Community

yegor256.com

121 Comments • Created 6 months ago

ORM Is an Offensive Anti-Pattern

TL;DR ORM is a terrible anti-pattern that violates all principles of object-oriented programming, tearing objects apart and turning them into dumb and passive data bags. There is no excuse for ORM existence in any application, be it a small…

(yegor256.com)

121 Comments

Recommend 4 Share Sort by Newest

Join the discussion…

Rao • 7 days ago

I fail to see the argument here. Many of the "ORM problems" you've mentioned here such as SQL not being hidden, dealing with two things, et al. do not exist in other frameworks. Entity Framework is a prime example. Using it is as simple as Db.Posts.Where(x => x.Title == "some title").ToList() or Db.Posts.Insert(new Post { Date = DateTime.Now, Title = "Hello" }). No messing with SQL, transactions handled automatically, and the time saved is astounding. If you change your objects (add/remove variables, refactor, etc), the changes to your database structure and/or rows are automatically handled (if automatic migrations are enabled). This enables you, the developer, to just write your interfaces and let the ORM handle the trivialities in the background. (Of course, you have full access to the SQL layer if you need it.) From an ease-of-use standpoint, Entity Framework is top-notch. Performance wise, extremely fast in our production environments.

Reply • Share ›

Alex Urbano • 16 days ago

This post should be called "Hibernate is offensive and ugly (and java for that matter)" as that is what the contents are about, just a bad implementation of ORM, does not mean the whole ORM patter is wrong, or at least your post does not expose it. another option is I failed to see your point againts ORM.

2 • Reply • Share ›

John Fielder • 3 months ago

Great article, might use these concepts for my thesis. Just a question though, what are your thoughts on the Eloquent ORM provided by Laravel? There, Models extend an Eloquent object, and they know how to handle all the persistency themselves. For example, retrieval of Posts would be done by Post::all(). Conditions can also be easily imposed, by using Post::where('date', '>', 'somedate')->get(). Inserting a new post would easily be done by saying new Post(['date' => new DateTime(), 'message' => 'new message'])->save(). I think it is one of the most easy to use and elegant solution for an ORM (it also saves a lot of time by not having to write SQL except for more complex queries). Everything is generalized inside the Eloquent base class, and whatever needs to be specifically implemented (like searching for a field) can be implemented using scopes.
(BTW, those static calls are not static methods in the models themselves. Laravel makes extensive use of the Facade pattern, thus not hindering any testability problems).
I also like your solution, however it seems that a lot of boilerplate code has to be written for every single model. How would you overcome that problem?

• Reply • Share ›

Yegor Bugayenko author John Fielder • 3 months ago

Yes, this is one of the approaches - to put all SQL operations into the parent class. I would not call it the most elegant solution, but it's definitely one of the possibilities. I think that the best and the most elegant way would be to create some data manipulation libraries specifically for SQL. I'm not sure how exactly, but that would be the best. Of course, my code above is not as compact as most people used to see their SQL interaction code, but it's the first step in the right direction. If you can, make it shorter. But always remember that everything that happens between the app and SQL storage should stay inside the object. Nobody else should know that the object is talking to the SQL storage.

• Reply • Share ›

Marcos Douglas Santos Yegor Bugayenko • 3 months ago

"...put all SQL operations into the parent class [...] definitely one of the possibilities"
You had said do not use inheritance, only decorate. :)

• Reply • Share ›

bpechaz • 3 months ago

Nice article and innovative idea. Thanks, I leaned many thing and got bunch of ideas.

• Reply • Share ›

Marcos Douglas Santos • 4 months ago

What about queries that do not returns data for only one object, how do you work with them?

them?
Example: if you have some tables like Order, OrderItem, Vendor, User... and you need to do a query to show these data in a grid. So do you create a new object (interface and implementation)? If yes, what the name of this class?
What about the method to return this list, do you elects a class (eg, Oder) and implement a new method? If yes, I'm think about your "rule" that say a class needs to have max of five methods... if you will have many queries, you need to have many others classes? What about the names of these classes?

• Reply • Share ›

Yegor Bugayenko author Marcos Douglas Santos • 3 months ago
Good question. I'm planning to write a post about it soon. In a nutshell, instead of using "getters" that return data, use "printers" that can print your data on demand. Here is a getter:

```
public class Table {
  Collection<order> orders() {
    // it's a getter
  }
}
```

This is a printer:

```
public class Table {
  void printOrdersInto(Target target) {
    // it's a printer
  }
}
public interface Target {
  void print(int amount, String desc, String customer, ...);
}
```

See the point?

• Reply • Share ›

Marcos Douglas Santos Yegor Bugayenko • 3 months ago
If I understood right:
You elects a class and implements a new method on it...

Well I think doesn't matter if this method have some argument (eg Target) or a long name to show me what it will do. The problem is that this class will have a lot of "arguments" or "long method names" because it is common for a real application has a lot of grids and reports to show data.
Your example, is OK. But I'm thinking in real comercial applications; lot of grids and reports.

I will waiting your article. Thank you.

• Reply • Share ›

Marcos Douglas Santos Marcos Douglas Santos • 4 months ago

Marcos Douglas Santos › Marcos Douglas Santos • 4 months ago

@Yegor Bugayenko did you understand my questions? Thanks.

• Reply • Share ›

Oleg Majewski • 4 months ago

Hello, one more time ;-)

Assuming you have a mysql database, with three tables: post, article, balance.

What you suggest is to access the data in that tables by:

```
@Immutable
final class PgPost implements Post {
  private final Source dbase;
  private final int number;
  public PgPost(DataSource data, int id) {
    this.dbase = data;
    this.number = id;
  }
  public int id() {
    return this.number;
  }
  public String title() {
    return new JdbcSession(this.dbase)
      .sql("SELECT title FROM post WHERE id = ?")
      .set(this.number)
      .select(new SingleOutcome<string>(String.class));
  }
}



@Immutable
final class PgArticle implements Article {
  private final Source dbase;
  private final int number;
  public PgPost(DataSource data, int id) {
    this.dbase = data;
    this.number = id;
  }
  public int id() {
    return this.number;
  }
  public Date date() {
    return new JdbcSession(this.dbase)
      .sql("SELECT date FROM article WHERE id = ?")
      .set(this.number)
      .select(new SingleOutcome<utc>(Utc.class));
  }
  public String title() {
    return new JdbcSession(this.dbase)
      .sql("SELECT title FROM article WHERE id = ?")
      .set(this.number)
      .select(new SingleOutcome<string>(String.class));
  }
}



@Immutable
final class PgBalance implements Balance {
  private final Source dbase;
  private final int number;
  public PgPost(DataSource data, int id) {
    this.dbase = data;
    this.number = id;
  }
  public int id() {
    return this.number;
  }
  public Date date() {
    return new JdbcSession(this.dbase)
      .sql("SELECT date FROM balance WHERE id = ?")
      .set(this.number)
      .select(new SingleOutcome<utc>(Utc.class));
  }
  public String title() {
    return new JdbcSession(this.dbase)
      .sql("SELECT title FROM balance WHERE id = ?")
      .set(this.number)
      .select(new SingleOutcome<string>(String.class));
  }
}
```

After two weeks your project grows and you get 10 more tables. Next you get more columns on that table. You suggest to create for each that new table a new Class and new methods for each new column. Why?

