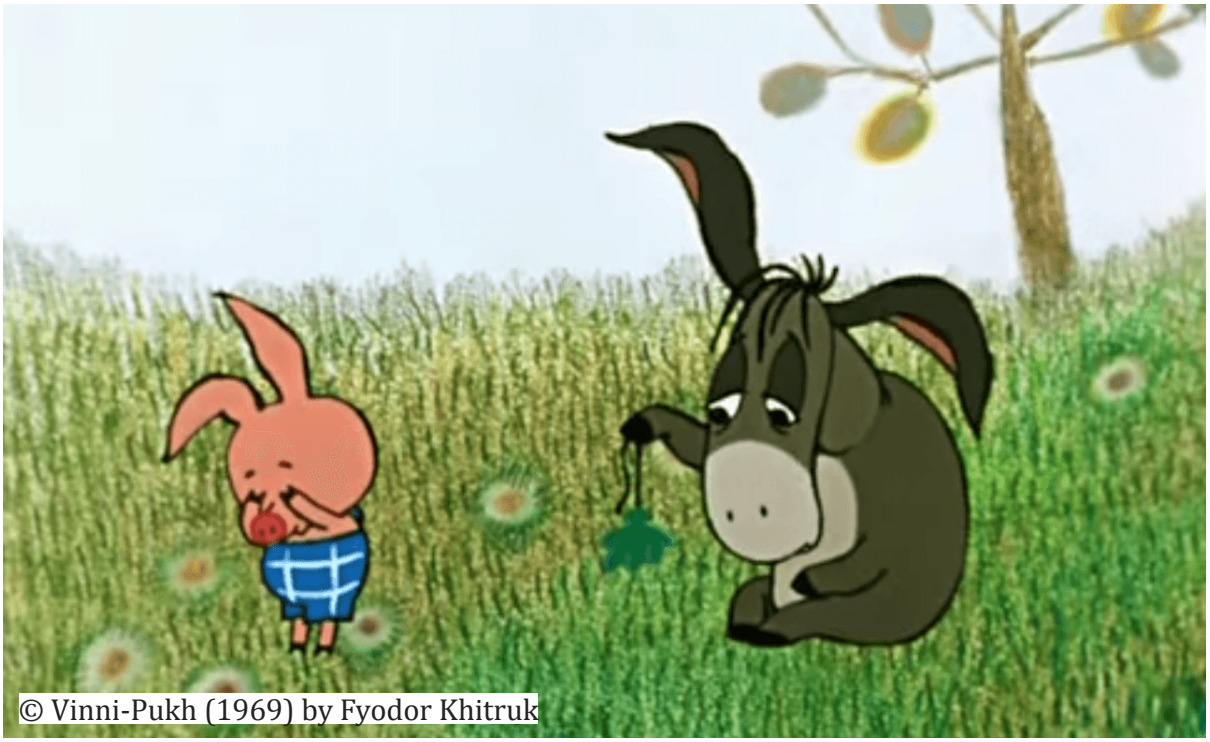
http://www.yegor256.com/2014/12/01/orm-offensive-anti-pattern.html

ORM Is an Offensive Anti-Pattern

1 December 2014 Yegor Bugayenko

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 web app or an enterprise-size system with thousands of tables and CRUD manipulations on them. What is the alternative? **SQL-speaking objects**.



© Vinni-Pukh (1969) by Fyodor Khitruk

How ORM Works

Object-relational mapping (ORM) is a technique (a.k.a. design pattern) of accessing a relational database from an object-oriented language (Java, for example). There are multiple implementations of ORM in almost every language; for example: Hibernate for Java, ActiveRecord for Ruby on Rails, Doctrine for PHP, and SQLAlchemy for Python. In Java, the ORM design is even standardized as JPA.

First, let's see how ORM works, by example. Let's use Java, PostgreSQL, and Hibernate. Let's say we have a single table in the database, called `post`:

```
+-----+------------+-------------------------+
| id  | date       | title                   |
+-----+------------+-------------------------+
|   9 | 10/24/2014 | How to cook a sandwich  |
|  13 | 11/03/2014 | My favorite movies      |
|  27 | 11/17/2014 | How much I love my job  |
+-----+------------+-------------------------+
```

Now we want to CRUD-manipulate this table from our Java app (CRUD stands for create, read, update, and delete). First, we should create a `Post` class (I'm sorry it's so long, but that's the best I can do):

```
@Entity
@Table(name = "post")
public class Post {
  private int id;
  private Date date;
  private String title;
  @Id
  @GeneratedValue
  public int getId() {
    return this.id;
  }
  @Temporal(TemporalType.TIMESTAMP)
  public Date getDate() {
    return this.date;
  }
```

```
  public Title getTitle() {
    return this.title;
  }
  public void setDate(Date when) {
    this.date = when;
  }
  public void setTitle(String txt) {
    this.title = txt;
  }
}
```

Before any operation with Hibernate, we have to create a session factory:

```
SessionFactory factory = new AnnotationConfiguration()
  .configure()
  .addAnnotatedClass(Post.class)
  .buildSessionFactory();
```

This factory will give us "sessions" every time we want to manipulate with `Post` objects. Every manipulation with the session should be wrapped in this code block:

```
Session session = factory.openSession();
try {
  Transaction txn = session.beginTransaction();
  // your manipulations with the ORM, see below
  txn.commit();
} catch (HibernateException ex) {
  txn.rollback();
} finally {
  session.close();
}
```

When the session is ready, here is how we get a list of all posts from that database table:

```
List posts = session.createQuery("FROM Post").list();
for (Post post : (List<Post>) posts){
```

```
  System.out.println("Title: " + post.getTitle());
}
```

I think it's clear what's going on here. Hibernate is a big, powerful engine that makes a connection to the database, executes necessary SQL `SELECT` requests, and retrieves the data. Then it makes instances of class `Post` and stuffs them with the data. When the object comes to us, it is filled with data, and we should use getters to take them out, like we're using `getTitle()` above.

