**Design Patterns**

# The Power Type Design Pattern

# Power Type
## Design Pattern

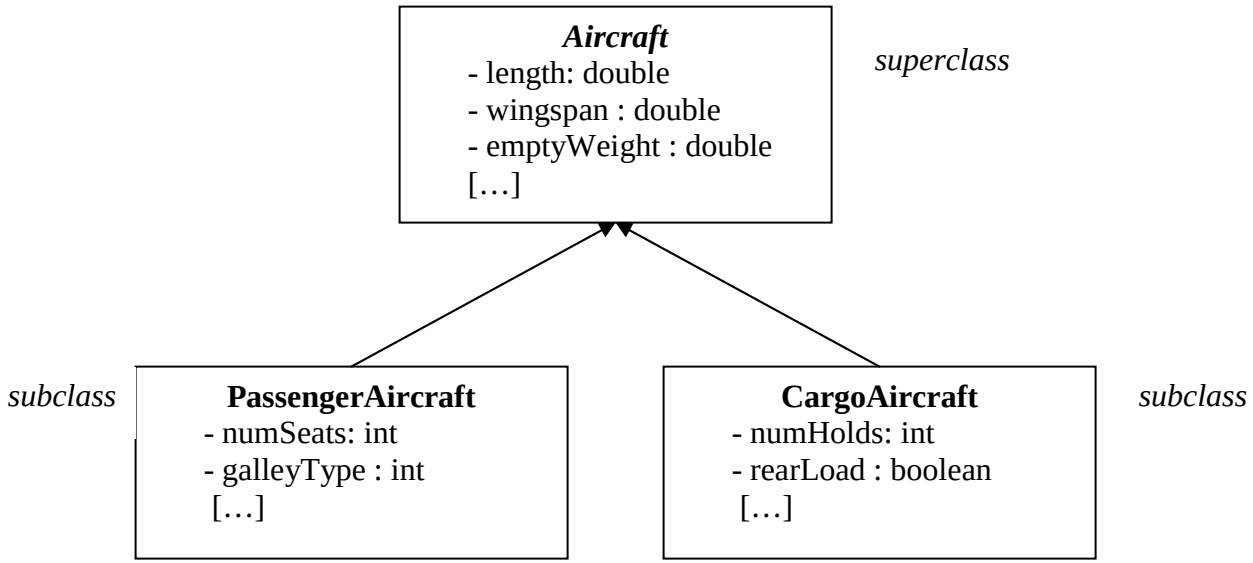
## The Issue:

Assume that you were designing an application that worked with various types of aircraft.

All of the aircraft have a common (generic) set of data and behaviors, but each may also have a set of specific data and behaviors particular to a given aircraft model. (i.e., passenger aircraft would have a different set of specific data and behaviors than would cargo aircraft).

Your design might quickly gravitate towards inheritance – the common set of data and behaviors would be part of a superclass, the specific data and behaviors particular to a given aircraft model would be located in subclasses.

Example:



## The Issue (cont):

This technique would work well, and additional types of aircraft could be introduced to the application by adding additional subclasses. These additions would not disturb any existing client code provided proper object oriented design practices were maintained.

However – there is a potential problem here. The problem does not involve the introduction of new subclasses, or even possible adverse effects on client code. The problem involves the duplication of data, as the number of actual aircraft objects grow.

Unlike other contexts, this does refer to multiply declaring the same data attributes in multiple classes – this refers to the existence of duplicated data values across hundreds, or thousands, or tens of thousands of objects.