After some time your data set grows, and you cannot fetch all Articles at once, you need pagination, how do you add this logic to all your classes?

Besides this, if you look carefully at that three classes above, what is the difference between them? The difference is only: the table and the column names, everything else is exactly the same. What is wrong with this approach?

DRY

Now lets take a look on what you can do with a project like spring data jpa:

```
public interface BalanceRepository extends CrudRepository<balance, long=""> { // no clue
}
```

What does it do for you? A lot!
few examples:

```
@Inject BalanceRepository repo; // spring data generates it for you
List<balance> all = repo.findAll();
need pagination? Not a problem:
Page<balance> paginated = repo.findAll(pageable);
how about sorting?
List<balance> all = balance.findAll(sort); // note, the sorting is done on db level, not
repo.save(balance)
repo.save(balances)
repo.delete(...)
repo.deleteAllInBatch()
```

ok, but what if you need to:
select * from balance where name='foo'

easy and without violating DRY:

```
public interface BalanceRepository extends CrudRepository<balance, long=""> {

     List<balance> findByName(String name);
}
List<balance> balances = repo.findByName("foo");
```

What's wrong with that? Why would you write that [1..] classes by hand and call it art instead of using a library?

• Reply • Share ›

Yegor Bugayenko author Oleg Majewski • 4 months ago

I'm not against optimization. I'm against the way optimization is done in JPA or many other ORMs. If you see that your classes are doing similar things and there is a possibility to make some of that code common, that's great. Create another class, which will be reused by that original classes. But always make sure that database operations are encapsulated inside your objects. Nobody outside of an object should know that the object is working with a database. This information is private and confidential. That's the principle being offensively violated by JPA. (btw, would be great if you format your question, as explained here: https://help.disqus.com/custom...

• Reply • Share ›

Oleg Majewski Yegor Bugayenko • 4 months ago

> If you see that your classes are doing similar things and there is a possibility to make some of that code common, that's great. Create

> another class, which will be reused by that original classes.

No. Somebody else much smarter then me (actually more then one) already did it. No point to reinvent the wheel.

> would be great if you format your question, as explained here: https://help.disqus.com/custom...

thx, done. Would be great if there would be a link to the markup wiki somewhere, like it's the case on github and/or a preview button.

• Reply • Share ›

Yegor Bugayenko author ➦ Oleg Majewski • 4 months ago

Absolutely, don't reinvent the wheel, use libraries for that. Use jOOQ or jcabi-jdbc or SpringTemplate or anything else. Anything that allows you to retrieve the data from SQL and manipulate with them **inside** the object. JPA treats an object as a data bag and manipulates its data **outside** of it. That's what's wrong.

• Reply • Share ›

Oleg Majewski ➦ Yegor Bugayenko • 4 months ago

JPA is not just about manipulating data. It is much more about the @Entity classes and writing type safe criteria queries, which are great!
In the normal case if you do anything with jpa, you do it inside a DAO and that is what you are talking about. A DAO manipulates data INSIDE and wraps all db operations perfectly. The question is only what do you put inside that dao, a clean and safe JPA query or plain sql like you suggest. So if I can choose, I take the first option, and there is absolutely nothing bad about it.

PS: well in spring world that DAO's are called repositories

• Reply • Share ›

Oleg Majewski • 4 months ago

Yegor, did you hear about spring data project?
What's the point of implementing of all that standard CRUD operations by hand again and again, if they can be easily generated for you?
see here: https://spring.io/guides/gs/ac...

• Reply • Share ›

Yegor Bugayenko author ➦ Oleg Majewski • 4 months ago

What's the point of drawing these pictures again and again by hand when you can easily make a photo, using a camera? :)

1 • Reply • Share ›

Oleg Majewski ➦ Yegor Bugayenko • 4 months ago

the point is the cpu and io does not understand art and people don't want to draw the same picture 1,000 times, that's why they built a xerox draw the same picture 1.000 times, that's why they built a xerox

• Reply • Share ›

Yegor Bugayenko author Oleg Majewski • 4 months ago

If the picture you're drawing is so boring that you don't like to draw it again and again, there is something wrong with the picture or you, as an artist. Try to find another profession. Maybe a Xerox copy machine operator? :)

1 • Reply • Share ›

Oleg Majewski Yegor Bugayenko • 4 months ago

It is not boring, if you draw one, also not if you draw ten and each one is different. Once you found how one that perfect picture looks like you start thinking about building a factory, as soon you need several thousands of the same picture, cause people are not machines. In the industrie people build factories, in the software industriy people build libraries. So you really mean I should change profession because I learned DRY and how to use software not written by myself? Common...
PS: really love a lot from your articles, but here you are wrong ;)

• Reply • Share ›

Dave • 4 months ago

Hi,

This is Dave, :)
I believe in kind of related idea you talked from some time but not entirely. In one enterprise application architecture I used below flow.
Its limiting database related query/ code only in small part. Most of the part/ application work on OOP base. Its easy to change/ modify database and UI part. And in way its better because there are times when you can explain some of the fields or property of database in real world or you don't need at that particular time/ phase of application.

I would like to know how many your points/issue idea fulfilling / solving ?

Can you please also give me answer about architecture ?

Simple flow.

Database <-----------> Classes(Entity) for getting/sending data <----------------------> Classes(OOP concept) for doing operation like OOP and behive like that. <--------------------> Presentation layer <-----------------> UI

1 • Reply • Share ›

Yegor Bugayenko author Dave • 4 months ago

Your architecture, if I understood it correctly, looks good. Presentation layer manipulates with business objects, which manipulate with persistence objects, which manage the database through SQL. Looks perfect to me :)

• Reply • Share ›

John Ohno • 4 months ago

This article doesn't go far enough. Objects should maintain their own representation, and be capable of updating that representation transparently, regardless of whether that representation is in an RDBMS, a flat file, or some structured and application-specific representation. Having a factory that translates objects between various representations is an anti-pattern, particularly when having such a factory is a distraction from the primary concern of the project.

This isn't even strictly an OO purity issue. ORMs are a typical example of Java syndrome -- creating a giant complex group of objects in order to avoid writing actually object-oriented code -- and while I'm not of the opinion that object orientation is ideal in all situations, in a language like Java that syntactically enforces its warped idea of object orientation on programmers, trying to write code in an imperative manner by building an ORM or an object factory or transformer classes simply isn't worthwhile. If you're just starting out, don't bother with an ORM and just write your code in an OO way in the first place, if you're stuck with Java (and part of that is making objects capable of maintaining their own representation, even if it means giving them all a copy of some object that lets them handle it in a backend-agnostic way). An object is nothing but a key-value store anyhow, so encapsulation is almost always easier than adding external structures to explicitly avoid encapsulation.

1 • Reply • Share ›

Yegor Bugayenko author John Ohno • 4 months ago

I agree with everything, except this: *An object is nothing but a key-value store anyhow*. This sounds offensive :) Check this article: http://www.yegor256.com/2014/1...

