Revised: 12/3/2017

Aggregation and Inheritance

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Aggregation
- Inheritance
- Polymorphic Functions
- Multiple Inheritance
  - The Diamond Problem

Colors/Fonts

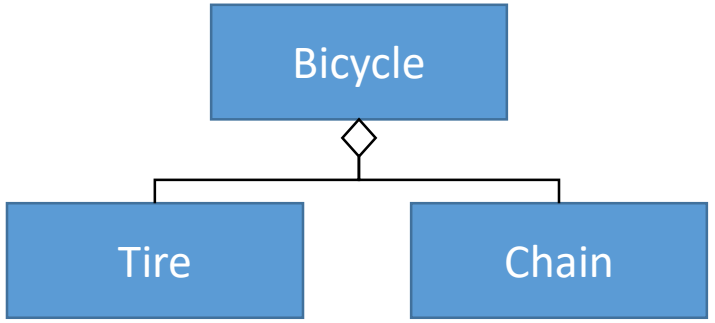
- Global Variable Names – Brown
- Local Variable Names – Lt Blue
- Literals – Blue
- Keywords – Orange
- Operators/Punctuation – Black
- Functions – Purple
- Parameters – Gold
- Comments – Gray
- Modules – Pink
- Object/Class Names – Green

Source Code – **Consolas**
Output – Courier New

Aggregation in Object Oriented Design

- Real-life objects are often comprised of several other objects.
  - For example, a bicycle is made up of tires, a chain, pedals, handlebars, etc.
    - Together, these smaller, simpler objects are used to create a larger, more complex object.

- A software object can be designed in a similar way, where we have the more complex objects *aggregating* more specific objects into it.



The “has a” Relationship

- In Object Oriented Design, aggregation is used to create a “has a” relationship among classes.
  - A bicycle “has” tires.
  - A car “has a” steering wheel.
  - A classroom “has a” whiteboard.

- The aggregated objects have attributes and behaviors.
  - The aggregating object incorporates these objects in its own design/functionality.

Aggregation in Object Oriented Design

- There is no special syntax or keywords for object aggregation.
- Aggregation is achieved by using objects as the fields of the aggregating class.
  - The data type of the field is the aggregated object's type.
- For example, a Bicycle class could have two fields, front_tire and back_tire.
  - Both of those fields could be Tire objects.

Aggregation in Object Oriented Design

- The below example shows a class for a Tire object.

tire.py

```
class Tire() :

  def __init__(self, pressure_in, radius_in) :
    self.pressure = pressure_in
    self.radius = radius_in

  def getpressure(self) :
    return self.pressure

  def setpressure(self, pressure_in) :
    self.pressure = pressure_in

  ----------//----------
```

Aggregation in Object Oriented Design

- The below example shows a Bicycle class aggregating Tire objects.

bicycle.py

```
from tire import Tire

class Bicycle :

  def __init__(self) :
    self.front_tire = Tire(45, 27)
    self.back_tire = Tire(50, 27)

  def getfrontpressure(self) :
    return self.front_tire.getpressure()

  def setfrontpressure(self, pressure_in) :
    self.front_tire.setpressure(pressure_in)
```

Aggregation in Object Oriented Design

- The use of a Bicycle object is demonstrated below.

bicycletest.py

```
from bicycle import Bicycle

def main() :
  test_bike = Bicycle()
  print(test_bike.getfrontpressure())        45
  test_bike.setfrontpressure(48)
  print(test_bike.getfrontpressure())        48

main()
```

Aggregation in Object Oriented Design

- To demonstrate how a Bicycle object can use an aggregate object, we'll add a speed field and a couple associated functions.

bicycle.py

```
class Bicycle :

  def __init__(self) :
    self.front_tire = Tire(45, 27)
    self.back_tire = Tire(50, 27)
    self.speed = 0

  ---------------//---------------

  def speedup(self) :
    self.speed += 5

  def getspeed(self) :
    return self.speed
```

Aggregation in Object Oriented Design

- We can use the tire pressure to determine how much speed to add.

bicycle.py

```
class Bicycle :

  ----------//----------

  def speedup(self) :
    if self.getfrontpressure() < 5 or self.getbackpressure() < 5 :
      self.speed = 0 #Tire is too flat
    else :
      self.speed += 5

  def getspeed(self) :
    return self.speed
```

Data Validation

- Where should data be validated in a “has a” relationship?

```
test_bike.setfrontpressure(5000)
test_bike.setfrontpressure("Blue")
test_bike.setfrontpressure(-10.3)
```

- Should data be validated in the Bicycle object (setFrontPressure) or in the Tire object (setPressure)?