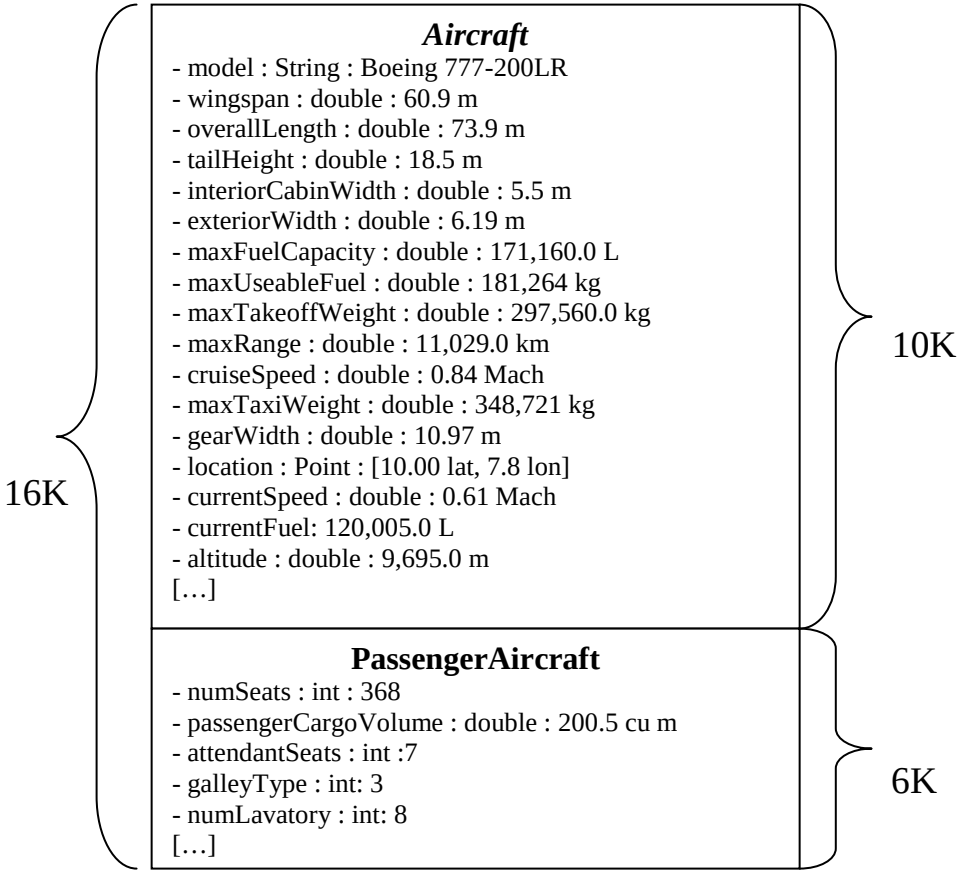
When we run our application, there may be thousands of passenger aircraft objects present at any given time. These thousands of aircraft objects would represent a set of real aircraft types such as Boeing 747, 777, 787, Airbus 320, 340, 380, etc.
Obviously, the set of real aircraft types is much smaller than the entire set of objects – perhaps 20 or 30 types of real aircraft vs. thousands or tens of thousands of aircraft objects. There are many, many actual aircraft created of a given type. In our application, those thousands of aircraft objects might be comprised of 1250 Boeing 747's, 2120 Boeing 777's, 450 Boeing 787's, 1125 Airbus 320's, etc.

This certainly makes sense, but it is also at the heart of the data value duplication problem.

## The Issue (cont):

**The Problem**

The following represents the data attributes and related values that one of our passenger aircraft objects might contain. Also shown are various size estimates for this aircraft object.




The entire aircraft object is approximately 16K. If there were 10,000 aircraft of this type in the application they would require 160,000K – 160 megabytes of space. Not an unreasonable amount, considering that PC's today commonly come with 1 – 2 gigabytes of real memory. However, this is just one type of aircraft. Remember, our application will have 20 – 30 types of aircraft represented. A large-scale application run could end up using almost 5 gigabytes of space. That *is* a significant amount.

Analysis of space utilization within our objects reveals some interesting findings. Of the 10,000 "Boeing 777-200LR" objects, only a small percentage of the data attribute values vary from one instance to another.

For example, the values for "location", "currentSpeed", "currentFuel", "altitude", etc. are different for each aircraft instance. However, the values for other data attributes such as "wingspan", "overallLength", "tailheight", etc. are the *same* for all "Boeing 777-200LR" instances.

## The Issue (cont):

Let's assume now that the data values that do *not* change from one instance of a "Boeing 777-200LR" to another makes up 12K of the overall PassengerAircraft size, and only the remaining 4K is made up of data attributes that are *different* for each aircraft instance.

