B-Y-UML

A Guide to UML Diagrams for CS 235

Forward

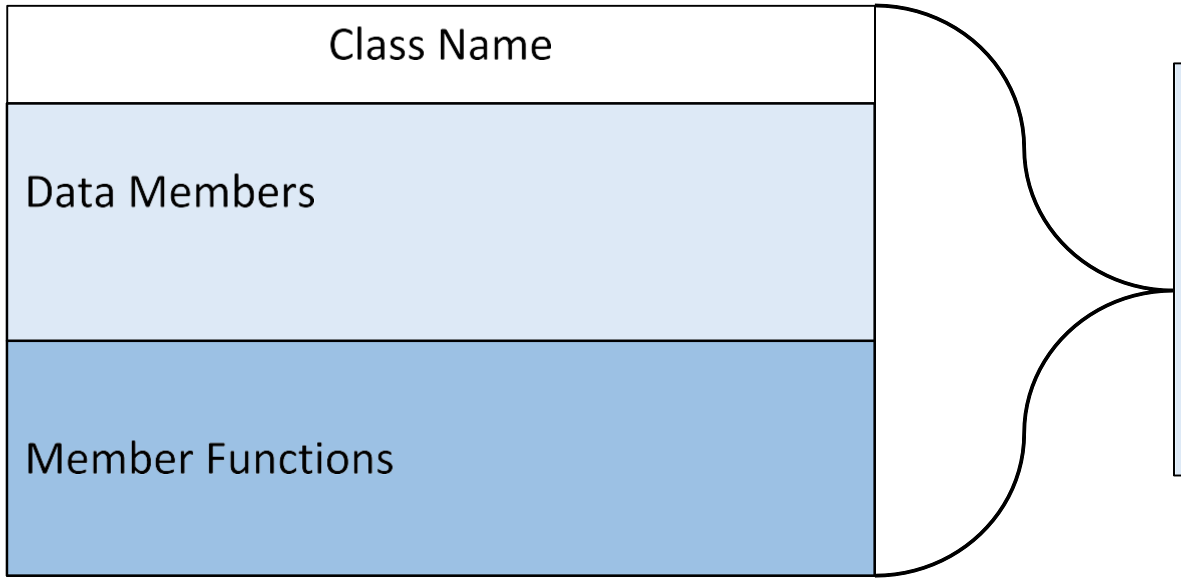
The Unified Modeling Language (UML) is a diagramming system that has been developed to help represent computer data structures and to show how structures interact. It is language independent, allowing you to show a data structure without really getting into the code. It can help you to better understand a project, and clearly shows how various classes interact.

Over time, UML has developed into a complex modeling system capable of showing many different types of structures and inter-class relationships.

This document is not meant to be the end-all-be-all of UML guides. Rather, it is designed to show you certain critical aspects of UML diagramming that will allow you to conceptualize the structures that you will build in this class. We will leave out a lot of other information, which will be given to you in CS 340.

Until then, we’re just going to focus on the basics.

