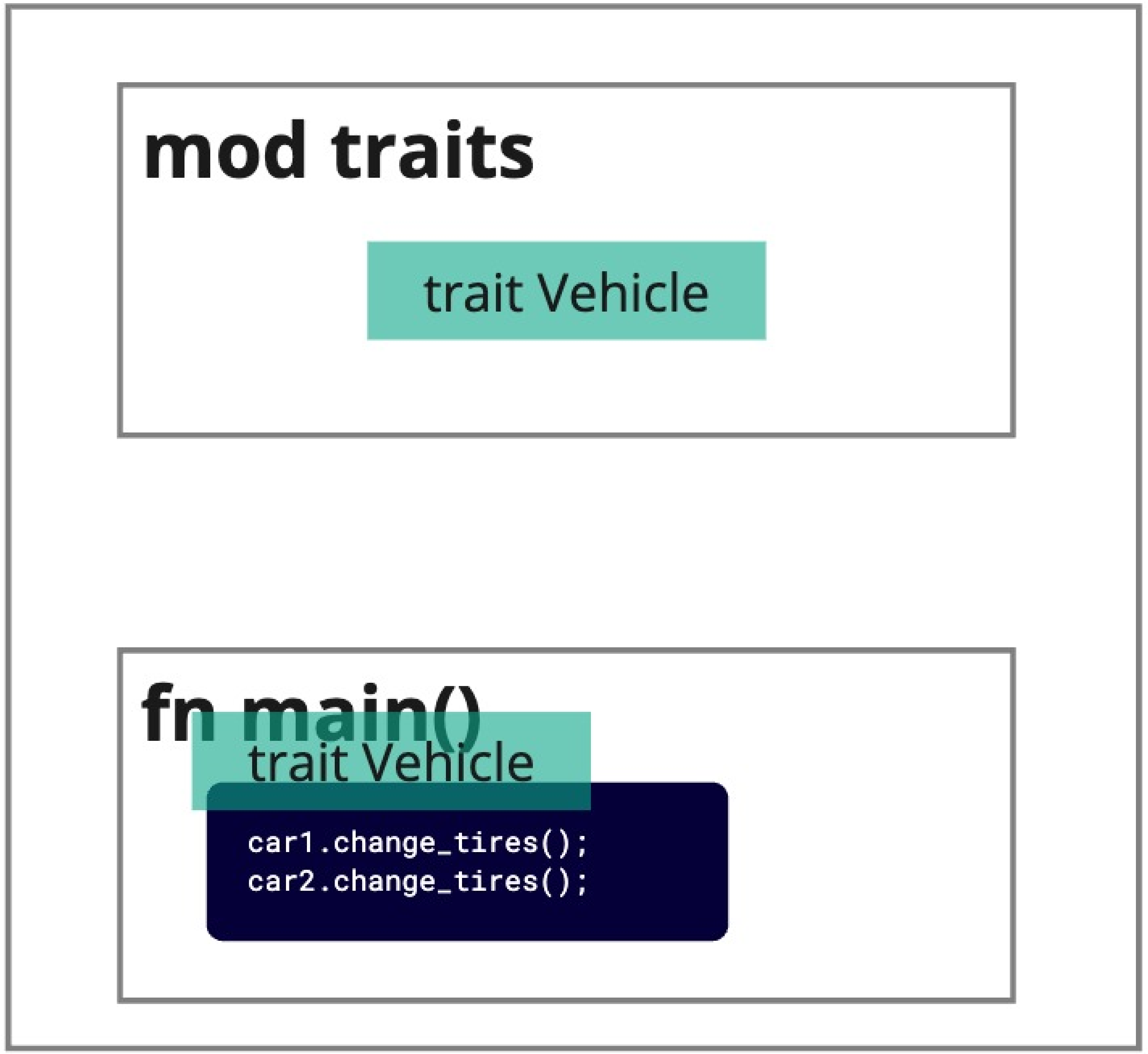
Most useful when all implementors should have the same exact method


trait Vehicle
trait Vehicle {
fn change_battery(&self);
}
Abstract method
No method body in the trait
Most useful when you want each implementor to decide how to implement the method
struct Tesla
impl Vehicle for Tesla {
fn change_battery(&self) {
println!("Changing Tesla Battery");
}
}
struct Civic
impl Vehicle for Civic {
fn change_battery(&self) {
println!("Changing Civic Battery");
}
}
Implementors have to decide how to implement this method

