How can an object-oriented programmer get his/her head around database-driven programming?

Asked 15 years, 11 months ago Modified 15 years, 11 months ago

Viewed 13k times

43

I have been programming in C# and Java for a little over a year and have a decent grasp of object oriented programming, but my new side project requires a database-driven model. I'm using C# and Linq which seems to be a very powerful tool but I'm having trouble with designing a database around my object oriented approach.

My two main question are:

How do I deal with inheritance in my database? Let's say I'm building a staff rostering application and I have an abstract class, Event. From Event I derive abstract classes ShiftEvent and StaffEvent. I then have concrete classes Shift (derived from ShiftEvent) and StaffTimeOff (derived from StaffEvent). There are other derived classes, but for the sake of argument these are enough.

Should I have a separate table for ShiftEvents and StaffEvents? Maybe I should have separate tables for each concrete class? Both of these approaches seem like they would give me problems when interacting with the

database. Another approach could be to have one Event table, and this table would have nullable columns for every type of data in any of my concrete classes. All of these approaches feel like they could impede extensibility down the road. More than likely there is a third approach that I have not considered.

My second question:

How do I deal with collections and one-to-many relationships in an object oriented way?

Let's say I have a Products class and a Categories class. Each instance of Categories would contain one or more products, but the products themselves should have no knowledge of categories. If I want to implement this in a database, then each product would need a category ID which maps to the categories table. But this introduces more coupling than I would prefer from an OO point of view. The products shouldn't even know that the categories exist, much less have a data field containing a category ID! Is there a better way?

sql linq database-design oop

Share

Improve this question

Follow

asked Jan 26, 2009 at 22:08

Martin Doms

8,748 ● 11 ● 45 ● 62

4 This is such a great question. You'll get great answers but few upvotes. That's how we roll here on SO.
– Dan Rosenstark Jan 26, 2009 at 22:47

He's right.. over 4 years later and only 22 votes. This question deserves better. – Inversus May 21, 2013 at 1:01

11 Answers

Sorted by: Highest score (default)

9

Linq to SQL using a table per class solution:

http://blogs.microsoft.co.il/blogs/bursteg/archive/2007/10/01/linq-to-sql-inheritance.aspx

Other solutions (such as my favorite, LLBLGen) allow other models. Personally, I like the single table solution with a discriminator column, but that is probably because we often query across the inheritance hierarchy and thus see it as the normal query, whereas querying a specific type only requires a "where" change.

All said and done, I personally feel that mapping OO into tables is putting the cart before the horse. There have been continual claims that the impedance mismatch between OO and relations has been solved... and there have been plenty of OO specific databases. None of them have unseated the powerful simplicity of the relation.

Instead, I tend to design the database with the application in mind, map those tables to entities and build from there.

Some find this as a loss of OO in the design process, but in my mind the data layer shouldn't be talking high enough into your application to be affecting the design of the higher order systems, just because you used a relational model *for storage*.

Share Improve this answer Follow

edited Jan 26, 2009 at 22:23

answered Jan 26, 2009 at 22:17

Godeke

16.3k ● 4 ● 64 ● 87

7

I had the opposite problem: how to get my head around OO after years of database design. Come to that, a decade earlier I had the problem of getting my head around SQL after years of "structured" flat-file programming. There are jsut enough similarities betwwen class and data entity decomposition to mislead you into thinking that they're equivalent. They aren't.

I tend to agree with the view that once you're committed to a relational database for storage then you should design a normalised model and compromise your object model where unavoidable. This is because you're more constrained by the DBMS than you are with your own code - building a compromised data model is more likley to cause you pain.

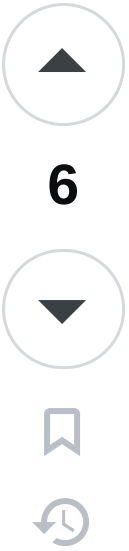
That said, in the examples given, you have choices: if ShiftEvent and StaffEvent are mostly similar in terms of attributes and are often processed together as Events, then I'd be inclined to implement a single Events table with a type column. Single-table views can be an effective way to separate out the sub-classes and on most db platforms are updatable. If the classes are more different in terms of attributes, then a table for each might be more appropriate. I don't think I like the three-table idea:"has one or none" relationships are seldom necessary in relational design. Anyway, you can always create an Event view as the union of the two tables.