The Basics



Class Name

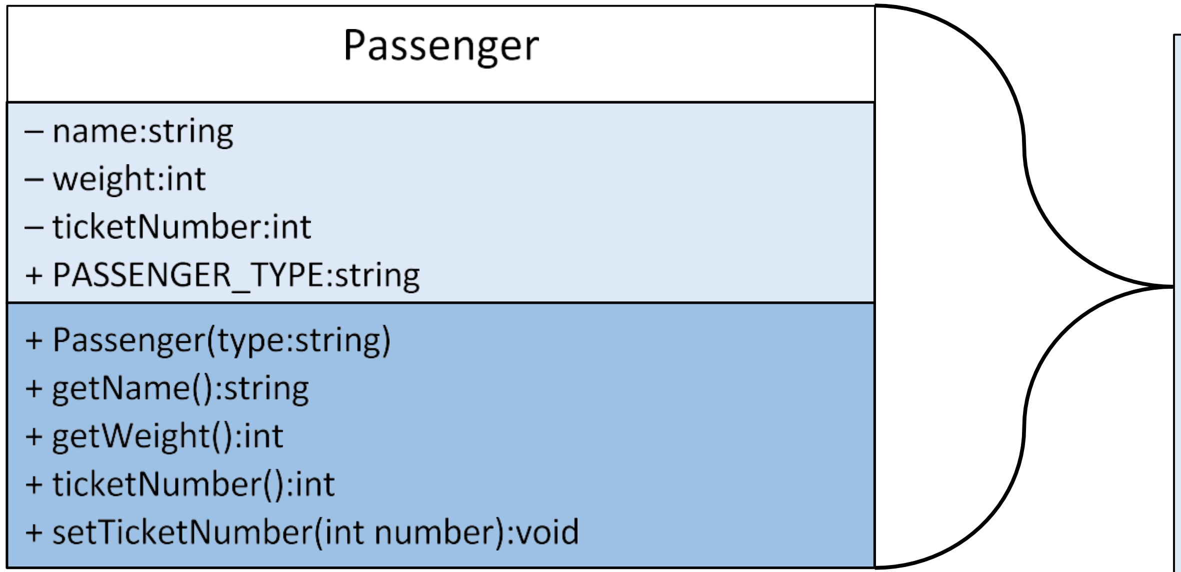
Data Members

Member Functions

These three boxes make up a class diagram. At the top is the name of the class.

The first section beneath that is filled with the types, names, and visibility of the data members.

The bottom section shows the return types, names, visibilities, and parameter types of the functions.



Passenger

```
– name:string
– weight:int
– ticketNumber:int
+ PASSENGER_TYPE:string
```

```
+ Passenger(type:string)
+ getName():string
+ getWeight():int
+ ticketNumber():int
+ setTicketNumber(int number):void
```

This is a sample class called Passenger

The data members are all listed by visibility, name, and type.

Visibility: A '-' in front of a data member or function means that the data or function is private. A '+' indicates public visibility, and a '#' indicates a protected visibility.