• Reply • Share ›

John Ohno Yegor Bugayenko • 4 months ago

Let me rephrase. On an implementation level, at runtime, an object is nothing but a key-value store.

Aesthetic and theological concerns about the design of objects have nothing to do with the code that you write when you are implementing the concept of an object within a general-purpose object system, and only come about when you start thinking about inheritance or class/object distinction.

Conceptually, an object can be many things; none of this really matters when you look at the implementation of the Object data type in any OO system -- which is functionally the same between Java, Javascript, and Smalltalk.

• Reply • Share ›

TrylO • 4 months ago

You know I was sceptical about your article, maybe I wasn't able to get the whole point, moreover I think the title is a little bit misleading. After some research and read some other stuff about DDD, I started to realize the sense of all of this and I like it a lot, but the article missed a spot: the right instruments to reach such goal. It's good to make a show of concept (or a sort of that), but in these days I think we can also provide a demonstration of what we are trying to show with a more practical examples. For instance I saw Lukas Eder from JooQ (nice guy, I often read his articles), do you think it's possible to apply your concepts with a framework like JooQ (or another one like QueryDSL)?

• Reply • Share ›

Yegor Bugayenko author TryIO • 4 months ago

Yes, jOOQ is a good instrument to implement what I suggest above. Actually Lukas was the first one who commented on this article (see below). Besides jOOQ you can use Ollin, JDBI, Spring JdbcTemplate and others. There are plenty of tools that allow you to connect to the JDBC data source and fetch the data. You can even use JDBC directly. It doesn't matter how you do this. What matters is that it all happens **inside the object**.

• Reply • Share ›

valenterry • 5 months ago

Hey Yegor,

do you know of or even worked with scalas slick? It is not ORM and looks a bit similar to your approach. I would love to hear your thoughts of slick.
If you don't know slick, you might be interested to read the following article which is related to the topic of this blogpost: http://slick.typesafe.com/doc/...

• Reply • Share ›

Yegor Bugayenko author valenterry • 5 months ago

According to their documentation (http://slick.typesafe.com/doc/...), it is very similar to a traditional ORM, where objects are filled with data by some engine...

• Reply • Share ›

Roman • 5 months ago

Could I ask couple of questions.

1) Are fields in the interface static and final?
@Immutable
interface Post {
int id();
Date date();
String title();
}

2) Does PgPost has one argument constructor? To create instance like
@Override
public Post map(final ResultSet rset) {
return new PgPost(rset.getInteger(1));
}

• Reply • Share ›

Yegor Bugayenko author Roman • 5 months ago

I didn't understand the first question. You're asking about fields, but showing methods. Interfaces can't have static or final methods.

PgPost must encapsulate an instance of DataSource, in order to be able to connect to the database and fetch the data.

• Reply • Share ›

Roman Yegor Bugayenko • 5 months ago

Sorry, for the first question, its my mistake. Regarding the second, PgPost has only one constructor (two argument)

public PgPost(DataSource data, int id) {
this.dbase = data;
this.number = id;
}

Is it possible to create new PgPost(rset.getInteger(1)); passing only one argument?

• Reply • Share ›

Yegor Bugayenko author Roman • 5 months ago

No way, they are both mandatory. The instance of PgPost must know where to get the data ("data source") and what is the coordinate of the data ("id")

• Reply • Share ›

Michael Pasacrita • 5 months ago

This is totally a waste of time. If you want to reduce the amount of code you write with Hibernate, use generics and type casting in your service-level code.

All of your domain model objects should extend a base class, or in my case I used two, one for entities, and one for look-up/"type"/enum tables. Then, you only need ONE service class with methods for "list all", "get by ID", "save" and "delete" operations.

• Reply • Share ›

Michael Pasacrita Michael Pasacrita • 5 months ago

P.S. I also used Spring's Hibernate Transaction Manager Factory to automatically open transactions, and either commit or roll them back with AOP.

Which is also an OOP anti-pattern right? Tell that to the business that's sponsoring your development. "It's going to take a lot longer, and have a ton more code to maintain, but it'll be purely object-oriented."

• Reply • Share ›

llorenç Chiner • 5 months ago

Neat implementation and explanation but anyway no way to retrieve for example a Collection of Employees in a single statement as we can do with Hibernate without the need to assign column by column as the examples here exposed

• Reply • Share ›

Hans-Peter Störr • 5 months ago

Thanks for the interesting links to ORM criticisms! Two points about your arguments:
1. In a decent ORM application you don't smear HQL all over the place, but just use it (and the SessionFactory, for that matter) only within the data access objects - which corresponds to your Posts object. So, no difference there. Quite easily mockable, too.
2. I do share your worry that the common way ORMs are used is mostly prodecural programming with anemic "objects". However, that's not mandatory: I believe you can also have the ORM do its work on private fields and/or protected methods of the object, and let the project provide only a rich object oriented interface without any kind of (public) getters/setters.

1 | • Reply • Share ›

Adithya R • 5 months ago

I have the same sentiment of the author about ORM.

I believe that the underlying problem being the design of systems where data is architecture with a different design - ER design and the application is designed with OOP in mind and hence we do have an ORM as a bridge. the solution you have given is good if we dont have a way to change the DB design, but if we do, I believe a good solution is to use NoSQL (with whatever quirks it has today - but we do have many options). In NoSQL systems the data design also follows a similar architecture as that of the oop application itself and hence the connection between the systems is much direct and simpler.

| • Reply • Share ›

Eumaho Heraldo • 5 months ago

Your approach is similar with a Domain Driven Design approach.

your PgPost class is actually a DAO . I think it is not practical to use a DAO everywhere in an application.
DTOs are there for a reason.
POJOs also , for the same reason.
I think in J2EE everything comes from the "layering" approach.

I don't like ORMs but couldn't find anything better.Yet. :)
I also don't like the idea of J2EE layers but again, nothing better yet.

Fowler is correct. ORMs solve 80% of the problems in a manageable way.

And you are also correct. A POJO is breaking the OOP definition.

NoSQL might be a solution.

| • Reply • Share ›

incognito • 5 months ago

Well, any object oriented code interacting with relational database is ORM, whether it's third-party lib or part of your own codebase. Maybe you could say that JPA is anti-pattern, which I'm totally not agree with but it would make more sense for the discussion.

| • Reply • Share ›

Brandon • 5 months ago

I get ride of JPA, Hibernate by develiping my own web application template:

Thumbnail

^ | v • Reply • Share ›

Anonymous Long Tooth • 5 months ago

Oh boy. Yegor appreciate your passion. Sounds like the crux of the issue is with how the session factory for instance, exposes the ORM internals. There are ways to hide the implementation for your callers via facades, delegates, strategies etc .. For most garden variety applications ORM works well - high performance / low latency is a different matter

^ | v • Reply • Share ›

Invisible Arrow • 5 months ago

I was thinking about the strategies used for mapping relationships between objects.
For example, let's say there are comments for a particular post.
Would the `Post` interface expose a behaviors to add/remove/view comments?

In terms of feature evolution, let's say, to begin with we didn't allow comments on posts.
At a later date, it was decided that comments are allowed for certain posts and not others.