That means that 75% of the space needed by our PassengerAircraft objects in the application will not change from one instance of a given aircraft type to another. Therefore, although we have 10,000 "Boeing 777-200LR" unique aircraft objects, 75% of the data in those objects is the same from one aircraft to the next. The same is true for aircraft objects of other aircraft types as well.

Therefore, of the 160 megabytes of space that the "Boeing 777-200LR" aircraft objects are using, 120 Meg of that is actually duplicated/copied across all those objects. There is in fact only 12K of unique data values present. Across objects of *all* aircraft types, that means that 75% of the previously mentioned 5 Gigabytes of space (3.75 Gb) is duplicated/copied data and there is only 120 megabytes of unique data values present

So – if all aircraft objects of a given type share a set of common data values, it is wasteful for each instance to maintain a copy of that data.

In fact, this duplication of data values is also error prone – what if someone mistakenly changes the value of one of these "common" data values (i.e., wingspan) for only one aircraft instance? Then one "Boeing 777-200LR" object would exist in our application with a different (and probably incorrect) data set than the rest of the "Boeing 777-200LR" objects – this would lead to inconsistent results that would be difficult to track down & debug.

How can we redesign these classes to remove the data value duplication issues and also insure that a single instance of an aircraft cannot have it's data values inadvertently changed in an inconsistent fashion?

## The Solution:

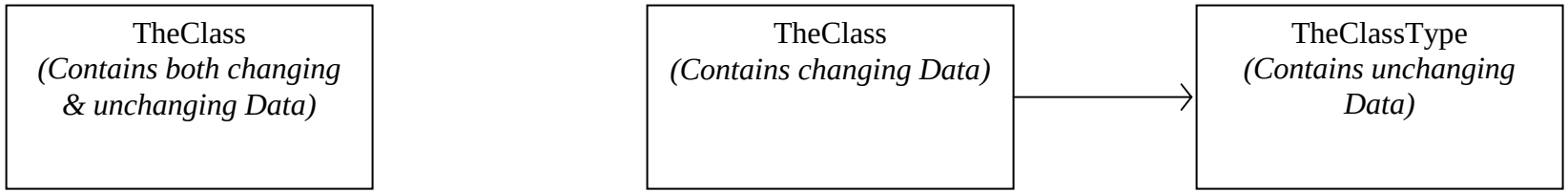
### The Power Type Pattern

The Power Type patterns specifies a technique for removing the unchanging "type" data from classes representing individual objects and instead creating a new second object that will contain that unchanging "type" data (i.e., the "Power Type").

The individual objects of a given type (i.e., the individual aircraft in our example) will own a reference to a "power type" object that they can use to access the power type object data values.

Shown below, "TheClass" as originally designed contains data attributes whose values are the same for all objects of a given type as well as data attributes whose values are different for each object of a given type.

The Power Type pattern splits this "TheClass" into 2 classes – "TheClass" and "TheClassType"