Big Gotcha #1

The trait has to be in scope when a method from it is called

```rust
mod traits {
    pub trait Vehicle {
        fn change_tires(&mut self) {
            /* ... */
        }
    }
}

impl traits::Vehicle for Tesla { };
impl traits::Vehicle for Civic { };

fn main() {
    let mut car1 = Tesla::new();
    let mut car2 = Civic::new();

    car1.change_tires();
    car2.change_tires();
}
```



```rust
trait Vehicle {
    fn change_tires(&mut self) {
        // Bad! Traits can't assume the
            implementor has a particular field
        if self.km_driven > 0 {

        }
    }
}
```

Big Gotcha #2

A trait can't refer to fields, only methods

```rust
trait Vehicle {
    fn change_tires(&mut self) {
        if self.get_km_driven() - self.tires_replaced_at() > 100 {
            self.set_tires_replaced_at(self.get_km_driven());
            println!("Tires Replaced!")
        }
    }

    // Implementor has to implement these methods:
    fn get_km_driven() -> u32;
    fn tires_replaced_at() -> u32;
    fn set_tires_replaced_at(km: u32);
}


impl Vehicle for Tesla {
    fn get_km_driven() -> u32 {
        self.km_driven
    }
    fn tires_replaced_at() -> u32 {
        self.tires_replaced_at
    }
    fn set_tires_replaced_at(km: u32) {
        self.tires_replaced_at = km;
    }
}
```

Big Gotcha #2
A trait can't refer to fields, only methods

Workaround:
Add methods that give access to the fields you need and require the implementor to implement them

There is a downside to this approach

The names of these methods strongly imply that an implementor will have 'km_driven' and 'tires_replaced_at' fields

```rust
trait Vehicle {
    fn change_tires(&mut self) {
        if self.get_km_driven() - self.tires_replaced_at() > 100 {
            self.set_tires_replaced_at(self.get_km_driven());
            println!("Tires Replaced!")
        }
    }

    // Implementor has to implement these methods:
    fn get_km_driven() -> u32;
    fn tires_replaced_at() -> u32;
    fn set_tires_replaced_at(km: u32);
}


impl Vehicle for Tesla {
    fn get_km_driven() -> u32 {
        self.km_driven
    }
    fn tires_replaced_at() -> u32 {
        self.tires_replaced_at
    }
    fn set_tires_replaced_at(km: u32) {
        self.tires_replaced_at = km;
    }
}
```

Default impl of 'change_tires' assumes we want to change tires when a certain km has been driven

```rust
trait Vehicle {
    fn change_tires(&mut self) {
        if self.get_km_driven() - self.tires_replaced_at() > 100 {
            self.set_tires_replaced_at(self.get_km_driven());
            println!("Tires Replaced!")
        }
    }

    // Implementor has to implement these methods:
    fn get_km_driven() -> u32;
    fn tires_replaced_at() -> u32;
    fn set_tires_replaced_at(km: u32);
}


impl Vehicle for Tesla {
    fn get_km_driven() -> u32 {
        self.km_driven
    }
    fn tires_replaced_at() -> u32 {
        self.tires_replaced_at
    }
    fn set_tires_replaced_at(km: u32) {
        self.tires_replaced_at = km;
    }
}
```

We're working on a trait called 'Vehicle'