As to Product and Category, if one Category can have many Products, but not vice versa, then the normal relational way to represent this is for the product to contain a category id. Yes, it's coupling, but it's only data coupling, and it's not a mortal sin. The column should probably be indexed, so that it's efficient to retrieve all products for a category. If you're really horrified by the notion then pretend it's a many-to-many relationship and use a separate ProductCategorisation table. It's not that big a deal, although it implies a potential relationship that doesn't really exist and might mislead somone coming to the app in future.

answered Jan 26, 2009 at 23:10

Mike Woodhouse

52.3k ● 12 ● 89 ● 127



In my opinion, these paradigms (the Relational Model and OOP) apply to different domains, making it difficult (and pointless) to try to create a mapping between them.

The Relational Model is about representing *facts* (such as "A is a person"), i.e. intangible things that have the property of being "unique". It doesn't make sense to talk about several "instances" of the same fact - there is just *the* fact.

Object Oriented Programming is a programming paradigm detailing a way to construct computer programs to fulfill certain criteria (re-use, polymorphism, information hiding...). An *object* is typically a metaphor for some tangible thing - a car, an engine, a manager or a person etc. Tangible things are not facts - there may be two distinct objects with identical state without them being the same object (hence the difference between *equals* and == in Java, for example).

Spring and similar tools provide access to relational data programmatically, so that the facts can be represented by objects in the program. This does not mean that OOP and the Relational Model are the same, or should be confused with eachother. Use the Realational Model to design databases (collections of facts) and OOP to design computer programs.

TL;DR version (Object-Relational impedance mismatch distilled):

Facts = the recipe on your fridge. Objects = the content of your fridge.

Share Improve this answer

Follow

answered Jan 26, 2009 at 23:12

Eek

1,060 ● 8 ● 7

Everything Eek has written here is true, but I will also point out that certain databases such as PostGre allow the definition of custom types, letting you store objects as the "nouns" in these facts. – Breton Jan 26, 2009 at 23:18

Facts may not have instances, but tables do. – Superbest Dec 16, 2014 at 20:43

4

Frameworks such as

1. Hibernate http://www.hibernate.org/
2. JPA http://java.sun.com/developer/technicalArticles/J2EE/jpa/

can help you to smoothly solve this problem of inheritance. e.g. http://www.java-tips.org/java-ee-tips/enterprise-java-beans/inheritance-and-the-java-persistenc.html

Share Improve this answer

Follow

answered Jan 26, 2009 at 22:16

Pierre

35.2k ● 32 ● 117 ● 196

I also got to understand database design, SQL, and particularly the data centered world view before tackling the object oriented approach. The object-relational-impedance-mismatch still baffles me.

The closest thing I've found to getting a handle on it is this: looking at objects not from an object oriented progamming perspective, or even from an object oriented design perspective but from an object oriented analysis perspective. The best book on OOA that I got was written in the early 90s by Peter Coad.

On the database side, the best model to compare with OOA is not the relational model of data, but the Entity-Relationship (ER) model. An ER model is not really relational, and it doesn't specify the logical design. Many relational apologists think that is ER's weakness, but it is actually its strength. ER is best used not for database design but for requirements analysis of a database, otherwise known as data analysis.

ER data analysis and OOA are surprisingly compatible with each other. ER, in turn is fairly compatible with relational data modeling and hence to SQL database design. OOA is, of course, compatible with OOD and hence to OOP.

This may seem like the long way around. But if you keep things abstract enough, you won't waste too much time on the analysis models, and you'll find it surprisingly easy to overcome the impedance mismatch.

The biggest thing to get over in terms of learning database design is this: data linkages like the foreign key to primary key linkage you objected to in your question are not horrible at all. They are the essence of tying related data together.

There is a phenomenon in pre database and pre object oriented systems called the ripple effect. The ripple effect is where a seemingly trivial change to a large system ends up causing consequent required changes all over the entire system.

OOP contains the ripple effect primarily through encapsulation and information hiding.

Relational data modeling overcomes the ripple effect primarily through physical data independence and logical data independence.

On the surface, these two seem like fundamentally contradictory modes of thinking. Eventually, you'll learn how to use both of them to good advantage.

Share Improve this answer

Follow

answered Jan 27, 2009 at 4:54

Walter Mitty

18.9k ● 2 ● 31 ● 59

1

My guess off the top of my head:

On the topic of inheritance I would suggest having 3 tables: Event, ShiftEvent and StaffEvent. Event has the

common data elements kind of like how it was originally defined.

The last one can go the other way, I think. You could have a table with category ID and product ID with no other columns where for a given category ID this returns the products but the product may not need to get the category as part of how it describes itself.

Share Improve this answer Follow

answered Jan 26, 2009 at 22:19

community wiki
JB King

0

The big question: how can you get your head around it? It just takes practice. You try implementing a database design, run into problems with your design, you refactor and remember for next time what worked and what didn't.