Data Validation

- The object that uses/stores the data should be responsible for validating it.

tire.py

```
class Tire() :

    ----------//----------

  def setpressure(self, pressure_in) :
    if isinstance(pressure_in, int) :
      if pressure_in >= 0 and pressure_in <= 60 :
        self.pressure = pressure_in
      else :
        raise ValueError("Invalid Tire Pressure")
    else :
      raise TypeError("Invalid Data Type")
```

Data Validation

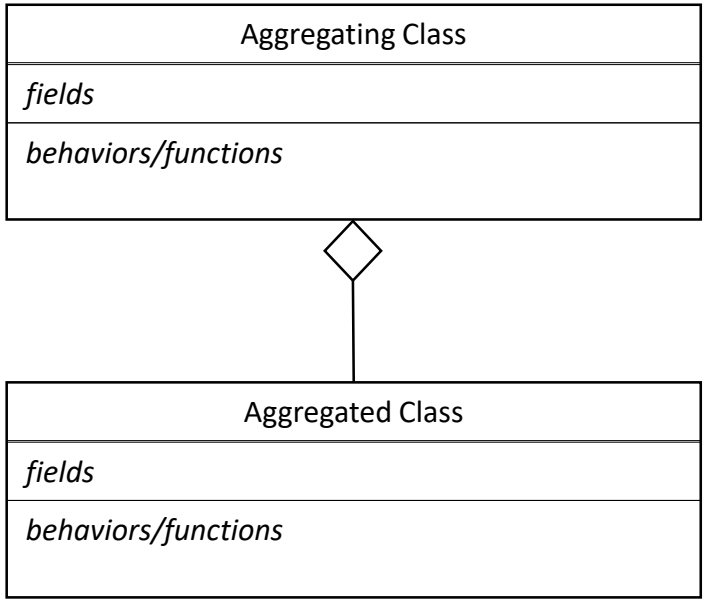
- However, the aggregating object needs to handle the effects of invalid data.

bicycle.py

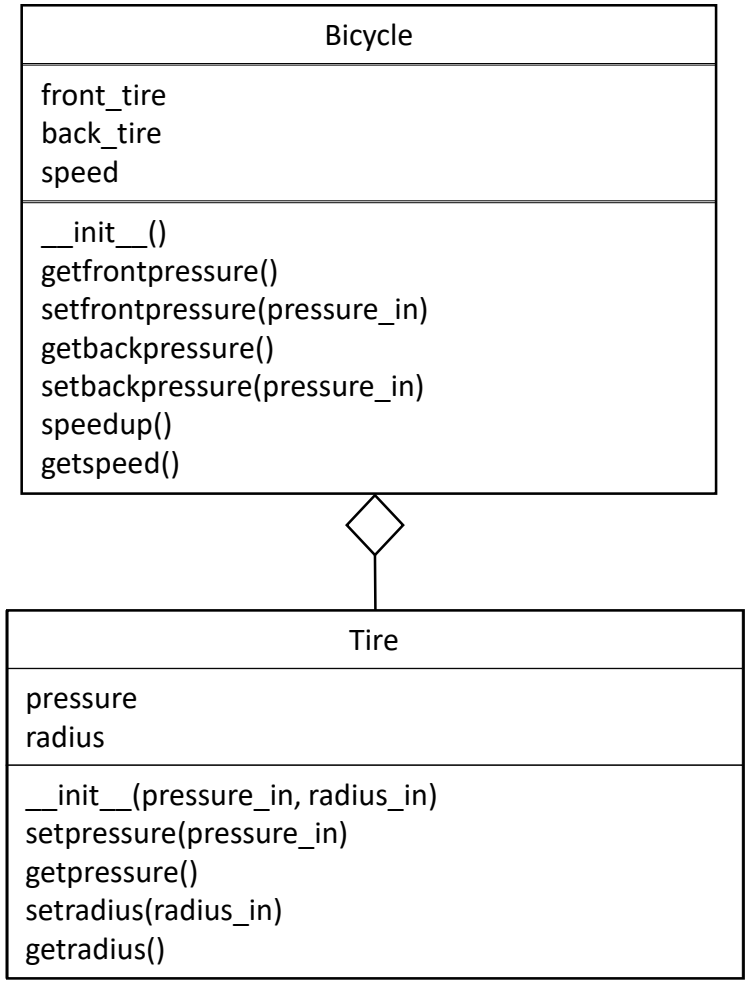
```
class Bicycle :

    ----------//----------

  def setfrontpressure(self, pressure_in) :
    try :
      self.front_tire.setpressure(pressure_in)
    except ValueError :
      print("Invalid Front Tire Pressure Provided")
    except TypeError :
      print("Invalid Front Tire Data Type Provided")
```

Aggregation in Class Diagrams

- Aggregation is shown in a Class Diagram using lines and a diamond shape.
  - The diamond is always below the aggregating class.