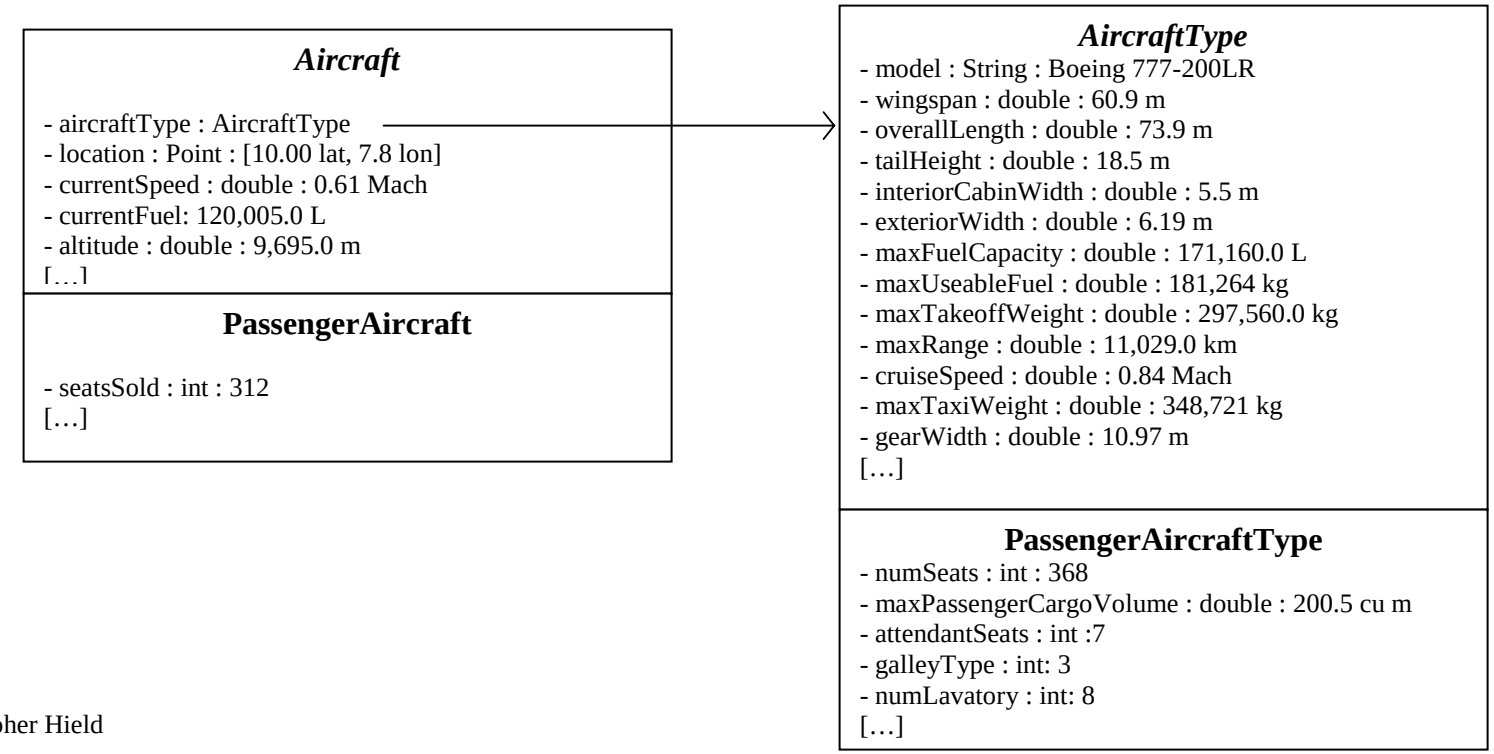


## The Solution (cont.):

**Power Type Example**

In our aircraft example, the original "Aircraft" class would be changed to include only the data that is different for each aircraft object of a given type (i.e., only the data that would actually vary from aircraft to aircraft).

The remainder of the data attributes would be migrated to a new class called "AircraftType" which would contain the data values that are common to all aircraft of a given type (i.e., all "Boeing 777-200LR"). All aircraft objects representing the "Boeing 777-200LR" aircraft would contain a reference to the "AircraftType" object that represents the "Boeing 777-200LR" aircraft.

