design decisions for Challenge One:

I put all kinds of persons into a person package, and all kins of buses into a bus package. For Challenge One, I mainly used the **decorator pattern**, **information hiding** and **polymorphism**, **delegation**, **Inheritance**.

Specifically, within more than 20 methods for a person type, I only need to modify getName(), getBoardingTime(), getCapacity(), getLuggageSpace(), getDisabledSpace() for each kind of person. Because except for these attributes, all person types share the same methods. I only need to write these methods once in the OrdinaryPerson class. I combined payment methods and time to get on bus as one attribute, boardingTime. I set the space of different type of person as capacity, luggageSpace, disabledSpace. I will check them in Bus's isBoardable() method to see if they can get on a bus.

information hiding and **polymorphism:** Overall, users cannot directly modify contents of each instance of each class, instead they have to use methods to get and modify instance variables. I also use the interface Person, which extends Entity, as a interface of different kinds of persons. Different specific classes implement the same original interface. They share some same method, and may have some specific methods.

decorator pattern: Take Person as example (the same as Bus), I have a Person interface, an AbstractPerson class, which is an abstract class that have a Person type instance, as the abstract decorator. I also have a concrete component called OrdinaryPerson, as the basic concrete class. OrdinaryPerson implements Person and have all concrete methods from its interface. Then I set some first level concrete decorator. For example, RFIDPerson extends AbstractPerson, the abstract decorator, it override some of its superclass's methods, such as getBoardingTime(). Actually, it updates superclass's method. Similarly, BikePerson, CashPerson, WheelchairPerson, etc. are first level concrete decorators. I wrote CashBikePerson, RFIDLuggagePerson, etc. as second level concrete decorators. For example, CashBikePerson extends CashPerson, and has an Person instance variable that is initialize as a BikePerson type in the constructor. In this way, by **delegation** and **inheritance**, the subclass can update some methods.

And Bus types are nearly the same. Moreover, each bus has the attributes of speedRatio and unboardingRatio. SpeedRatio is because buses have different engines, they can decrease the schedule time between two stop by 5 percent if the bus is later than schedule and speed up 10 percents at night. I did this in setRealTimeCurrentStop() method. unboardingRatio is for the multi door buses. It can decrease the unboarding time for passengers unboarding in this ratio, but cannot influence boarding time.

By these methods, **code can be reused** and I do not need to write same code. (I also use the **strategy pattern** in route planer, but that's not challenge one).

advantages and disadvantages of alternative design choices: By these patterns, I can realize **code reuse**. I haven't copy any code part during this homework because I don't have to. But this pattern may make my package full of different classes, because I have to build every class for each kind of person or bus, even they just differ some minor attributes and share most methods.