Aggregation in Class Diagrams



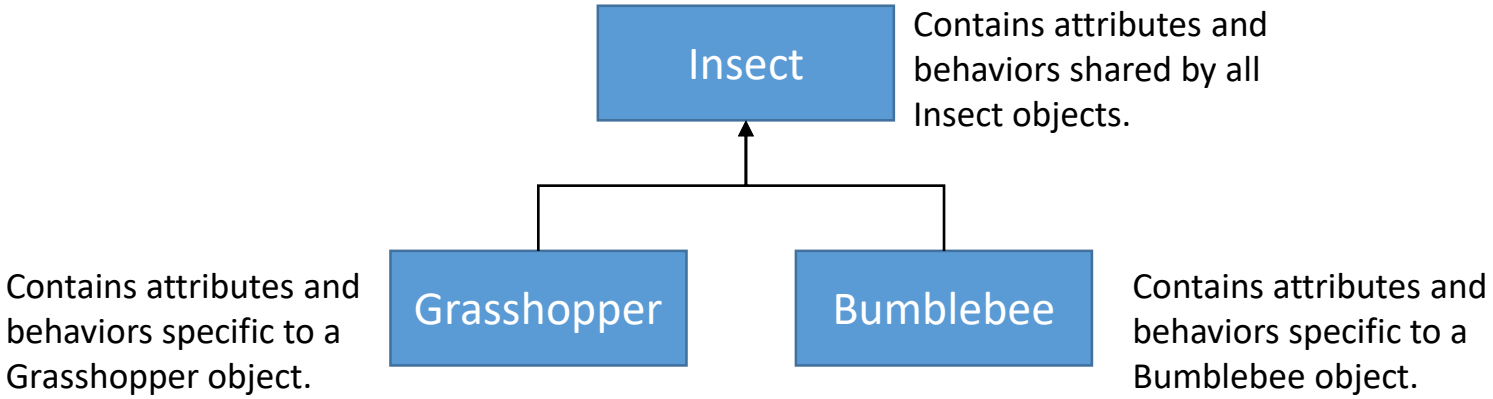
Inheritance in Object Oriented Design

- Real-life objects are often a specialized version of a more general object.
  - For example, a hammer and screwdriver are both tools.
    - They are both instruments used to build something.
    - But, they each have their own special use.
  - As another example, grasshoppers and bumblebees are both insects.
    - They share the general characteristics of an insect.
    - But, they each special characteristics of their own.
      - Grasshoppers have a jumping ability.
      - Bumblebees have a stinger.

Inheritance in Object Oriented Design

- An object oriented system can be designed in a similar way.

- We have the more specific objects *inheriting* attributes and behaviors from a more general object.



The “is a” Relationship

- In object oriented design, inheritance is used to create an “is a” relationship among classes.
  - A grasshopper “is an” insect.
  - A poodle “is a” dog.
  - A car “is a” vehicle.

- The specialized objects have:
  - All of the characteristics of the general object.
  - Additional characteristics that make it special.

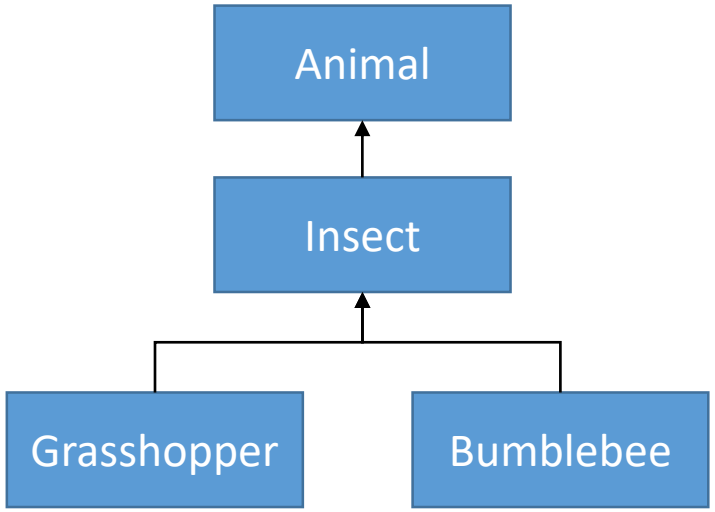
Superclasses and Subclasses

- A **superclass** (sometimes called a *base class* or *parent class*) is a class whose attributes and behaviors are inherited by other classes.
  - In the previous model, the Insect class is a superclass.

- A **subclass** (sometimes called a *derived class* or *child class*) is a class that inherits the attributes and behaviors of another class.
  - In the previous model, the Grasshopper and Bumblebee classes are subclasses.

Superclasses and Subclasses

- A class can be both a superclass and a subclass.
  - In the model below, the Insect class is the superclass of the Grasshopper and Bumblebee classes, but is itself a subclass of an Animal class.



Creating a Subclass

- The below example shows a superclass (Automobile) and a subclass (Car)

automobile.py

```
class Automobile :

  def __init__(self,
               make_in,
               model_in,
               year_in) :
    self.make = make_in
    self.model = model_in
    self.year = year_in
```

car.py

```
from automobile import Automobile

class Car(Automobile) :

  def __init__(self, make_in, model_in,
               year_in, doors_in) :
    Automobile.__init__(self,
                        make_in,
                        model_in,
                        year_in)
    self.doors = doors_in
```

Creating a Subclass

car.py

```
from automobile import Automobile

class Car(Automobile) :

  def __init__(self, make_in, model_in,
               year_in, doors_in) :
    Automobile.__init__(self,
                        make_in,
                        model_in,
                        year_in)
    self.doors = doors_in
```

If the superclass is in another file, it must be imported.