Type: You may indicate parameter type and return type before or after the name of the data member/function. e.g.

'+ string getName()' and '+ getName():string' are both acceptable.

Interfaces And Abstract Functions

Locomotive

+ AMOUNT_OF_COAL:int
MAX_PASSENGERS:int
# next_car:TrainCar*
# final_destination:String
# train_ID:int
# passengers: vector<Passenger*>
# velocity:int

+ Locomotive(ID:int, MAX_CAP:int)
+ *pull_brake():void*
+ *add_passenger(passenger:Passenger*):bool*
+ removePassenger(name:string):bool
+ getNextPassenger():Passenger*
+ get ID():int
+ addNextCar(nextCar:TrainCar*):void
+ removeNextCar():void
+ getNextCar():TrainCar*
+ getVelocity():int

There is an important distinction between a concrete class and an abstract class or interface.

An interface has strictly pure virtual functions.

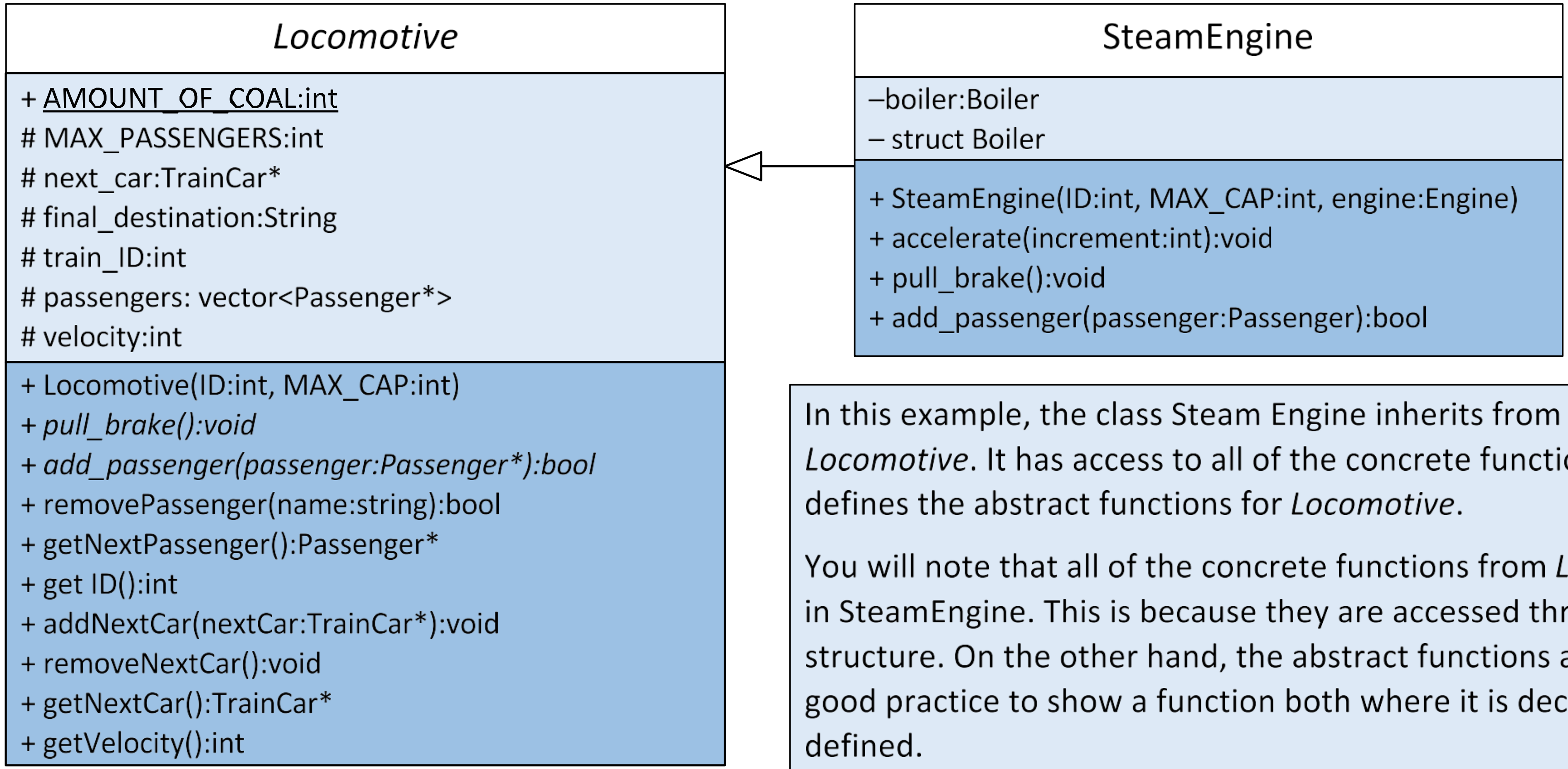
An abstract class has at least one pure virtual function, but other functions may be defined.

