**Delegation Notes**

Design Principle: [Law of Demeter](https://en.wikipedia.org/wiki/Law_of_Demeter) – "Only talk to your neighbors." A class shouldn't have to know much about distant classes. A method/class should have limited knowledge of an object model.

**Example**

1. Consider the collaboration shown below. A *RegularFlight,* for example Flight 482 flies from NYC/LaGuardia to Atlanta at 9am every weekday. A *RegularFlight* contains references to actual occurrences on specific dates, *SpecificFlight*s, for example, instances of Flight 482 that occur on June 1,2,3,4,5. Each *SpecificFlight* has a number of *Booking*s where each *Booking* represents an individual person in a specific seat. As shown, the *flightNum* originates in the *RegularFlight* class. Suppose there is a method, *m1* in the *Booking* class that needs the *flightNum*. The implementation shown below violates the Law of Demeter as *Booking* is talking directly to *RegularFlight*. As shown by the dashed arrows, there are three dependencies between the classes.
2. A better way to handle this is to introduce a *flightNum* method in *SpecificFlight* that *delegates* to *RegularFlight*’s *flightNum* method. By making the reference to *SpecificFlight*’s reference to *RegularFlight* private, then we force *Booking* to use *SpecificFlight*’s *flightNum* method. Notice that now we have two dependencies. This adheres to the Law of Demeter.
3. Suppose there are several methods in *Booking* that require the *flightNum*. The approach below now has two dependencies of *Booking* on *SpecificFlight*.



1. A better approach is to have a *flightNum* method in *Booking* that delegates to *SpecificFlight* and then *m1* and *m2* use that method. This reduces the dependence.



1. Why is violation of the Law of Demeter a problem of changing requirements and debugging? If there is a change in *RegularFlight* not only do we have to look at its neighbors (*SpecificFlight*), but also the neighbors of neighbors (*Booking*) to assess the impact.