Indicates the superclass.

Call the superclass's initializer first. Pass as many arguments as possible to initialize its own fields with.

Initialize this class's fields.

Creating a Subclass

- Adding getters/setters to the two classes…

automobile.py

```
class Automobile :

    ----------//----------

  def getmake(self) :
    return self.make

  def setmake(self, make_in) :
    self.make = make_in

    ----------//----------
```

car.py

```
from automobile import Automobile

class Car(Automobile) :

    ----------//----------

  def getdoors(self) :
    return self.doors

  def setdoors(self, doors_in) :
    self.doors = doors_in
```

Creating a Subclass

- The use of a Car object (and it's inherited behavior) is demonstrated below.

autotest.py

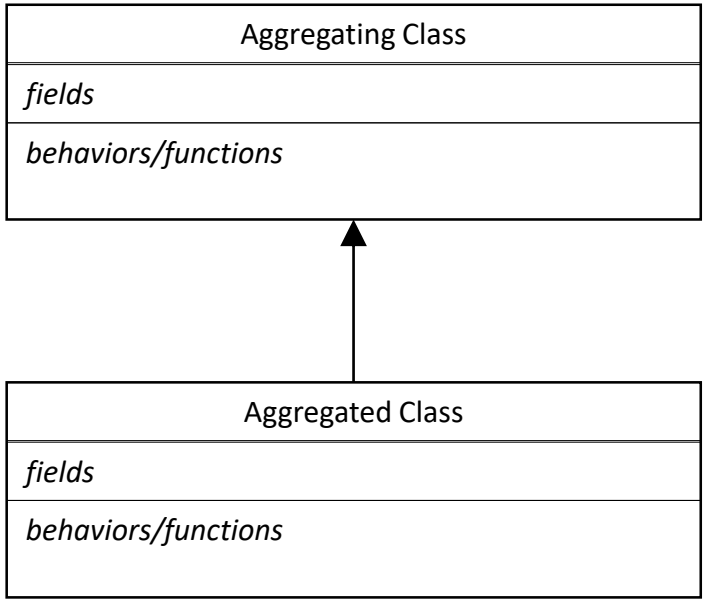
```
from car import Car

def main() :
  test_car = Car("Ford", "Focus", 2013, 2)
  print(test_car.getmake())
  print(test_car.getmodel())
  print(test_car.getyear())
  print(test_car.getdoors())

main()
```

Functions inherited from the superclass.

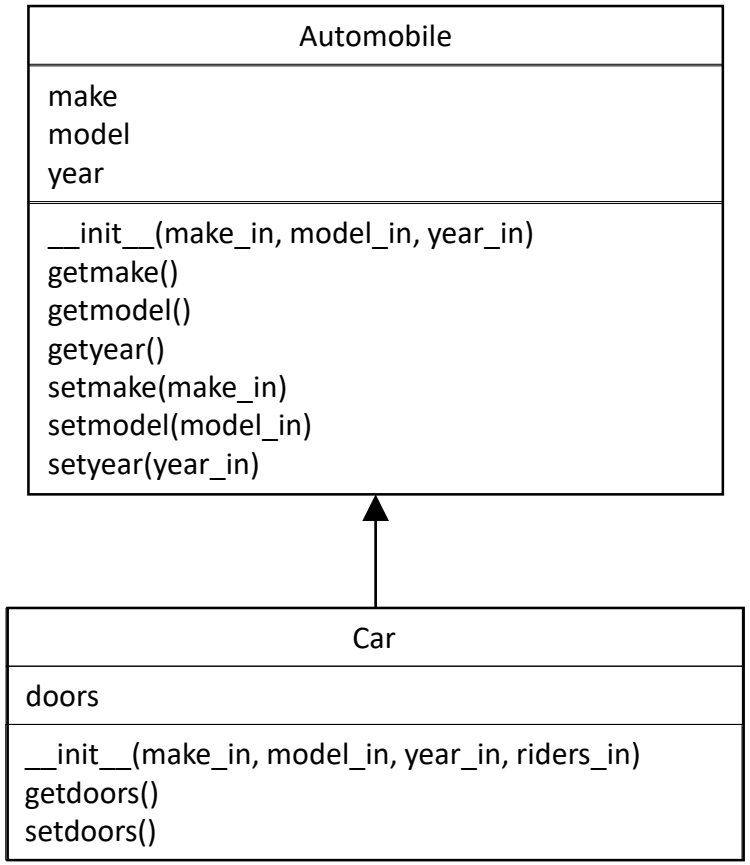
```
Ford
Focus
2013
2
```

Inheritance in Class Diagrams

- Inheritance is shown in a Class Diagram using arrows.
  - The arrow always points from the subclass to its superclass.