Both interfaces and abstract classes have italicized class names. Interfaces are denoted with <<interface>> (not in italics) above the class name.

Virtual functions are italicized.

In this example, because not all of Locomotive's functions are pure virtual, Locomotive is not an interface.

Inheritance ("Is a")


Locomotive
+ AMOUNT_OF_COAL:int
# MAX_PASSENGERS:int
# next_car:TrainCar*
# final_destination:String
# train_ID:int
# passengers: vector<Passenger*>
# velocity:int
+ Locomotive(ID:int, MAX_CAP:int)
+ *pull_brake():void*
+ *add_passenger(passenger:Passenger*):bool*
+ removePassenger(name:string):bool
+ getNextPassenger():Passenger*
+ get ID():int
+ addNextCar(nextCar:TrainCar*):void
+ removeNextCar():void
+ getNextCar():TrainCar*
+ getVelocity():int
SteamEngine
–boiler:Boiler
– struct Boiler
+ SteamEngine(ID:int, MAX_CAP:int, engine:Engine)
+ accelerate(increment:int):void
+ pull_brake():void
+ add_passenger(passenger:Passenger):bool

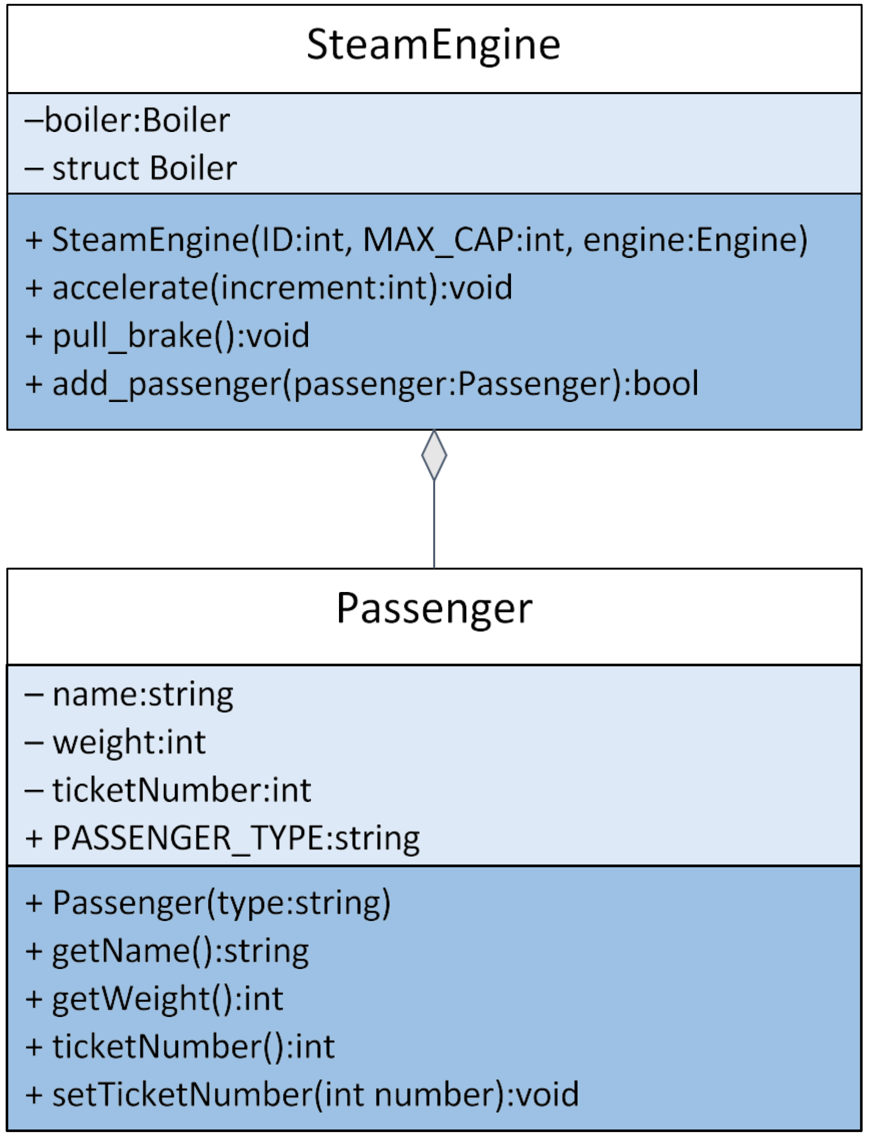
In this example, the class Steam Engine inherits from the abstract class *Locomotive*. It has access to all of the concrete functions of *Locomotive*, and also defines the abstract functions for *Locomotive*.

You will note that all of the concrete functions from *Locomotive* are not included in SteamEngine. This is because they are accessed through the inheritance structure. On the other hand, the abstract functions appear in both places. It is a good practice to show a function both where it is declared and where it is defined.

For the interfaces that you will use in this class, it is acceptable to omit the inherited functions from the child class.

Inheritance is indicated with a white triangle arrowhead facing the parent class.

Referencing/Association (“Has a”)