Using the inheritance approach, we would extend the `Post` interface to support comments, something like:

```
interface PostWithComments extends Post {
    List<comment> comments();
    PostWithComments addComment(Comment comment);
    PostWithComments addComment(Comment comment);
    PostWithComments removeComment(Comment comment);
}
```

The problem I see with the above inheritance approach is that with every new feature added to a post, the extended interfaces start to have too many responsibilities.

One other approach using composition would look like the following:

```
interface PostWithComments {
    // Refers to the normal post
    Post post();

    // Additional behavior for comments
    List<comment> comments();
    PostWithComments addComment(Comment comment);
    PostWithComments removeComment(Comment comment);
}
```

The problem I see with the compositional approach is that in order to access any behavior on the base `Post`, we would do something like `postWithComments.post().id()`
With more features added later, the chain would grow to access the base post's behavior.

Is there something fundamentally wrong with the two design approaches mentioned, and a better way out?

2 • Reply • Share ›

Yegor Bugayenko author → Invisible Arrow • 5 months ago

Good question, even though not directly related to the subject of this article :) I don't have anything against long constructs like `post.comments().add(comment)`. There is a famous Law of Demeter (http://en.wikipedia.org/wiki/L... that discourages/prohibits such constructs. I see more disadvantages in it than advantages. I think that a good object has less than five (!) public methods. If you need to have more, create a sub-object and return it in a method.

Speaking about your example, I think that `post()` method is a wrong workaround. Inheritance is a natural mechanism, if it naturally mirrors real-life entities. If you have a post and a post with comments, design them as two classes. Just like you did. If you worry about the amount of methods which is growing, think about your design and its cleanness. But don't replace inheritance by decoration only because of Java optimization. Java (or any other language) is just an instrument to model the reality. The reality is the king, not the instrument.

1 • Reply • Share ›

Invisible Arrow → Yegor Bugayenko • 5 months ago

Makes sense. I guess the biggest takeaway here is to properly model based on what the object really represents in the real world, rather than getting carried away with language constructs. Thanks again for your inputs :)

• Reply • Share ›

Konstantin Mikheev • 5 months ago

Hmm... Nice idea. I completely agree on ORM to be an anti-pattern. My last project clearly proves this (I think about data more than about anything else, which is a bad sign. :))) For me the main problem is having a few instances of the same object the same time in different parts of an application.

However, I would not go and create one zillion of handwritten SQL queries. I also do not like the idea of having 100-args constructors.

I think of using ORM-based factory, but createOrUpdate(), delete(), and other data changes should go without direct ORM call.

• Reply • Share ›

GordonJP • 5 months ago

So... you seriously suggest, we all better start HANDCRAFTING and maintaining all these brain-dead SELECT/INSERT/UPDATE/DELETE stmts?

Come on --- I cannot even begin to tell you how much that SUCKS when you move beyond a trivial demo-app.

Where you get most value from ORM, in my opinion, is when you start getting more complex/deeper graphs of objects and dependendent parent/child objects.

When you say HQL is just "a dialect" of SQL, you're totally missing the point.
An ORM query like "from T" will automatically bring you a list of object-GRAPHS, with ALL DEPENDENT objects and attributes, as specified by your annotations or xml.

Try to handcraft that with SQL when you have object-graphs, 10 or 20+ levels deep (which is not at all uncommon in many enterprise applications), and then maintaining all that SQL throughout the application life-cycle, and I'm sure you will start seeing the light! :D

On the other hand, nothing prevents you from also adding business-logic methods in the same java-objects you retrieve/persist with Hibernate --- nothing forces them to be *only* "dumb data bags".

And, finally... if you are only implementing something very trivial, with 3 db-tables or so, I totally agree -- no reason to bother with ORM.

5 • Reply • Share ›

JavaDev ↪ GordonJP • 5 months ago

That's very similar to what I was thinking as well, this is sort of looking at the data transfer object side of a standard ORM setup alone without the complimentary and (in my experience) ubiquitous data access object side, which encapsulates the storage/retrieval mechanism.

I think the base set of objects for the discussion is incomplete, although I may be misunderstanding.

• Reply • Share ›

Yegor Bugayenko author ↪ GordonJP • 5 months ago

Yegor Bugayenko author ➦ GordonJP • 5 months ago

Yes, I seriously suggest to handcraft SQL inside SQL-speaking objects. Here is a practical example: https://github.com/aintshy/hub... Look at the classes in this package. They all are talking to PostgreSQL. I can't say this application is big, just a few tables in the DBMS. I do realize that when an application gets bigger, the amount of objects grows as well, and the number of tables too. However, I don't see why this growth is a threat to complexity. Each object perfectly encapsulates all SQL manipulations and stays cohesive.

The problem with ORM is that is breaks the encapsulation concept. It turns objects into transparent data bags, where ORM engine injects data and our application retrieves data. This is not what object oriented programming is about. The ORM idea encourages programmers to betray the principles of OOP - that's what is wrong with ORM. It's not about performance, complexity, or maintainability. It's about a fundamental and fatal **damage to our mentality**.

3 ^ | ˅ • Reply • Share ›

JavaDev ➦ Yegor Bugayenko • 5 months ago

And I think that I would argue that what you're doing is basically wrapping you data access object into your data transfer object, which I believe violates separation of concerns and also bleeds functionality into layers that don't require it.

I believe that this type of setup would lead some clients, say an Android client that was communicating with the server, to be forced to either import all of you non-locally-working functionality (and required compile time dependencies) into their application, or create a translation layer to get back to simple data transfer objects and create redundant classes to hold and use the data separate from your self storing classes. You could automate this with code generation (like what I've seen with Google Cloud Endpoints), but why do this at all? What are you saving yourself really?

I think that this is a disagreement on which of various OOD principles take precedence in certain situations. To me maintainability and then speed to market takes precedence over theory. I think that what you would create with this for a non trivial application would be extremely expensive to maintain unless is was a trivial case, like an application with 4-5 tables.

Now I would agree that you generally don't see ORM/JPA used wholesale in the wild unless there aren't any real resource constraints (for instance due to low throughput or non-critical business applications) since most ORM make a mess of queries and joins compared what a DBA can come up with (which is RDBMS specific and tested). But JPA allows specific mapping of this also, so it's a bit of a hollow point to me.

^ | ˅ • Reply • Share ›

Yegor Bugayenko author ➦ JavaDev • 5 months ago

I believe that "object thinking" and fundamental principles of OOP are much more important than performance, time to market, or even maintainability (which can only happen if uneducated programmers are going to maintain it). The key principle of OOP is the encapsulation. Each object should be fully responsible for all its internals. Everything that is happening inside it should be fully under its control. ORM/JPA violates this principle.

• Reply • Share ›

AnotherJavaDev Yegor Bugayenko • 5 months ago

"I believe that "object thinking" and fundamental principles of OOP are much more important than performance, time to market, or even maintainability"...you sound like a fundamentalist! And as we see in other areas where fundamentalism is perpetuated, pragmatism, tolerance in other to allow for a greater/different good/benefit is always sacrificed on the alter of having to follow an idea to the letter!

2 • Reply • Share ›

Yegor Bugayenko author AnotherJavaDev • 5 months ago

Western civilization flourishes not because of tolerance and pragmatism, but because of **strict laws**. The same is true in software development. Where you don't have rules and strict principles, you will get chaos and disorder.