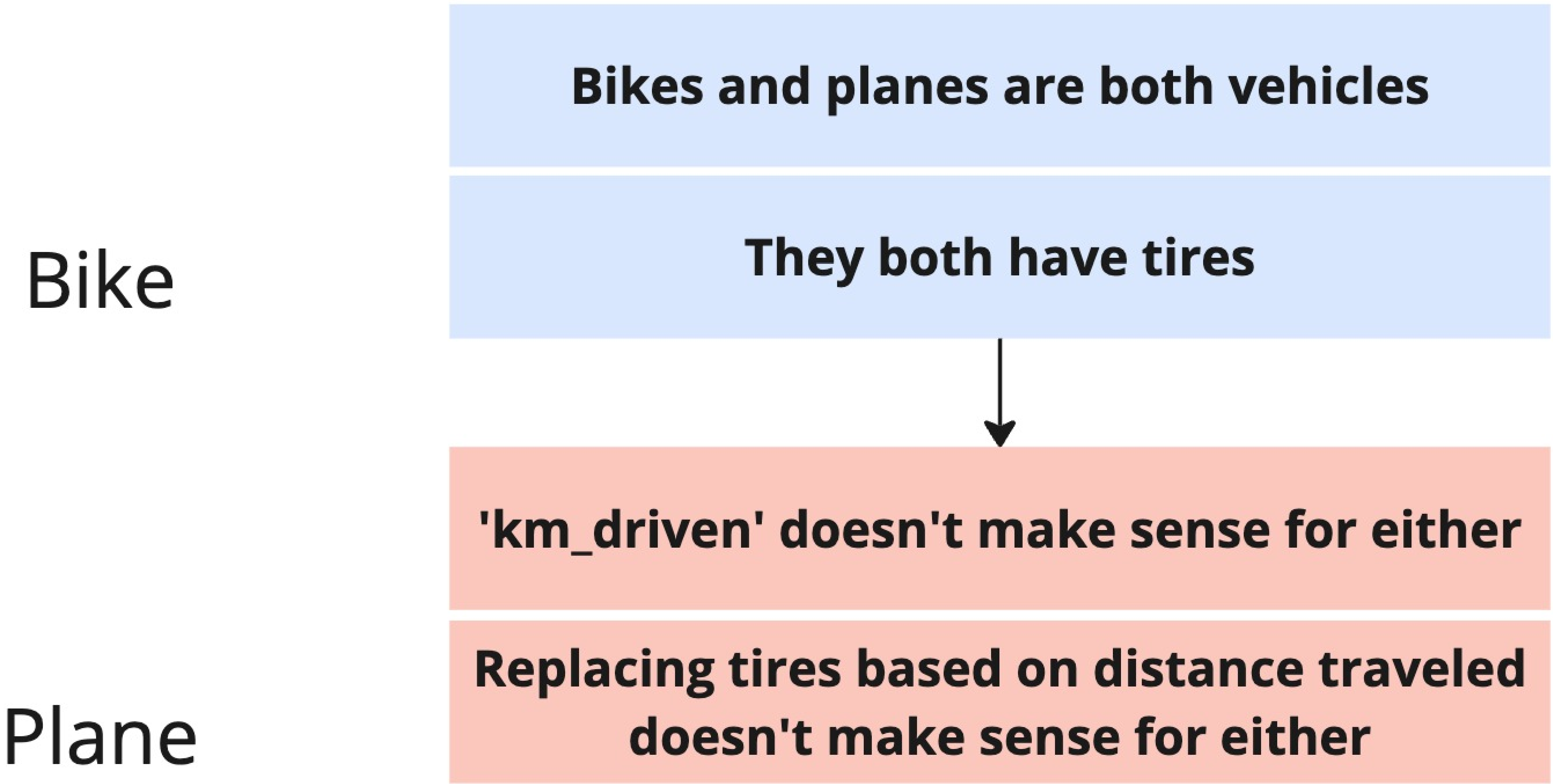
It assumes that all vehicles have a 'km_driven', a 'tires_replaced_at', and that we want to replace tires based on how long its been since the tires have been replaced

There are kinds of vehicles that *don't follow these rules at all*

Default impl of 'change_tires' assumes we want to change tires when a certain km has been driven

The names of these methods strongly imply that an implementor will have 'km_driven' and 'tires_replaced_at' fields

```
trait Vehicle {
    fn change_tires(&mut self) {
        if self.get_km_driven() - self.tires_replaced_at() > 100 {
            self.set_tires_replaced_at(self.get_km_driven());
            println!("Tires Replaced!")
        }
    }

    // Implementor has to implement these methods:
    fn get_km_driven() -> u32;
    fn tires_replaced_at() -> u32;
    fn set_tires_replaced_at(km: u32);
}


impl Vehicle for Tesla {
    fn get_km_driven() -> u32 {
        self.km_driven
    }
    fn tires_replaced_at() -> u32 {
        self.tires_replaced_at
    }
    fn set_tires_replaced_at(km: u32) {
        self.tires_replaced_at = km;
    }
}
```


Bike
Plane
Bikes and planes are both vehicles
They both have tires
'km_driven' doesn't make sense for either
Replacing tires based on distance traveled doesn't make sense for either

Traits should have methods that are flexible and leave the details to the implementor