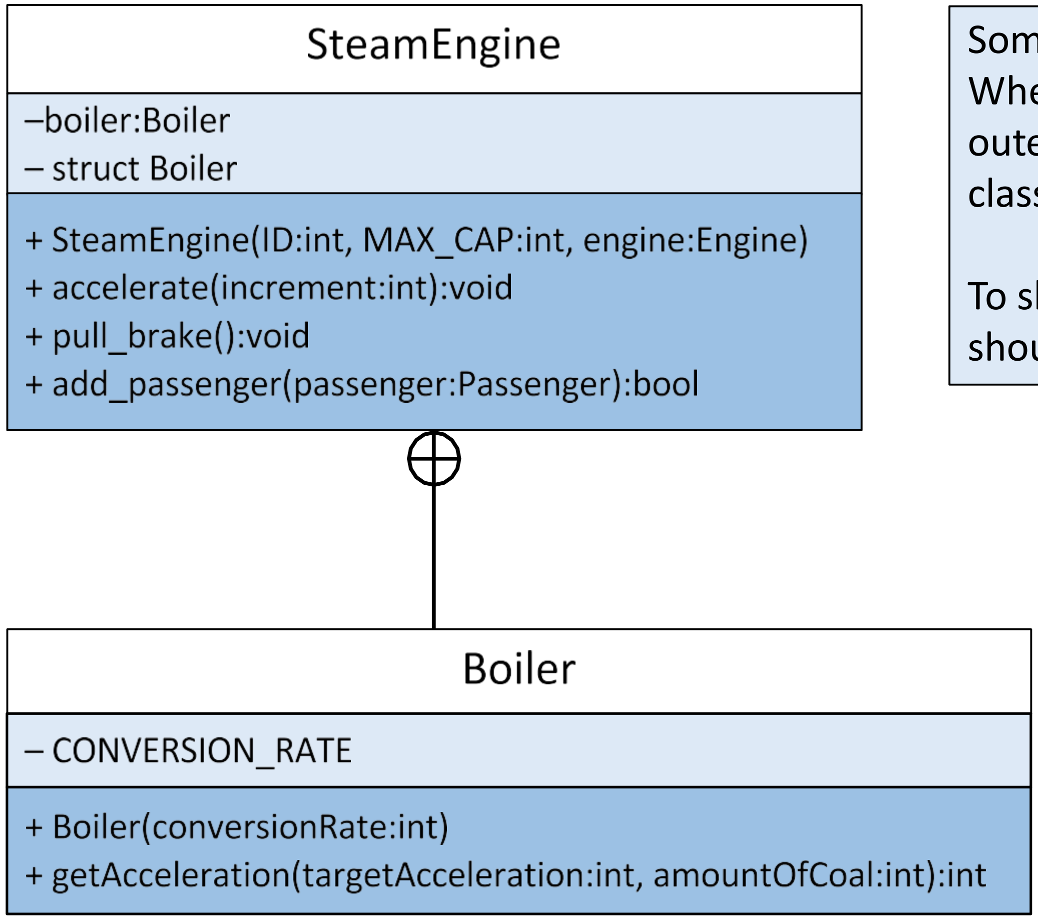


In this example, the class Steam Engine references the class Passenger, placing pointers to Passenger objects inside of its private vector.

This relationship is indicated by using a filled diamond arrow. The arrow faces towards the class that “has” the other thing in it.

For the purposes of this class, we do not distinguish between having an object and having a pointer to an object. Just use the same type of arrow.