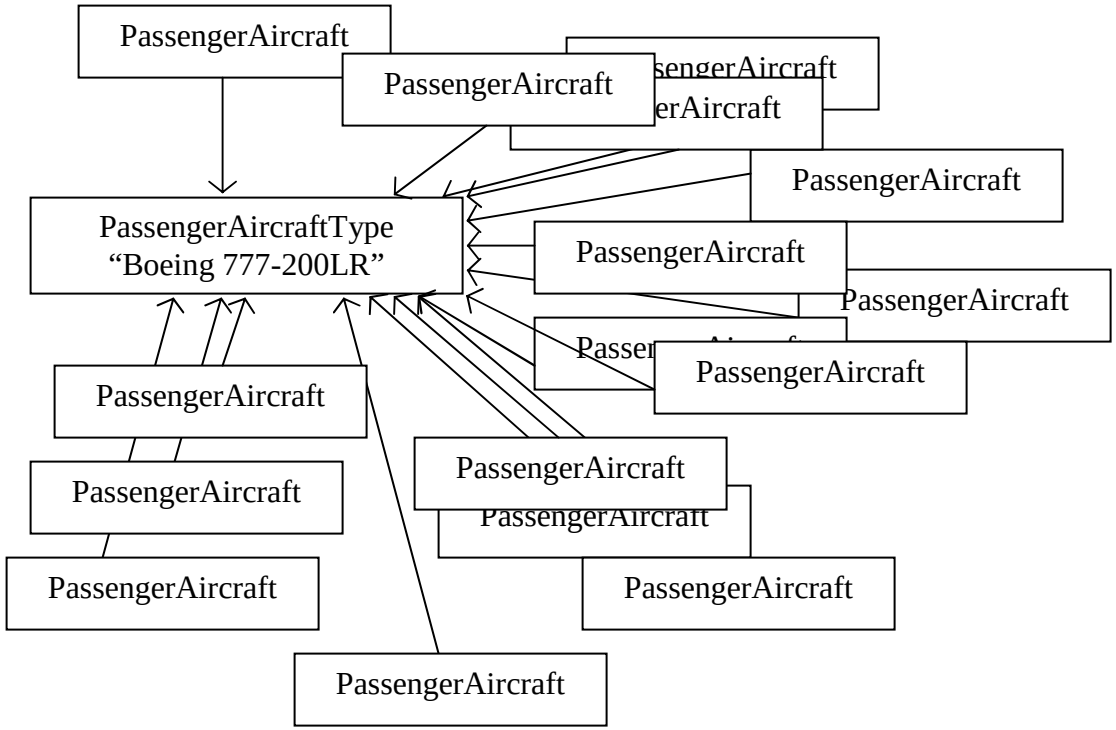


# Power Type
## Design Pattern

## The Solution (cont.):

**Power Type Example**

In our application, the 10,000 “Boeing 777-200LR” objects would now be 10,000 PassengerAircraft objects, each of which contains a reference to the one PassengerAircraftType object that represents the “Boeing 777-200LR”.




Rather than have thousands of copies of the data values common to all “Boeing 777-200LR” (one copy in each aircraft object), we have one copy of that data – in the “PassengerAircraftType” object.

All of the “Boeing 777-200LR” PassengerAircraft objects refer to that one “PassengerAircraftType” object for aircraft “type” related data.

This is often used in conjunction with delegation.
The “PassengerAircraft” class can contain get/set methods related to the data that is *now* part of the “PassengerAircraftType” class. Those methods will delegate the calls to the power type object.This makes the

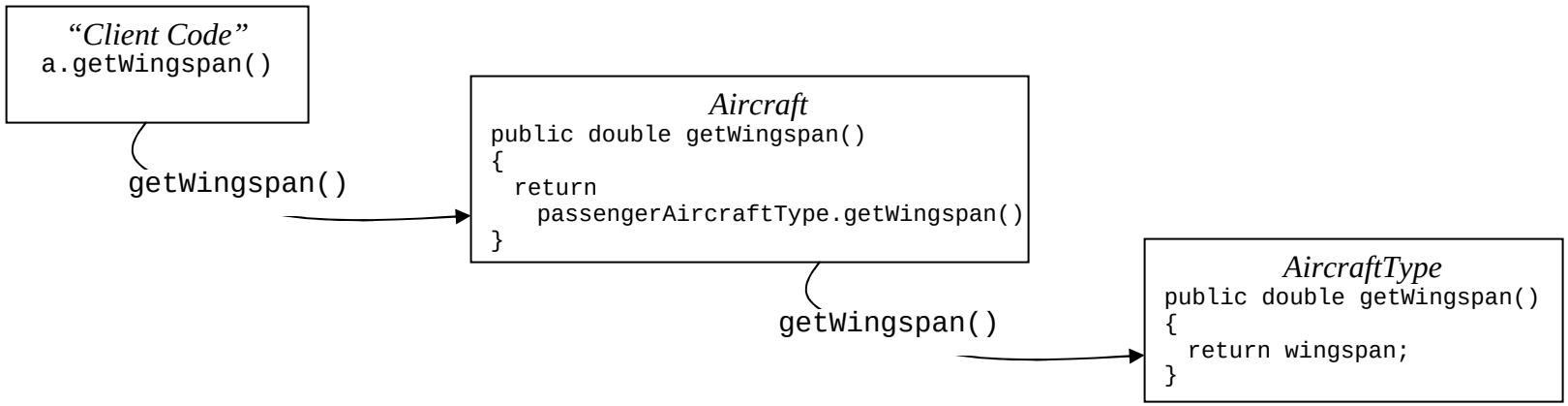
“PassengerAircraft “ object appear seamless to client code.