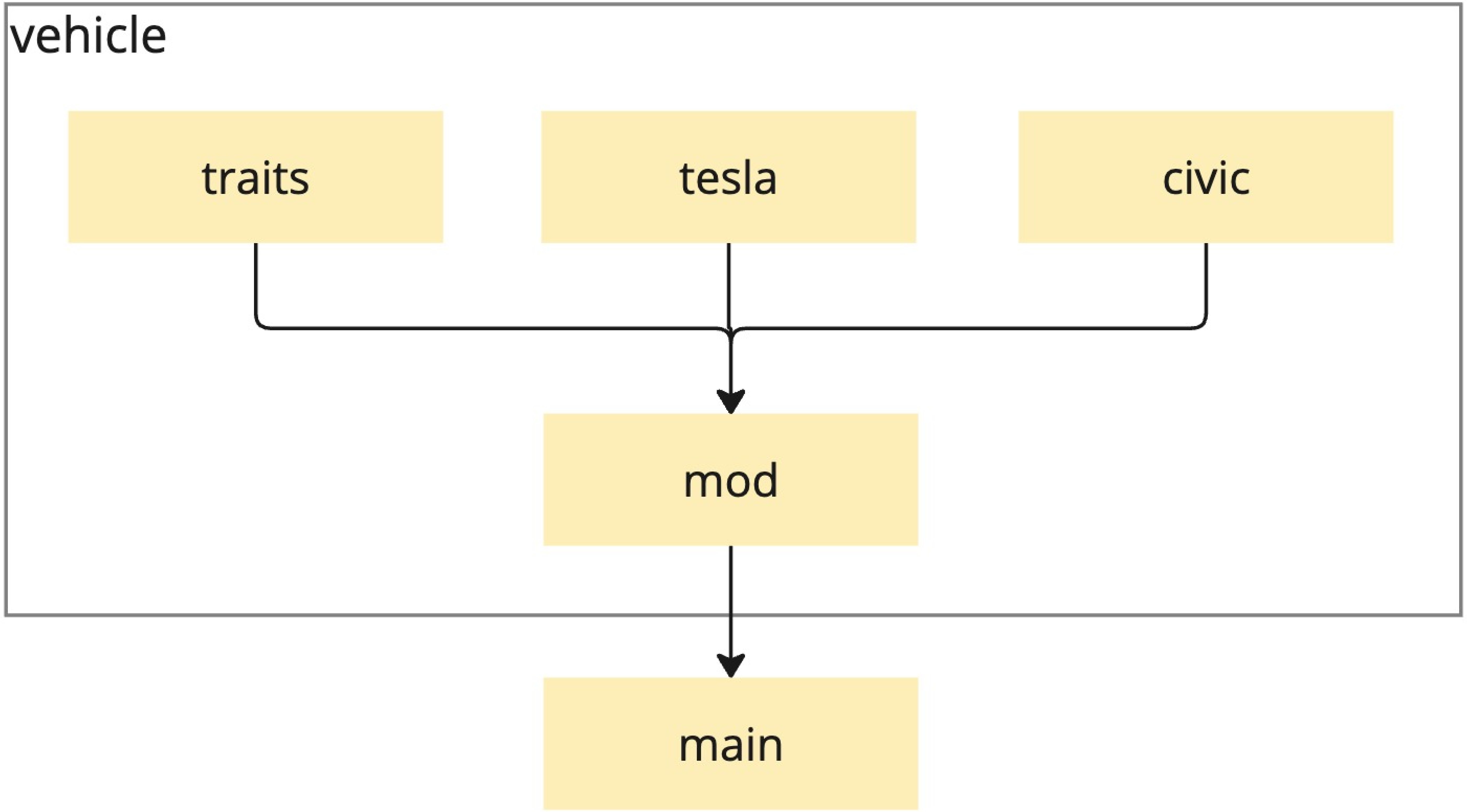
```rust
trait Vehicle {
    fn change_tires(&mut self) {
        if self.tire_change_required() {
            self.record_tire_change();
            println!("Replacing tire...");
        }
    }

    // Implementor has to implement these methods:
    fn tire_change_required(&self) -> bool;
    fn record_tire_change(&mut self);
}
```