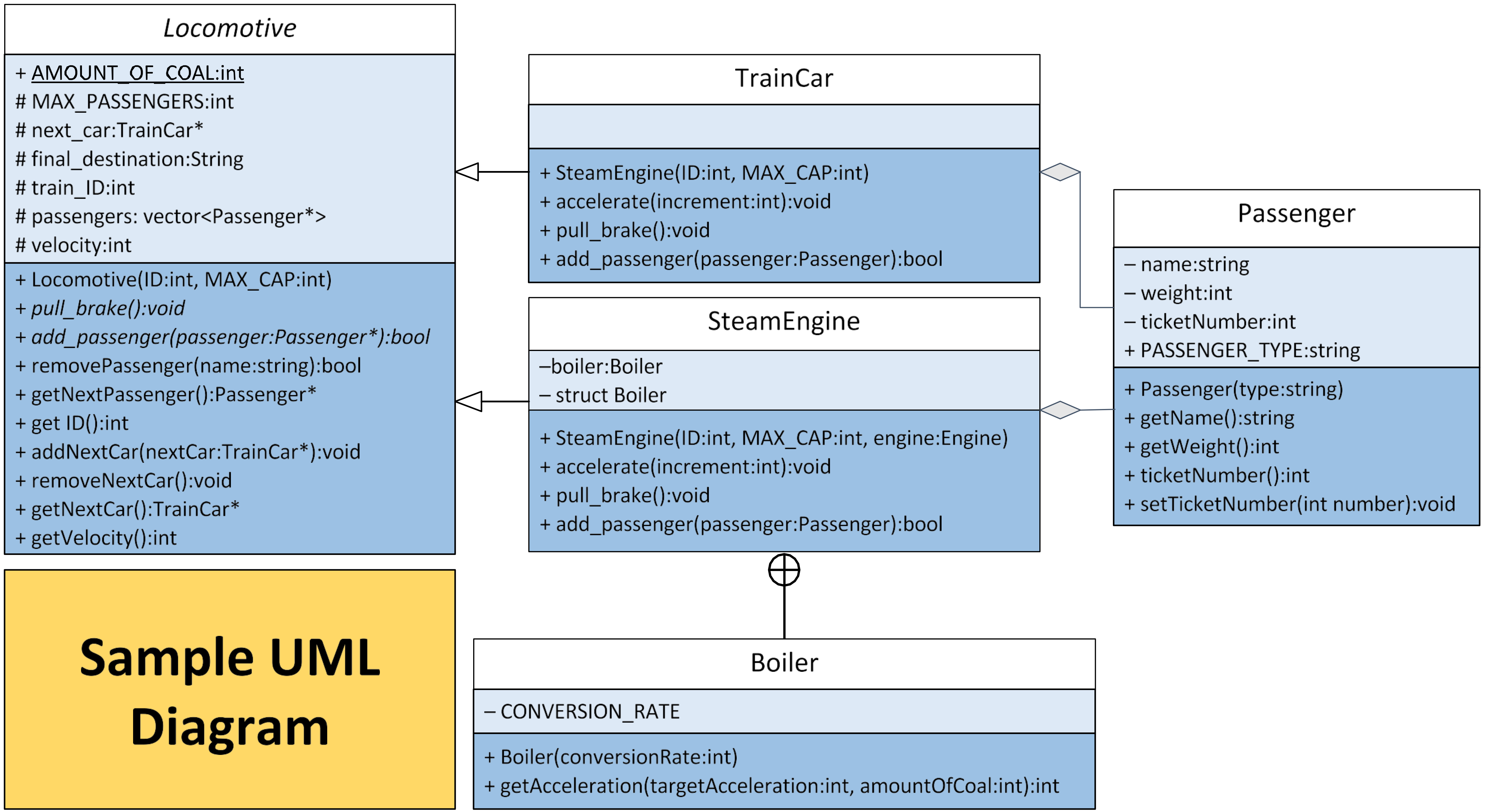
Inner Classes and Inner Structs


SteamEngine
–boiler:Boiler
– struct Boiler
+ SteamEngine(ID:int, MAX_CAP:int, engine:Engine)
+ accelerate(increment:int):void
+ pull_brake():void
+ add_passenger(passenger:Passenger):bool
Boiler
– CONVERSION_RATE
+ Boiler(conversionRate:int)
+ getAcceleration(targetAcceleration:int, amountOfCoal:int):int

Sometimes, we will define a class within a class, or a struct within a class. When we do this, we include the class/struct declaration inside of the outer class, and then create another box to hold the information for the class/struct.

To show this connection, we use a circle with a cross through it. The circle should be closest to the class that contains the inner class/struct.

Sample UML Diagram


Locomotive
+ AMOUNT_OF_COAL:int
# MAX_PASSENGERS:int
# next_car:TrainCar*
# final_destination:String
# train_ID:int
# passengers: vector<Passenger*>
# velocity:int
+ Locomotive(ID:int, MAX_CAP:int)
+ *pull_brake():void*
+ *add_passenger(passenger:Passenger*):bool*
+ removePassenger(name:string):bool
+ getNextPassenger():Passenger*
+ get ID():int
+ addNextCar(nextCar:TrainCar*):void
+ removeNextCar():void
+ getNextCar():TrainCar*
+ getVelocity():int
TrainCar
+ SteamEngine(ID:int, MAX_CAP:int)
+ accelerate(increment:int):void
+ pull_brake():void
+ add_passenger(passenger:Passenger):bool
SteamEngine
–boiler:Boiler
– struct Boiler
+ SteamEngine(ID:int, MAX_CAP:int, engine:Engine)
+ accelerate(increment:int):void
+ pull_brake():void
+ add_passenger(passenger:Passenger):bool
Passenger
– name:string
– weight:int
– ticketNumber:int
+ PASSENGER_TYPE:string
+ Passenger(type:string)
+ getName():string
+ getWeight():int
+ ticketNumber():int
+ setTicketNumber(int number):void
Boiler
– CONVERSION_RATE
+ Boiler(conversionRate:int)
+ getAcceleration(targetAcceleration:int, amountOfCoal:int):int

Resources

UML diagrams can be created using a variety of programs. Lucidchart is a diagraming service available on the Internet that can be used to create UML diagrams. A free account can be created on the website, lucidchart.com . You can also use MS office if you choose (though it is going to require more work). You are also free to look up other resources. No matter which online resource you use, make sure that you have rights to create and download/print your UML, and obey all regulations and laws regarding copyright and usage.

You are also welcome to draw your UML out by hand if you would like. If you choose to do this, **make sure that your graph is clean, orderly, and well drawn.** Sloppy diagrams are a pain for the TA's to read and understand, so we expect you to turn in a UML that looks nice and can easily be read. UML is all about being able to quickly and cleanly share information about your code structure, so orderliness is a must.

For more details on UML diagrams, you might find the following resources helpful:

Appendix B.1 of the course text ("Objects, Abstration, Data Structures and Design using C++" E. Koffman, 2006)

https://en.wikipedia.org/wiki/Unified_Modeling_Language (You only need the structuring part)

http://www.uml-diagrams.org/uml-25-diagrams.html (Lots of extra information)

http://www.tutorialspoint.com/uml/uml_standard_diagrams.htm (Covers UML without diagrams)