2 • Reply • Share ›

AnotherJavaDev Yegor Bugayenko • 5 months ago

Societies that have strict laws that prevents women from driving, from voting, from participating in government. Societies that have strict laws regarding which religious expressions to pursue must also be valid examples of flourishing civilizations because well, they have got that thing called *strict laws*

I hope you get to see the flawed repercussions any form of fundamentalism: be it in governance, religion or _even software development_ could bring about!

1 • Reply • Share ›

Martin • 5 months ago

I love how you just discovered the DAO pattern and implemented your own ORM. Also, you fail to mention that EJBs (over a decade ago) implemented this EXACT pattern, in the shape of home interfaces and entity EJBs.

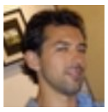
Congratulations on your freshly reinvented wheel!

24 • Reply • Share ›

Damian Tylczyński Martin • 5 months ago

Yeah, that's awesome! Amazing article! Congratulations to the author and happy New Year!

• Reply • Share ›


Álvaro Hernández Tortosa • 6 months ago

Hi!

While I certainly agree with the main concept of the post, and most of the techniques employed, I have certain doubts about your suggested pattern and specially its implementation. I value the design of objets that are Immutable, but I find the objects you named as such are nothing but Immutable. A Post object, for example, that queries the database can't never be Immutable. Any invariant designed on top of it may fail miserably if the external datasource is concurrently changed. ConstPost is immutable, but not Post.

Indeed, scaling this approach to a more general design pattern seems hard. Too many decorators and implementations will result in practical cases. That's why I preper to favor a pattern based on a combination of business objects (normal objects, designed for the application) and DAOs (call them controllers if you like). The latters would be the only ones serving as entry point, the business objects may be immutable. That coupled with the use of a good tool for interacting with the database (like the above mentioned jOOQ) is a receipe for success. And a good alternative to the fundamentally big problems that ORMs have.

• Reply • Share ›

Yegor Bugayenko author → Álvaro Hernández Tortosa • 5 months ago

I tried to answer your question about immutability in this post:
http://www.yegor256.com/2014/1...

• Reply • Share ›

Mathys • 6 months ago

What about business logic? Do you also use a decorator for that?

• Reply • Share ›

Yegor Bugayenko author → Mathys • 6 months ago

Yes, absolutely. If you want, for example, to create a post the doesn't accept short titles, you decorate an existing one:

```
class SmartPost implements Post {
  private final Post origin;
  SmartPost(Post post) {
    this.origin = post;
  }
  void rename(String title) {
    if (title.length() < 25) {
      throw new IllegalArgumentException("title is very short");
    }
    this.origin.rename(title);
  }
}
```

Then, you instantiate your post:

```
Post post = new SmartPost(new PgPost(source, 123));
```

```
post.update("very short"); // exception expected
```

• Reply • Share ›

Brandon Yegor Bugayenko • 5 months ago

I prefer property setup for event or object, and when service is initiated, read it into memory map:

One example:

################################
!!!!!!! Block Start !!!!!!!
################################

column_name = user_id
display_name_id = Your_ID

sequence_num = 1

column_max_length = db defined
# column_max_length = 5

default_values = blank
display_component = text

column_forbidden_char = NA
illegal_char_error_msg_id = NA

blank_error_msg_id = user_id_blank
excess_length_error_msg_id = user_id_too_long

################################
!!!!!!! Block End !!!!!!!
################################

• Reply • Share ›

Marcos Douglas Santos Yegor Bugayenko • 6 months ago

Do you not use inheritance? Maybe only from abstract classes?

• Reply • Share ›

Yegor Bugayenko author Marcos Douglas Santos • 5 months ago

Inheritance is much less flexible instrument than decorating by encapsulation. So, I prefer to stay away from it as much as it's possible. Only when it's inevitable, use inheritance.

• Reply • Share ›

Marcos Douglas Santos Yegor Bugayenko • 5 months ago

Less flexible because you does not create methods that are not part of the interface that the class implements.
The inheritance also makes writing less. However the encapsulation of the decorating is the "right way" to do considering that some

authors even say that inheritance breaks encapsulation - and it makes sense.

• Reply • Share ›

Invisible Arrow ➦ Marcos Douglas Santos • 5 months ago

In Scala, I've seen decoration done through 'stackable' traits: http://www.artima.com/pins1ed/...
Essentially, they are doing this through multiple inheritance rather than composition and has lesser lines of code, and seems just as flexible and readable. Do you see any catch with this approach compared to decoration through composition?

• Reply • Share ›

tuespetre • 6 months ago

I use this pattern -- more widely referred to as Active Record -- myself, although it is technically a form of object-relational mapping.

It seems to me that the pattern you are really trying to condemn is the repository pattern, and possibly the query object and unit of work patterns.

When you have full control over the deployment of your application, Active Record is a really simple solution for data access, especially when you're very fluent with SQL; however, there are times where you wouldn't want to use it.

If you were shipping an application for someone else to deploy, or releasing it to the public for anyone to use how they please, you wouldn't want to tie anyone to one specific means of data access. You would need to supply an interface so that there could be multiple implementations. That doesn't strictly mean you would need to use dependency injection, for the record.

I'd rather avoid the 'X is an anti pattern' mindset and instead think about when those patterns offer their greatest benefits.

4 • Reply • Share ›

greg ➦ tuespetre • 5 months ago

I can not find points where ActiveRecord (not pattern, but implementation used by Rails) share's issues provided in the article for ORM. Also way Ruby language works makes it hard to generalize - for example (difficult to test argument) there's no problem to mock Post.find method even though for Java or PHP developer it looks like static method. Regarding SQL polution - you can encapsulate it with scopes, and anyway you're free to not use SQL outside of models.

• Reply • Share ›

Yegor Bugayenko author ➦ tuespetre • 6 months ago

Definitely, there could be more layers between an object and SQL database, but all of them should be **inside** the object. ORM is staying outside of the object and this is what makes it a terrible violator of object paradigm. I don't see how ActiveRecord is related to what I explained above. ActiveRecord instance represents a single row in a relational database, while I'm suggesting to forget about that one-on-one mapping a relational database, while I'm suggesting to forget about that one-on-one mapping and think in terms of objects speaking SQL. What will they fetch from SQL tables - it's totally up to them. We, their clients, have no idea about that "mapping".

• Reply • Share ›

tuespetre Yegor Bugayenko • 6 months ago

If you truly had a business reason for supporting multiple database vendors (SQL Server, MySQL, MongoDB, etc.) you would not want to bloat your class with all of that behavior. That behavior has nothing to do with the business logic of your class. That is when you would declare an interface or abstract class and either let your 'Post' class be configured to use a given implementation (also known as the 'strategy' pattern) or declare public constructors on your 'Post' class that enforce the integrity of the business model and let the data access implementations construct instances and subscribe to events on the instances to persist changes (also known as the 'observer' pattern [and in case of creation or deletion, these would be static events.])

After all, violating the Single Responsibility Principle is violating OOP and so forth... really, I think there is a lot of fetishism surrounding paradigms, principles, and patterns, and there are so many valid ways to get things done, using what you need on a case-by-case basis. At some point you just have to not sweat it and say "well, this is what works for me/us, for this particular project."