To answer your specific questions... this is a little bit of opinion thrown in, as in "how I would do it", not taking into account performance needs and such. I always start fully normalized and go from there based on real-world testing:

```
Table Event
EventID
Title
StartDateTime
EndDateTime
```

```
Table ShiftEvent
ShiftEventID
EventID
ShiftSpecificProperty1

...

Table Product
ProductID
Name

Table Category
CategoryID
Name

Table CategoryProduct
CategoryID
ProductID
```

Also reiterating what Pierre said - an ORM tool like Hibernate makes dealing with the friction between relational structures and OO structures much nicer.

Share Improve this answer

Follow

answered Jan 26, 2009 at 22:23

Rex M

144k ● 34 ● 291 ● 315

0

There are several possibilities in order to map an inheritance tree to a relational model. NHibernate for instance supports the 'table per class hierarchy', table per subclass and table per concrete class strategies: http://www.hibernate.org/hib_docs/nhibernate/html/inheritance.html

For your second question: You can create a 1:n relation in your DB, where the Products table has offcourse a foreign key to the Categories table. However, this does not mean that your Product Class needs to have a reference to the Category instance to which it belongs to. You can create a Category class, which contains a set or list of products, and you can create a product class, which has no notion of the Category to which it belongs. Again, you can easy do this using (N)Hibernate; http://www.hibernate.org/hib_docs/reference/en/html/collections.html

Share Improve this answer

Follow

answered Jan 26, 2009 at 22:23

Frederik Gheysels

56.9k ● 11 ● 102 ● 155

0

Sounds like you are discovering the Object-Relational Impedance Mismatch.

Share Improve this answer

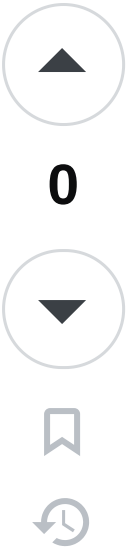
Follow

answered Jan 26, 2009 at 22:31

moffdub

5,314 ● 2 ● 37 ● 32

2 And here are the doctors to describe your illness once again! – Dan Rosenstark Jan 26, 2009 at 23:06



> The products shouldn't even know that the categories exist, much less have a data field containing a category ID!

I disagree here, I would think that instead of supplying a category id you let your orm do it for you. Then in code you would have something like (borrowing from NHib's and Castle's ActiveRecord):

```
class Category
  [HasMany]
  IList<Product> Products {get;set;}
```

...

```
class Product
  [BelongsTo]
  Category ParentCategory {get;set;}
```

Then if you wanted to see what category the product you are in you'd just do something simple like:

```
Product.ParentCategory
```

I think you can setup the orm's differently, but either way for the inheritence question, I ask...why do you care? Either go about it with objects and forget about the database or do it a different way. Might seem silly, but unless you really really can't have a bunch of tables, or

don't want a single table for some reason, why would you care about the database? For instance, I have the same setup with a few inheriting objects, and I just go about my business. I haven't looked at the actual database yet as it doesn't concern me. The underlying SQL is what is concerning me, and the correct data coming back.

If you have to care about the database then you're going to need to either modify your objects or come up with a custom way of doing things.

Share Improve this answer

Follow

answered Jan 26, 2009 at 22:32

rball

6,955 ● 7 ● 50 ● 78

> "The underlying SQL is what is concerning me, and the correct data coming back" What do you mean by this? Are you looking at the SQL generated by the ORM? Isn't it generated automatically? – RussellH Jan 26, 2009 at 22:59

0

I guess a bit of pragmatism would be good here. Mappings between objects and tables always have a bit of strangeness here and there. Here's what I do:

I use Ibatis to talk to my database (Java to Oracle). Whenever I have an inheretance structure where I want a subclass to be stored in the database, I use a "discriminator". This is a trick where you have one table for all the Classes (Types), and have all fields which you could possibly want to store. There is one extra column in

the table, containing a string which is used by Ibatis to see which type of object it needs to return.

It looks funny in the database, and sometimes can get you into trouble with relations to fields which are not in all Classes, but 80% of the time this is a good solution.

Regarding your relation between category and product, I would add a categoryId column to the product, because that would make life really easy, both SQL wise and Mapping wise. If you're really stuck on doing the "theoretically correct thing", you can consider an extra table which has only 2 colums, connecting the Categories and their products. It will work, but generally this construction is only used when you need many-to-many relations.

Try to keep it as simple as possible. Having a "academic solution" is nice, but generally means a bit of overkill and is harder to refactor because it is too abstract (like hiding the relations between Category and Product).

I hope this helps.

answered Jan 26, 2009 at 22:51

Rolf

7,268 ● 6 ● 42 ● 57