

# Re: Retrieving unnecessary data

---

*Source:* <http://coding.derkeiler.com/Archive/General/comp.object/2008-02/msg00126.html>

---

- *From:* "H. S. Lahman" <[hsl@xxxxxxxxxxxxxxxxxxxxx](mailto:hsl@xxxxxxxxxxxxxxxxxxxxx)>
  - *Date:* Fri, 15 Feb 2008 15:14:29 GMT
- 

Responding to ShaneLM...

I'd like some advice on how to structure my classes. Let me use an example to convey my situation. Pretend I am modeling a Car.

```
Class Car
{
String make;
String model;
Enum color;
List<People> passengers;
}
```

My problem arises because sometimes I'm just interested in the stats of the car, and not the people in the car. Although in other situations I do care about the people in the car.

Let's go back to basic OOA/D. What you have in the problem space is:

```
[Car]
+ make
+ model
+ color
| 1
|
| R1
|
| carries
| 0..*
[Passenger]
```

That is, a Passenger is a peer entity relative to Car, not an intrinsic part of a car. The problem you have is to implement the R1 association at OOP time. Typically that would be done with a collection class and [Car] would have a reference to the collection class.

If no one but [Car] ever needs to know about passengers, then that reference could be private or the collection class instance could be an embedded object in [Car]'s implementation. However, if other objects need to

## Re: Retrieving unnecessary data

interact with a car's passengers, then they need to navigate the R1 relationship to get that particular car's passengers. In that situation they need public access to the association implementation so you would need a public getter for the collection class reference.

It is important to note that association implementation, instantiation, and navigation is orthogonal to class semantics. That allows us to use library classes for collections and provide idioms for access. Thus one can implement the association in a manner that completely decouples it from the class semantics needed to solve the problem in hand. (As a bonus, one can use encapsulation of the relationship instantiation to isolate and capture problem space rules and policies to limit access.)

For example, keeping track of whether the Car in hand actually has passengers at the time of collaboration is a logical responsibility of the collection class. So whoever needs to talk to a car's passengers would determine that through accessing the collection class rather than the Car in hand. More to the point, the client would do that exactly the same way for quite different objects than Car and Passenger. IOW, one has an OOP construction idiom that is independent of the objects.

If I'm writing a database access method, say GetCar(), I feel like time is wasted retrieving info about the passengers that I will never use in some situations. So I feel like I have a few alternatives:

This is a different problem that has to do with encapsulating DB access...

A) Return the class with all fields filled out, regardless of whether they will be used or not. This is OO, right?

Not OO per se. It is an optimization issue. Typically the DB access will be the performance bottleneck so one may need to optimize it via caching. IOW, one might want to read all the possibly relevant data from the DB in one transaction and then worry about how much of it is actually needed later.

Typically this sort of caching would be done in the subsystem that encapsulates the DB paradigm. The application solution would ask for Car data, Passenger data, or both depending on the solution context. Whether than data was already cached in the subsystem or needed to be read from the DB would be transparent to the solution.

B) Create overloads for GetCar where I indicate whether I want all fields filled out for me. Ex: GetCar(bool fill\_passengers);

Typically the interface to the DB access subsystem (think: Facade pattern) would provide whatever interfaces the solution needs for its various contexts. If the solution needs both then an interface method would provide that. If the solution needs just Car data or just Passenger data in some other context, an interface method would be provided for just that.

Re: Retrieving unnecessary data

C) Remove the passengers field, and retrieve them with another call. I'd call GetCar(), and then if I want the passenger list, I'd call GetPassengers(Car);

This is quite close to the above discussion. However, in the solution Car objects and Passenger objects are different, peer abstractions. They might be created and instantiated from DB data. But the R1 association will be assigned dynamically in the solution as particular passengers are associated with a particular car.

D) Make 2 classes – Car and CarWithPassengers (that can derive from Car).

Yech. In the problem space cars and passengers are quite differnt things.

—

There is nothing wrong with me that could not be cured by a capful of Drano.

H. S. Lahman

hsl@xxxxxxxxxxxxxxxxxxxx

Pathfinder Solutions

<http://www.pathfindermda.com>

blog: <http://pathfinderpeople.blogs.com/hslahman>

"Model-Based Translation: The Next Step in Agile Development". Email

info@xxxxxxxxxxxxxxxxxxxx for your copy.

Pathfinder is hiring: [http://www.pathfindermda.com/about\\_us/careers\\_pos3.php](http://www.pathfindermda.com/about_us/careers_pos3.php).

(888)OOA-PATH

.