Inheritance in Class Diagrams



Polymorphic Functions

- Let's say both the Automobile and Car classes have a soundHorn function.

automobile.py

```
class Automobile :

    ---------//---------

  def soundhorn(self) :
    print("Honk Honk!")
```

car.py

```
from automobile import Automobile

class Car(Automobile) :

    ---------//---------

  def soundhorn(self) :
    print("Beep Beep!")
```

Polymorphic Functions

- Which function (the superclass's or subclass's) is called?

autotest.py

```
from car import Car

def main() :
  test_car = Car("Ford", "Focus", 2013, 2)
  test_car.soundhorn()

main()
```

- The subclass's soundHorn function is called.
  - The superclass's soundHorn function has been *overridden*.

Polymorphic Functions

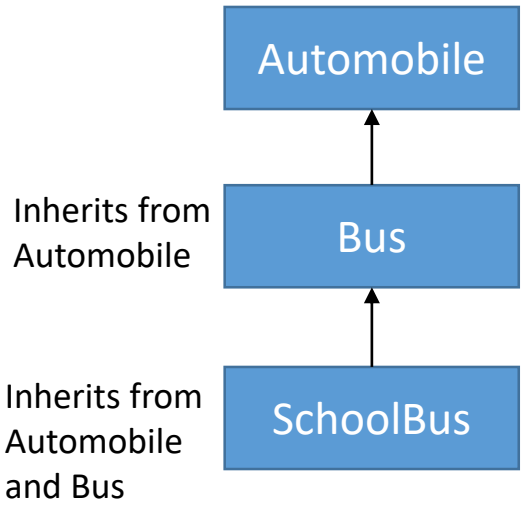
- An ***polymorphic function*** (or *override function*) is a function in a subclass, with the same name and parameter list length as a function in its superclass.

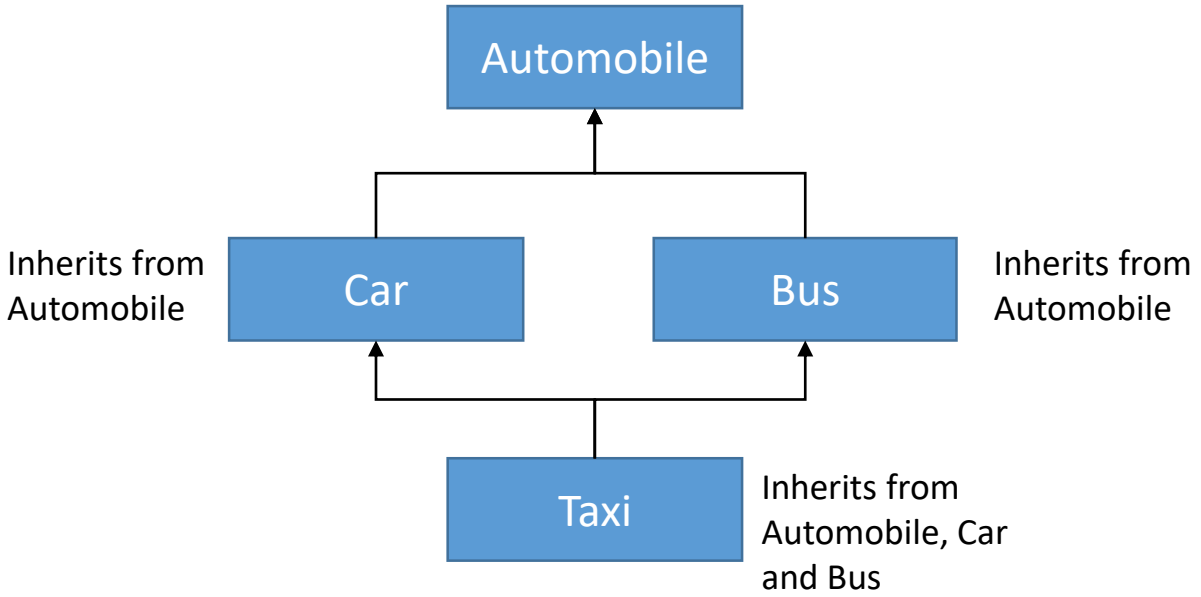
- When this occurs, the subclass's function is called instead of the superclass's function.

- This allows the programmer to override/morph a behavior inherited by a subclass.
  - Any other subclasses (without their own override) will still be using the superclass version of the function.

Multiple Inheritance

- ***Multiple inheritance*** describes a relationship where a subclass inherits from more than one superclass.