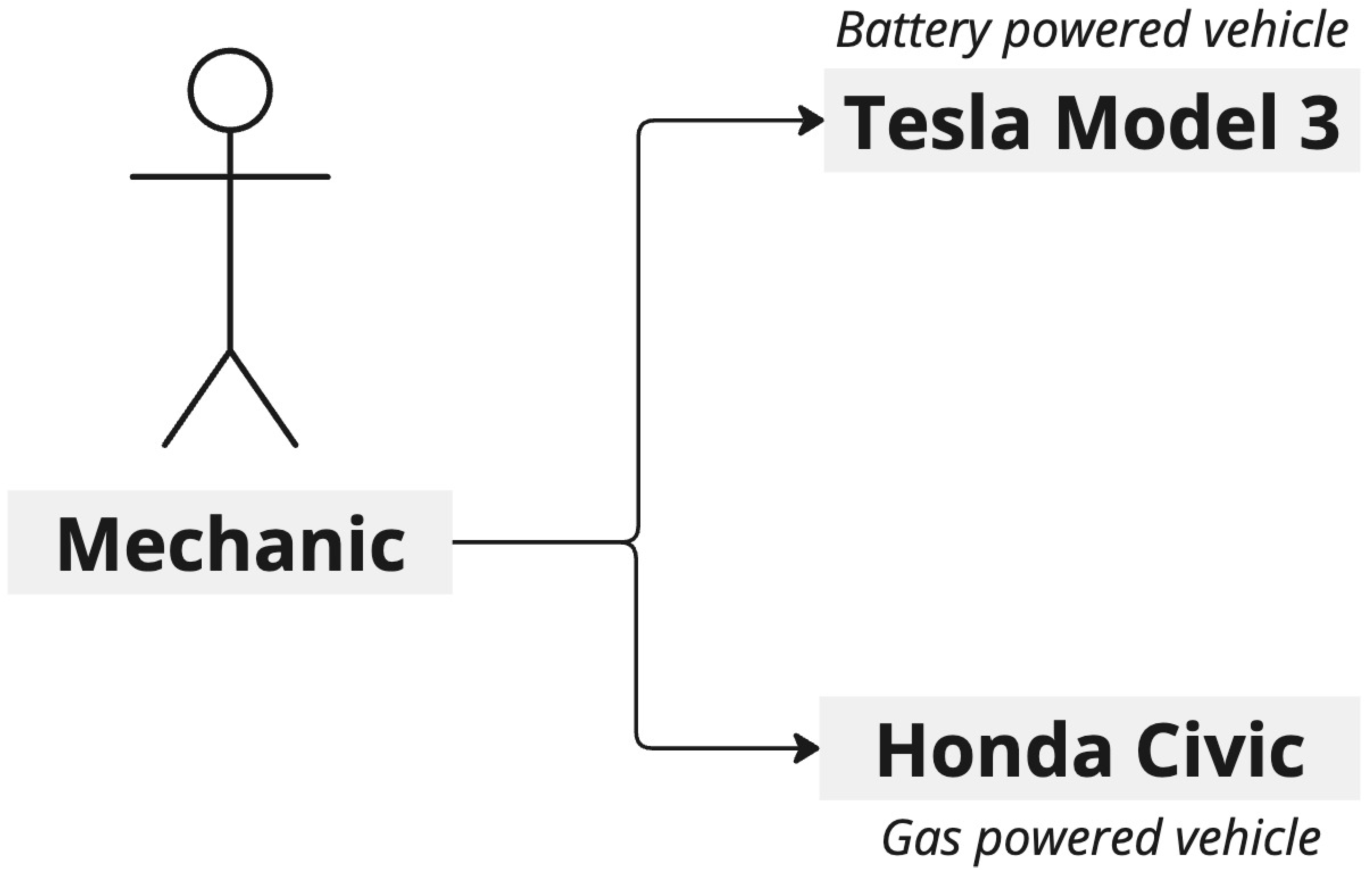
Vehicle Trait

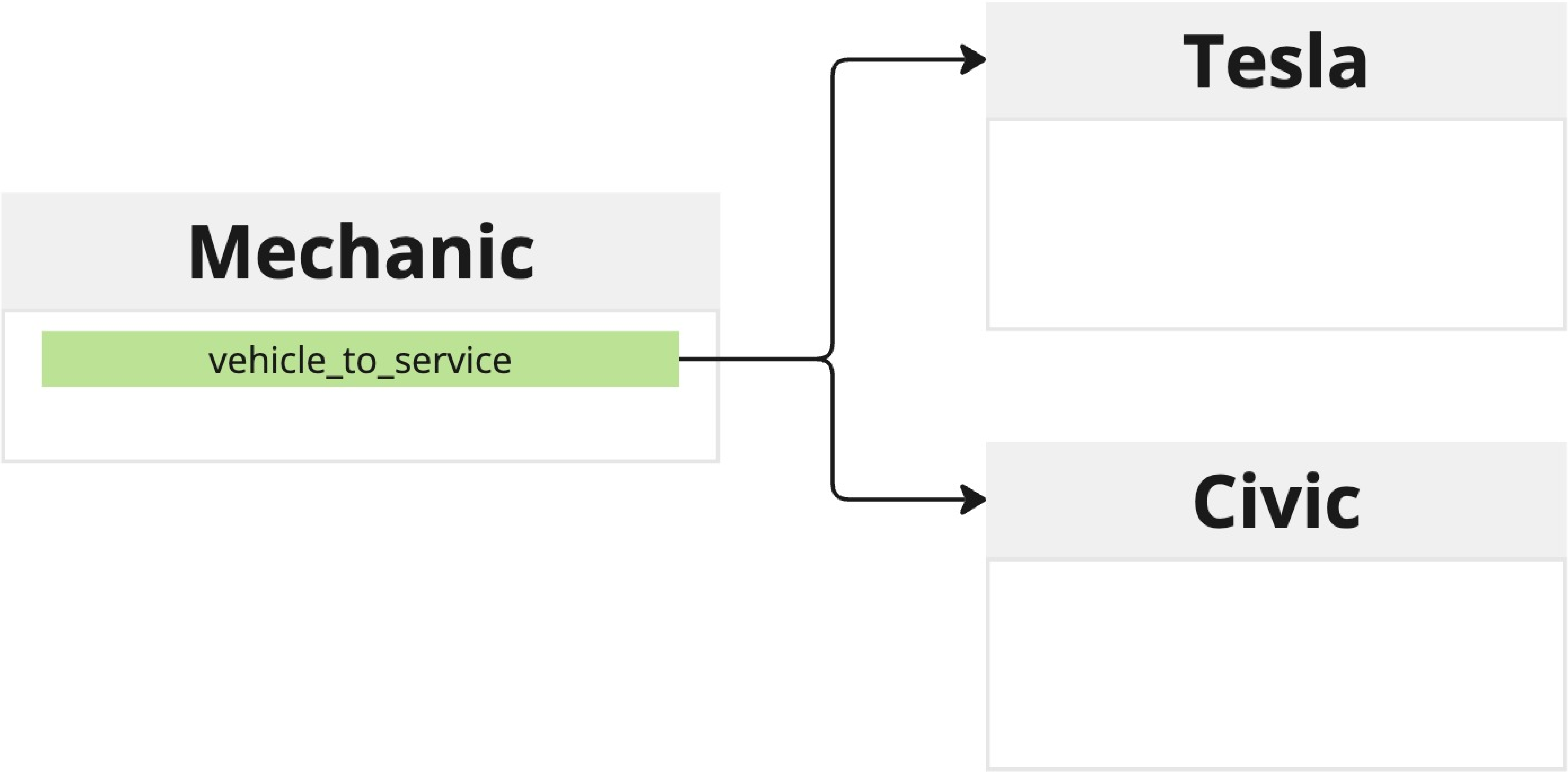
```rust
impl Vehicle for Tesla {
    fn tire_change_required(&self) -> bool {
        self.km_driven - self.tires_replaced_at > 10000
    }
    fn record_tire_change(&mut self) {
        self.tires_replaced_at = self.km_driven;
    }
}
```

```rust
impl Vehicle for Plane {
    fn tire_change_required(&self) -> bool {
        self.landing_on_current_tires > 10
    }
    fn record_tire_change(&mut self) {
        self.landing_on_current_tires = 0;
    }
}
```

```rust
impl Vehicle for Bike {
    fn tire_change_required(&self) -> bool {
        self.tire_is_flat
    }
    fn record_tire_change(&mut self) {
        self.tire_is_flat = false;
    }
}
```


vehicle
traits
tesla
civic
mod
main


Battery powered vehicle
Tesla Model 3
Mechanic
Honda Civic
Gas powered vehicle


Tesla
Mechanic
vehicle_to_service
Civic

Tesla

fn service_electric_motor()

Civic

fn service_fuel_tank()

Electric Vehicles

Tesla	Rivian R1S	Hyundai Ioniq
fn service_electric_motor()	fn service_electric_motor()	fn service_electric_motor()

Civic	Hyndai Elantra	BMW M3
fn service_fuel_tank()	fn service_fuel_tank()	fn service_fuel_tank()

Gas Vehicles

```rust
pub trait Vehicle {
    fn change_tires(&mut self) {
        /* code */
    }

    fn tire_change_required(&self) -> bool;
    fn record_tire_change(&mut self);
}

pub trait BatteryPowered: Vehicle {
    fn service_electric_motor(&self);
}

pub trait GasPowered: Vehicle {
    fn service_fuel_tank(&self);
}
```