## The Solution (cont.):

**Power Type Example**

Assume that the "getWingSpan()" method was originally a part of the "Aircraft" class – this method returned the current "wingspan" data value. The design change (to make use of the Power Type pattern) moved the "wingspan" data attribute to the "AircraftType" class.

However, it is still necessary for a given aircraft object to be able to supply it's wingspan value to other objects when needed. Therefore, the "Aircraft" class would *still* need to have a "getWingSpan()" method – but that method will simply delegate the call to the "AircraftType" object, as shown:




Those methods that do not rely on the "AircraftType" class data attributes do not need to do any delegating, they will perform the method behavior themselves as expected.

## The Solution (cont.):

**Power Type**

Using this design pattern will significantly reduce the memory needs of our application:

Original:

- 12K duplicated data per object * 10,000 objects = 120 Megabytes duplicated data
  - 120 Megabytes duplicated data * 30 different aircraft types = 3.6 Gigabytes duplicated data

- 4K unique data per object * 10,000 objects = 40 Megabytes unique data
  - 40 Megabytes unique data * 30 different aircraft types = 1.2 Gigabytes unique data

4.8 Gigabytes total.

With Power Type:

- 12K unique type data per object * 1 objects = 12K unique type data
  - 12K unique type data * 30 different aircraft types = 360 Megabytes unique type data

- 4K unique data per object * 10,000 objects = 40 Megabytes unique data
  - 40 Megabytes unique data * 30 different aircraft types = 1.2 Gigabytes unique data

1.56 Gigabytes total.

## The Solution (cont.):

**Power Type**

Using the Power Type design pattern reduces the chances that one might inadvertently modify the common “type” data of one single aircraft instance.

For example, a “Boeing 777-200LR” aircraft located in a servicing facility could have it’s “wingspan” attribute inadvertently modified by some client code, thereby creating one “Boeing 777-200LR” with a wingspan different than all the rest.
And what if the new “wingspan” value was larger than the previous value, and now the application finds that the aircraft is too large to exit from the service building doors?

Since the “AircraftType” object is a delegate, no “set” capabilities need to be given to the individual “Aircraft” objects. The “AircraftType” objects should manage their own data attributes.

Changes to “AircraftType” object data affect all related “Aircraft” instances.
If the AircraftType object representing a “Boeing 777-200LR” has it’s “wingspan” attribute changed, then since all “Boeing 777-200LR” aircraft objects refer to the same AircraftType object, then all “Boeing 777-200LR” aircraft objects would experience that change.

# Power Type
## Design Pattern

**Summary:**

- The Power Type patterns specifies a technique for removing the unchanging "type" data from classes representing individual objects and instead creating a new second object that will contain that unchanging "type" data (i.e., the "Power Type").

- The Power Type pattern helps to reduce the space needed by a set of objects by removing the need to store duplicate data values.

- The Power Type pattern helps to reduce application errors by centralizing common data values and removing the chance that an individual object may change it's own version of "type" based data.

- Power Type pattern has the following features:
  - Two classes, a type class and an instance class.
  - The instance class has an instance variable whose type is the type class.
  - The instance class delegates its type behavior to the type class via the instance variable.

- The Power type pattern may also include these variations on the pattern:
  - The system may maintain a list of its type class instances.
  - The type class instances may maintain a list of their instances.