Multiple Inheritance

bus.py

```
from automobile import Automobile

class Bus(Automobile) :

  def __init__(self, make_in, model_in,
               year_in, riders_in) :
    Automobile.__init__(self,
                        make_in,
                        model_in,
                        year_in)
    self.riders = riders_in

  def getriders(self) :
    return self.riders
```

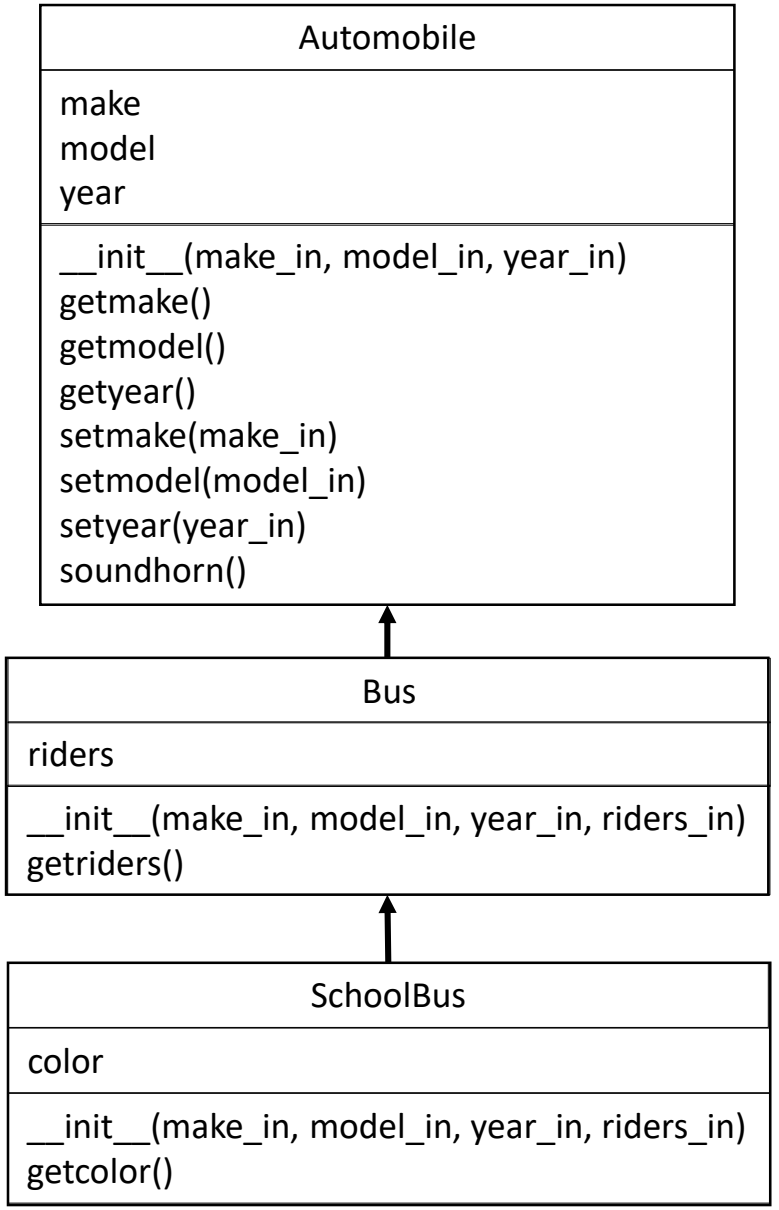
schoolbus.py

```
from bus import Bus

class SchoolBus(Bus) :

  def __init__(self, make_in, model_in,
               year_in, riders_in) :
    Bus.__init__(self, make_in, model_in,
                 year_in, riders_in)
    self.color = "Yellow"

  def getcolor(self) :
    return self.color
```

Multiple Inheritance

- The Bus class inherits attributes and behaviors from the Automobile class.

- The SchoolBus class inherits attributes and behaviors from the Bus class, as well as the attributes and behaviors of the Automobile class.



Multiple Inheritance

bus_test.py

```
from schoolbus import SchoolBus

def main() :
  test_bus = SchoolBus("Bus Co.", "SB101", 1998, 25)
  print(test_bus.getmake())
  print(test_bus.getmodel())
  print(test_bus.getyear())
  print(test_bus.getriders())
  print(test_bus.getcolor())

main()
```

```
Bus Co.
SB101
1998
25
Yellow
```