And as far as how your post relates to the 'active record' pattern, many people have recognized that as being the case, and even Martin Fowler in his book stated that an 'active record' class could map to a view or ad-hoc query instead of a single table -- so he left some wiggle room in his definition, and even though you and I are using this pattern to encapsulate any number of rows instead of just one, it doesn't mean we are really using a new pattern that needs a new name that, to be honest, doesn't really roll off the tongue or consider other scenarios such as classes developed against non-SQL data stores.

1 • Reply • Share ›

tuespetre tuespetre • 6 months ago

Disclaimer: you could wire up static factories underneath an Active Record-style facade to accomplish multi-database support if you really wanted. Would it be a good idea? That depends.

• Reply • Share ›

ENOTTY • 6 months ago

ActiveRecord repeats itself, first as tragedy, second as a farce.

4 • Reply • Share ›

roman • 6 months ago

you're a f-king idiot, all i can say

3 ^ | v • Reply • Share ›

swarchitect • 6 months ago

And for a shorter, higher-level version of my preceding post, ORMs are just a tool like any other software/development tool,and there are good ones and bad ones (I've never liked hibernate/nhibernate btw), and they can be used/misused like any other software/development tool. To write off the *entire* ORM paradigm and any/all implementations of it is, IMHO, a mistake.

^ | v • Reply • Share ›

swarchitect • 6 months ago

So... with your "SQL-speaking objects" it seems to be that you have basically created your own ORM implementation. I would recommend that you invest 20 minutes in watching the youtube video on Entity Framework in the .NET space, in which you can see a very elegant architecture that handles all of the concerns at play when interacting with a database (simple access of the data, transactions, caching, easy extension, dependency injection / ease of mocking, etc etc):

I won't even take the time to try to rebut every statement in this post that I think you have either completely wrong or actually backwards (I lost count about halfway through it). Now, don't get me wrong - this comment of mine is not just a "hater" comment; I absolutely LOVE a lot of your other posts (and find the ideas in "Stop Chatting, Start Coding" particularly interesting). I just think you're way off-base in these sentiments about ORMs. If I had to pick one thing to say, it is that I think in general, as well as yourself, there has been a confusion over the last 10-15 years about the nature of "ORM"s versus "DAL"s and the functions that are provided by each/both. When I utilize the type of architecture seen in the youtube video (which is in fact what I am doing nowadays for .NET application architecture/development, specifically web apps using ASP.NET MVC), I use what is, yes, generally regarded as an "ORM", Entity Framework, more like a DAL that provides the generated code that I need to interface with a database, then just place a simple generic repository over it and only hand-code interface'd service objects when I need them for specific functionality in an application (and at that point, in the service layer, I can also extend the "base" ORM/DAL-generated code to do such things as more complex mapping to real-world objects that may need to persist data to multiple database tables, or the opposite scenario in which data from one database table needs to be broken into multiple service layer objects). IOW the base E/F implementation acts more like a base DAL, with the generic repository and service layer giving one the flexibility to implement any *actual* more complex Object-To-Relational relationships between one's *business objects* (the definitions of which exist in the service layer) and one or more physical database tables. If you watch the youtube video all the way through, you can see just how much work it saves to use the reference architecture it shows, especially within the context of MVC controllers in LOB web applications. And shouldn't that be the bottom line? i.e., how much work using something like this saves and how much quicker/easier it makes producing working software? I can create the entire initial version of my DAL/ORM/generic repository/service layer (service layer = business object layer, or BLL, for the old-school term) within minutes, by pushing a few buttons, and then (IF I need to) tailor it in the service layer for any cases in which there is not a one-to-one correspondence between a business object and a physical database table. At least in the case of an existing database before creating a new application, anyway, which constitutes the vast majority of the work I do, and even when it isn't I choose to do "database-first" development anyway. (Note that the only significant issue I have with the .NET Entity Framework (like for any ORM) is performance, although that has greatly improved with newer versions and a good developer can optimize it perfectly adequately for any OLTP type of business app's data access requirements by using its caching features and other coding techniques; for ETL type of processes basic ADO.NET code should always be used anyway.)

• Reply • Share ›

William • 6 months ago

I think it would help many developers if they were to understand the relational model better.

If you understand the model better then you will begin to appreciate how elegant and effective it is. SQL clouds the issue a great deal, firstly because it doesn't follow the model correctly and secondly because there are many different versions of SQL.

The most effective way of dealing with the different versions of SQL is to code according to the relational model. This is the real common ground between all SQL implementations.

It cannot be said to often that the relational model has nothing to do with persistance or storage. This common misunderstanding is repeated again by Fowler in his article. One of the central principles of the relational model is that the programmer or user only has to deal with the model and does not need to know anything about how the data is represented physically. A relational representation of the data could be in memory or on disk, but as programmer or user you shouldn't need to know about this.

I think it is a terrible shame that so many programmers regard the relational model as something problematic or outdated. It is really one of the most simple, elegant and effective approaches ever thought of in computer science. Once you understand it better you will be able to take full advantage of it in your work and be able to produce more reliable, simpler code.

I've spend years working with various frameworks, with OO and with RDBMSs but it is really only in the last 5-10 years that I have come to appreciate what a truly brilliant idea the relational model is. I would really recommend everyone to have a closer look.

1 ^ | ⌄ • Reply • Share ›

Outrunner64 ➦ William • 6 months ago

Same here. I've tried for years to treat the database simply as a storage that persists my objects. We even wrote our own OR mapper like suggested in this article and ran into all the problems that are solved in the established OR mappers, like

- Inefficent queries (n+1 problem)
- No transaction boundary concept (resulting in the need for nested transactions, *shudder**)
- the need to build an identity map to avoid creating multiple instances of objects representing the same database record
- Massive instantiation of objects when the final output for the user is a table that a simple SQL query could deliver in fractions of a second

and much more problems.

To my excuse we made these errors 10 years ago. Our current architecture distinguishes between read and write models (see Martin Fowlers CQRS). The read models are relational.

1 ^ | ⌄ • Reply • Share ›

swarchitect ➦ William • 6 months ago

I started working with RDBMSs in the 1980's (IBM mainframe DB2) and I couldn't agree more with everything you said.

^ | ⌄ • Reply • Share ›

Outrunner64 • 6 months ago

So you are basically implementing you own OR Mapper. Bad idea; read the article from Martin Fowler again (or for the first time, because it seems you didn't read it all). Your domain models do not follow a basic OO principle, they have too much responsibility (knowing about persistence). Your suggestions will lead to chaotic, unmaintainable applications when they grow beyond triviality. Let me tell you that as an experienced lead developer / architect that made these errors years ago ;-)

1 ^ | ⌄ • Reply • Share ›

gilligan_MH • 6 months ago

I lost count of all the SOLID principles you are breaking with your object design.

4 ^ | ⌄ • Reply • Share ›

disqus_A97A9EPYY2 • 6 months ago

ORM doesn't matter because it isn't any different from a VB3 app that tries to do RBAR via CBAC.