Multiple Inheritance

taxi.py

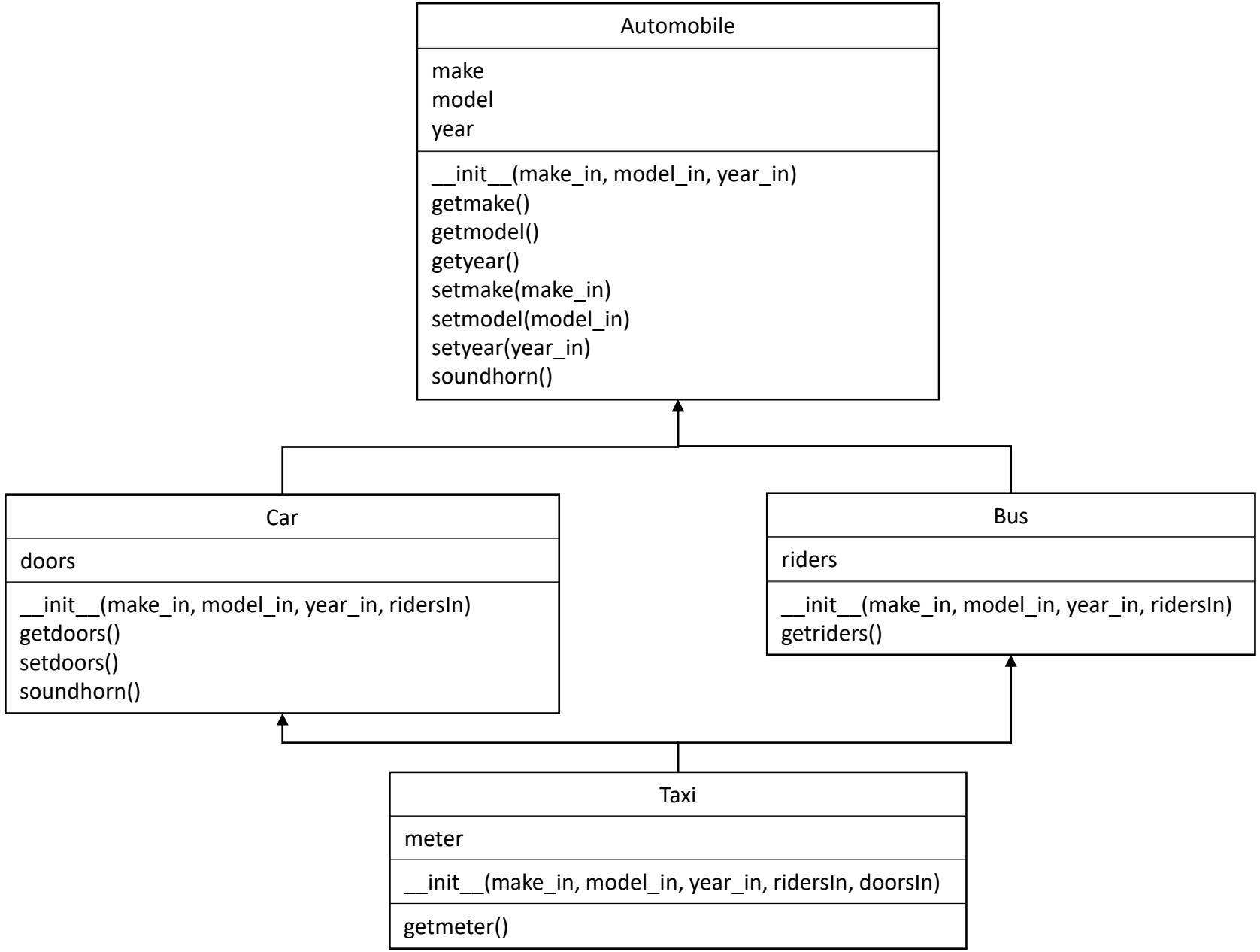
```
from bus import Bus
from car import Car

class Taxi(Bus, Car) :

  def __init__(self, make_in, model_in, year_in, riders_in , doors_in) :
    Bus.__init__(self, make_in, model_in, year_in, riders_in)
    Car.__init__(self, make_in, model_in, year_in, doors_in)
    self.meter = 5.0

  def getmeter(self) :
    return self.meter
```

Multiple Inheritance

taxi_test.py

```
from taxi import Taxi

def main() :
  test_taxi = Taxi("Yellow Cab", "NYC1", 2005, 4, 2)
  print(test_taxi.getmake())
  print(test_taxi.getmodel())
  print(test_taxi.getyear())
  print(test_taxi.getriders())
  print(test_taxi.getmeter())

main()
```