App developers all stripes forget the most basic, fundamental, elemental approach to

accessing and changing data in almost any database. I'm tempted not to tell you. In fact I'm not going to tell you. If I tell you then you will start doing code against databases correctly and then it will reduce the work of Consultants-To-The-Rescue.
Only nimrods, buffoons and ignoramuses think they can process data in a front-end or middle-ware. Let them go back to working at McDonald's where they can provide some public service. Such coders are NOT providing any writing code.

1 • Reply • Share ›

Igor Rozenberg • 6 months ago

Yegor, "Я тучка, тучка, тучка - я вовсе не медведь!"
Do not forget flagship of the Microsoft ORM - Entity Framework!
I would add one more argument - DAL based on ORM suppose to work with MULTIPLE databases, while 90% of implementation
work with single one!
Most of Web developers HATE SQL - it's not a GLAMOR technology. And because they refuse to learn SQL they would re-invent
data processing bicycle by bringing all data from server to client and ignoring a proven ways (database views, stored procedures
and user-defined functions). As a result everybody is happy to develop a "compiler" but unable to deliver a simple business report
that required more than one table :(

2 • Reply • Share ›

Yegor Bugayenko author Igor Rozenberg • 6 months ago

Yes, totally agree, thanks for sharing your thoughts!

• Reply • Share ›

Igor Rozenberg Yegor Bugayenko • 6 months ago

Now instead of SPAGHETTI code we have a LASAGNA code (multilayer aka Russian Napoleon cake) :)

• Reply • Share ›

Marcos Douglas Santos • 6 months ago

How would you implement an update method to update the title of an instance of Post? Thanks.

• Reply • Share ›

Yegor Bugayenko author Marcos Douglas Santos • 6 months ago

Something like this:

```
public void title(String text) {
  new JdbcSession(this.dbase)
    .sql("UPDATE post SET title = ? WHERE id = ?")
    .set(text)
    .set(this.number)
    .update(Outcome.VOID);
}
```

• Reply • Share ›

Reply • Share ›

Marcos Douglas Santos ➦ Yegor Bugayenko • 6 months ago

Ok, now I understood you.
In your example the `void title(String text)` doesn't change the state of object. It continuous immutable.

• Reply • Share ›

Yegor Bugayenko author ➦ Marcos Douglas Santos • 6 months ago

Yes, exactly. The `post` object remains immutable, he doesn't change his state. Instead, he modifies the row in the database table.

1 • Reply • Share ›

Guest ➦ Yegor Bugayenko • 6 months ago

This method in `ConstPost` class? Sorry, but for me this is a setter without "set" prefix and this method already exists - a "get" method :)

• Reply • Share ›

Martin ➦ Guest • 5 months ago

Of course it is. But his conceptual magic wand changes it into something else...

Plus, his ORM performs N updates , one for each attribute. Really nasty.

• Reply • Share ›

Nash • 6 months ago

Patternitis... This time the Onion pattern: There are endless layers, and peeling all of them back makes you cry.

3 • Reply • Share ›

Jcl • 6 months ago

One of the biggest advantages to ORMs (among many others) is that they are almost SQL-agnostic... and you may reckon T-SQL is very different than PL/SQL, or other SQLs in different database engines . Also, whenever any of your tables changes you need to find where it's used in your SQL sentences and change it accordingly... that's done automatically on a ORM. Change a navigation property from 1..n to n..n? No problem, the ORM does that for you (otherwise find and replace all usages of that property on every object that "may" use that property).

An ORM aims to make the code readable and storage-agnostic... and if they keep evolving, you'll soon not need to know *anything* about the underlying storage (I get we're not there yet, but we are approaching). It makes your life way easier when coding any application... I'd risk it and say even makes life easier for small programs.

You are complicating everything in favour of "patterns"... patterns should be there to make development easier and/or more structured and easy to maintain. You are regretting years of research just to comply to a "pattern"... a pattern that, for the most part, is wrongly thought according to today's development techniques thought according to today's development techniques.

ORMs exist and are popular for a reason, not just because "they are mainstream". Sure, they have their problems (which are being actively worked on, at least on Hibernate and Entity Framework), but all those problems are nothing related to what you describe on your article, and your way of programming brings more trouble than it solves.

The more I read your articles, the more I think you just have fun "arguing against the system" without truly understanding the implications. I wonder if you have ever worked on big projects with big programming teams and tried to solve the problems they pose (despite the pomposity of your 'About' page).

2 • Reply • Share ›

Igor Rozenberg ➦ Jcl • 6 months ago

@jcl why professional drivers do not drive automatic cars? Because they would like to stay in control and not do not treat car components as a black box. You could completely ignore database schema during development BUT it would be bite you back when performance suddenly became an issue or data model should be changed.

1 • Reply • Share ›

Jcl ➦ Igor Rozenberg • 6 months ago

Igor, following your pattern, addmitedly exaggerating, we should just disregard the OSes and write everything down to the bare metal ourselves. That's absurd. Tools are there to use them. *Professional* programmers have coded ORMs... professional programmers *use* ORMs. And they make life easier... at expense of performance, maybe, but everything that makes anything easier in computing comes at the cost of some performance. From operating systems to using OpenGL. One would never think these days to write raw direct access to a graphics card today, but feel OpenGL or DirectX is a good choice... isn't it underperformant compared to bare metal? Definitely. Is it worth of it? I'd say yes.

ORMs are not a "static technology" either.. it's not that ORMs were invented then were stuck there... they are advancing and getting better. I don't feel you can use an ORM today and really forget about the underlying storage/persistence technology, but we will eventually get there. Today's performance problems will be fixed aswell.

Disregarding ORMs in favour of writing direct SQL queries in the objects, these days, is just wrong for any decently sized software... and voicing it saying it's "offensive" (as this article does) is just plainly wrong.

I don't say writing SQL queries in the objects may not work for some scenarios... however the author is doing precisely that (or, the reverse of that), saying an ORM should never be used because it's an antipattern (an offensive one, in fact!).

• Reply • Share ›

Igor Rozenberg ↪ Jcl • 6 months ago

You are exadurating of course! I would use any tools that improve productivity rather than re-write OS in assembler. The question is how mature those tools and would they be resilient enough to survive next version. I could trust code (mine or somebody else), but got a lot of painful experience when buggy designers crashed my IDE. If I would follow you logic, we have to use Gui designers to manage OS tasks instead of automating them with PowerScript.

• Reply • Share ›

Ben • 6 months ago

What about the idea to not access tables directly at all, but use stored functions or view's? Basically I just don't like to access tables at all. There are always problems if you want's to change something on the table-level. What if now a table is the way it's defined okay, but some weeks later you'd like to organize your data differently? I don't mean any changes from the "row" settings, but maybe you rename it, do some normalization or partitioning? If you access the data via a stored function or view you can hide this database specific handling of your data completely. Can your method used for this as well?

• Reply • Share ›

Yegor Bugayenko author ↪ Ben • 6 months ago

Yes, I'm totally in favor of VIEWs vs. TABLEs. On Java level, it doesn't make any difference. The object of class "Post" can do whatever he wants with the SQL requests he is sending to the database. He can call stored procedures or select from VIEWs.

• Reply • Share ›

anon • 6 months ago

Did you even read the fowler article you cited?

7 • Reply • Share ›

tkruse • 6 months ago

See http://en.wikipedia.org/wiki/D..., the solution that existed before ORMs. ORMs just generalize DAOs, and provide much more functionality that would else be replicated among entities and projects.

1 • Reply • Share ›

domm_plix • 6 months ago

DBIx::Class, the de-facto standard Perl ORM, works quite like what you're describing: https://metacpan.org/pod/DBIx:...

It also has a very powerful and smart syntax to join tables and prefetch data so you don't end up hitting the DB thousands of times.

• Reply • Share ›

DavidGale • 6 months ago

Where is @Immutable annotation you are using in your examples defined?

• Reply • Share ›

Yegor Bugayenko author ➦ DavidGale • 6 months ago

In jcabi-aspects: http://aspects.jcabi.com/annot...

1 ^ | ˅ • Reply • Share ›

aRRRRrrr • 6 months ago

You can implement the Posts object using ORM, I don't think it's a problem with ORM but with the way we are using the persistent layer.

Though I can't stop noticing that the Posts class is quite similar to a DAO object? or am I missing something.

3 ^ | ˅ • Reply • Share ›

Yegor Bugayenko author ➦ aRRRRrrr • 6 months ago

If "ORM" stays inside an object and nobody outside it can touch it - I have no problem with it. As soon as you start using "ORM" outside of the object, this is where you break encapsulation and object-oriented programming doesn't work any more. No matter how we call it, ORM, DAO or Active Record - we should encapsulate SQL interactions into our objects.

3 ^ | ˅ • Reply • Share ›

noah ➦ Yegor Bugayenko • 6 months ago

I'm definitely in this camp. I'd rather let ORM write SQL for me. I think the problem is that Rails-like frameworks tend to use the ORM models at the top layer, instead of just using them for data access.

Where I work, all our models implement an interface and have a HibernateModel implementation. We haven't had a lot of need to wrap our models yet, but it's always an option.

^ | ˅ • Reply • Share ›

aRRRRrrr ➦ Yegor Bugayenko • 6 months ago

I guess it's not so much an anti orm argument then :) the classic anti orm arguments might not be missing the point.

1 ^ | ˅ • Reply • Share ›

Lyubomyr Shaydariv • 6 months ago

One little technical question, not about the ORM: do you consider Iterable<t> harmful while returning it from Posts? I do. As far as I can see, Collection<t> (or even List<t>) is a better choice here, because the data tends to be fetched _before_ any processing in the Java domain code. Returning an Iterable<t> in such cases reminds me attempts to adapt a ResultSet to an Iterable<t> (may produce resource leaks if an Iterable<t> is not fully iterated or if an exception is thrown). I think that if you need something to iterate over, you better use a consumer accepting a single item (Consumer<t>). The callbacks hell may grow, but this is safe, especially with lambdas.

^ | ˅ • Reply • Share ›

I think that it's much better to design a method in the interface as returning an `Iterable`. Mostly because it gives much more flexibility to the implementing class. Even if there are millions of rows, we still can return something back in a few milliseconds. In real-life projects I usually use my own interface `Pageable`, instead of `Iterable`. It's not a SQL, but still can explain what I mean: https://github.com/netbout/net...

1 • Reply • Share ›

Lyubomyr Shaydariv → Yegor Bugayenko • 6 months ago

I think that `Pageable<t>` tends to bind to more concrete `Pageable<t>` rather than `Iterable<t>`, but it looks an interesting idea to me, and I hope to check it deeper later, thanks. What I meant in my comment above is that I find `Iterable<t> foo()` somewhat more unsafe than `void bar(Consumer<t> consumer)` might be due to possible resource leaks due to resources that might be left unclosed if the iterable is not read completely. Or do I miss the point?

1 • Reply • Share ›

Yegor Bugayenko author → Lyubomyr Shaydariv • 6 months ago

Yes, you're right, resource leakage may easily happen if we don't destroy the paginator/iterator. But consuming all database rows into memory is even worse than that, I think. Resource leakage can be fixed, since it's a bug. Fetching everything in memory is a design flaw, which is not a bug and is not reproducible/fixable.

1 • Reply • Share ›

Lyubomyr Shaydariv → Yegor Bugayenko • 6 months ago

No-no, I didn't mean to "fetch everything". This may be extremely harmful out of my experience. I rather meant that an eager Collection<t> is usually an in-memory cached storage whereas a lazy Iterable<t> is supposed to be generated from any source not forcing you to read untill its Iterator.hasNext() returns false. This is the root of evil.

I just meant that exposing anything Iterable<t> is not that good idea as it might look, because your iterators may be left in unfinished state unless you're really sure (you can't be -- it all depends on the call-site) this iterable is either copied into a Collection<t> and then exposed out (I find re-iterating over the very source causing acquiring another JDBC Connection and fetching another ResultSet somewhat strange and expensive + you must be sure that Itetable<t> --> Collection<t> is fail-safe [i.e. Guava, Lists.newArrayList(Iterable) is not]).

... Or reading every single row and delegating the row to a consumer thus letting any Consumer.accept() to fail making sure your JDBC resources are closed or pushed back to the pool without leakage (no matter if it's a chained/mapping/whatever consumer -- out of scope right now).

Thus, if I'd have to write an RDBMS-->CSV routine:

* I would not use a lazy Iterable<t> (tends to possible resource leaks if csv.writeRow() fails):
for (final Row row : dbSource.all()) {
csv.writeRow(row);
}

* I would not use an eager Collection<t> (due to memory wasting for a data streaming operation)

* I would use a Consumer<t> inversing the iteration control and taking care the resource management:
dbSource.all(row -> csvExporter.writeRow(row))

• Reply • Share ›

Rudi Angela • 6 months ago

Though I do share your aversion to 'anemic entities' (coined by Martin Fowler a while ago), I don't think it's inherent to ORM's. AAMOF, I've used full blown entities (non-anemic) in combination with Hibernate and other ORM's. The ORM's I've used have not impeded my use of full blown entities in any way. But indeed, employing entities as mere DTO's is something that is propagated in the JEE community.

1 • Reply • Share ›

yortus • 6 months ago

What about join-heavy scenarios? What classes would "own" the data and operations for complex joins?

I had a colleague implement something like what you've presented. Very neat for single table CRUD. Every table had singular/plural class pairs implementing their own CRUD methods. Unfortunately, all DB access was now siloed on a per-table basis, so the many queries that involved joins were horribly inefficient, basically loading all the data for each table into memory, and joining it there instead of letting the database engine handle the joins. After instrumenting this code, we found it to be 15x-100x slower than code that let the DB engine handle the joins.

1 • Reply • Share ›

Hlodowig yortus • 6 months ago

Use MongoDB and forget all the join nonsense ;)

1 • Reply • Share ›

Yegor Bugayenko author yortus • 6 months ago

Of course, each SQL-speaking object should be "database-savvy". He has to know

More from yegor256.com

yegor256.com recently published

There Can Be Only One Primary Constructor

33 Comments Recommend

yegor256.com recently published

Three Things I Expect From a Software Architect

23 Comments Recommend

yegor256.com recently published

How to Implement an Iterating Adapter

13 Comments Recommend