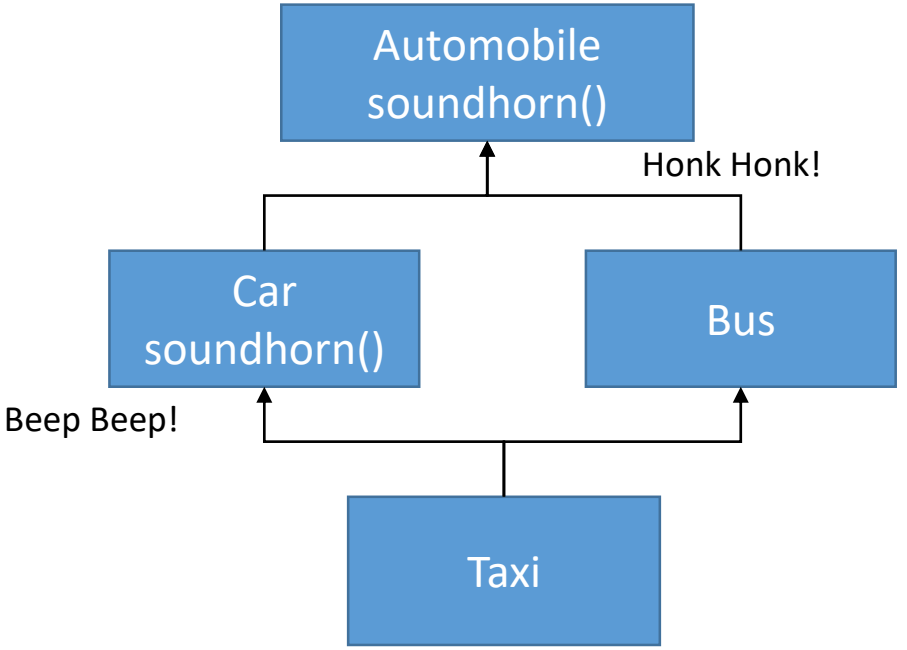
```
Yellow Cab
NYC1
2005
4
5.0
```


Automobile
make
model
year
__init__(make_in, model_in, year_in)
getmake()
getmodel()
getyear()
setmake(make_in)
setmodel(model_in)
setyear(year_in)
soundhorn()
Car
doors
__init__(make_in, model_in, year_in, ridersIn)
getdoors()
setdoors()
soundhorn()
Bus
riders
__init__(make_in, model_in, year_in, ridersIn)
getriders()
Taxi
meter
__init__(make_in, model_in, year_in, ridersIn, doorsIn)
getmeter()

The Diamond Problem

- The ***Diamond Problem*** occurs when there is a conflict between attributes or functions inherited from superclasses.

- Our Taxi class has this problem.
  - The Bus class inherits the Automobile class's soundhorn() function.
  - The Car class overrides the Automobile class's soundhorn() function.
  - Which one does the Taxi class inherit?



The Diamond Problem

- The closest function is the one inherited.
  - The Car class's soundhorn() function.

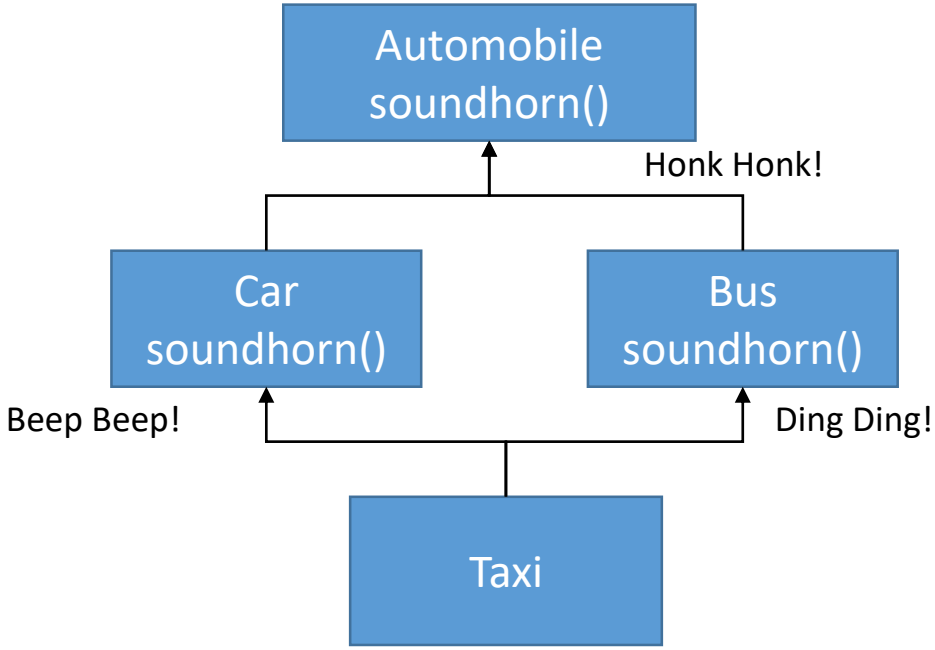
taxitest.py

```
from taxi import Taxi

def main() :
  test_taxi = Taxi("Yellow Cab", "NYC1", 2005, 4, 2)
  test_taxi.soundhorn()

main()
```

```
Beep Beep!
```

The Diamond Problem

- What if the Bus class also had an override function?
- Will the Taxi class inherit the soundhorn function from the Car class or the Bus class?



The Diamond Problem

- It depends on the order of the superclasses in the class header.

taxi.py

```
from bus import Bus
from car import Car

class Taxi(Bus, Car) :
```

taxitest.py

```
from taxi import Taxi

def main() :
  test_taxi = Taxi("Yellow Cab", "NYC1", 2005, 4, 2)
  test_taxi.soundhorn()

main()
```

```
Ding Ding!
```

The Diamond Problem

- It depends on the order of the superclasses in the class header.

taxi.py

```
from bus import Bus
from car import Car

class Taxi(Car, Bus) :
```

taxitest.py

```
from taxi import Taxi

def main() :
  test_taxi = Taxi("Yellow Cab", "NYC1", 2005, 4, 2)
  test_taxi.soundhorn()

main()
```

```
Beep Beep!
```