When we want to do a reverse operation and send an object to the database, we do all of the same but in reverse order. We make an instance of class `Post`, stuff it with the data, and ask Hibernate to save it:

```
Post post = new Post();
post.setDate(new Date());
post.setTitle("How to cook an omelette");
session.save(post);
```

This is how almost every ORM works. The basic principle is always the same — ORM objects are anemic envelopes with data. We are talking with the ORM framework, and the framework is talking to the database. Objects only help us send our requests to the ORM framework and understand its response. Besides getters and setters, objects have no other methods. They don't even know which database they came from.

This is how object-relational mapping works.

What's wrong with it, you may ask? Everything!

What's Wrong With ORM?

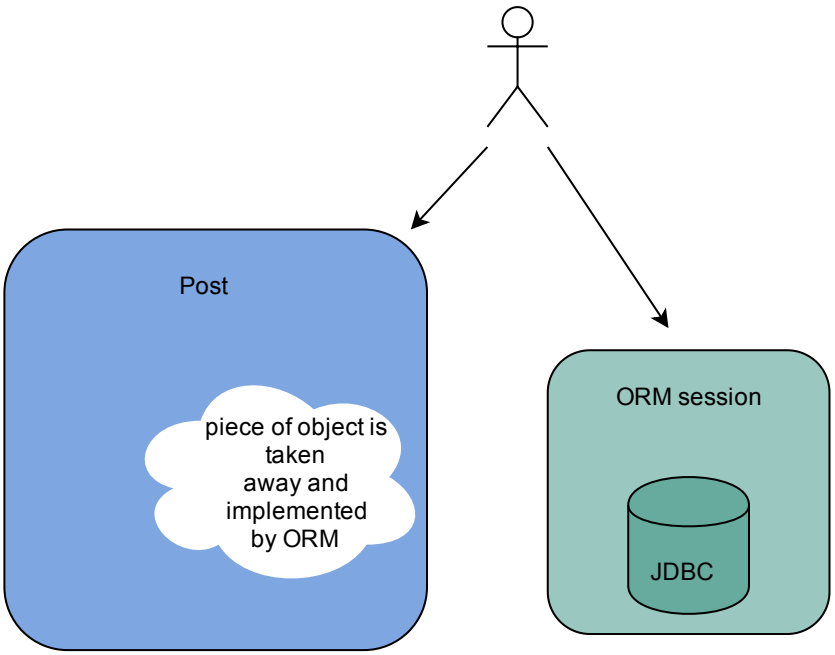
Seriously, what is wrong? Hibernate has been one of the most popular Java libraries for more than 10 years already. Almost every SQL-intensive application in the world is using it. Each Java tutorial would mention Hibernate (or maybe some other ORM like

TopLink or OpenJPA) for a database-connected application. It's a standard *de-facto* and still I'm saying that it's wrong? Yes.

I'm claiming that the entire idea behind ORM is wrong. Its invention was maybe the second big mistake in OOP after NULL reference.

Actually, I'm not the only one saying something like this, and definitely not the first. A lot about this subject has already been published by very respected authors, including OrmHate by Martin Fowler (not against ORM, but worth mentioning anyway), Object-Relational Mapping Is the Vietnam of Computer Science by Jeff Atwood, The Vietnam of Computer Science by Ted Neward, ORM Is an Anti-Pattern by Laurie Voss, and many others.

However, my argument is different than what they're saying. Even though their reasons are practical and valid, like "ORM is slow" or "database upgrades are hard", they miss the main point. You can see a very good, practical answer to these practical arguments given by Bozhidar Bozhanov in his ORM Haters Don't Get It blog post.



The main point is that ORM, instead of encapsulating database interaction inside an object, extracts it away, literally tearing a solid and cohesive living organism apart. One part of the object keeps the data while another one, implemented inside the ORM

engine (session factory), knows how to deal with this data and transfers it to the relational database. Look at this picture; it illustrates what ORM is doing.

I, being a reader of posts, have to deal with two components: 1) the ORM and 2) the "ob-truncated" object returned to me. The behavior I'm interacting with is supposed to be provided through a single entry point, which is an object in OOP. In the case of ORM, I'm getting this behavior via **two** entry points — the ORM engine and the "thing", which we can't even call an object.

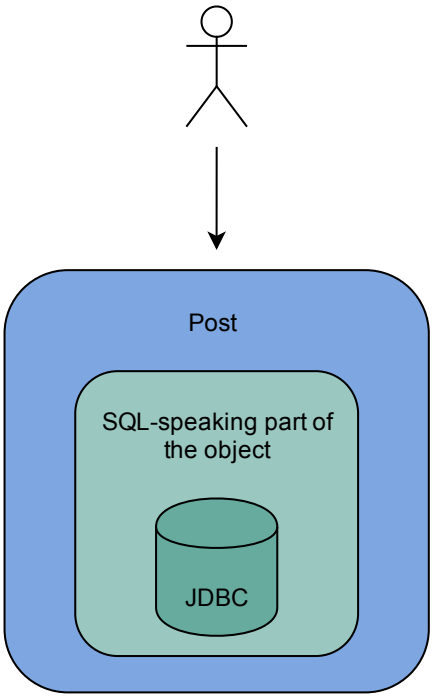
Because of this terrible and offensive violation of the object-oriented paradigm, we have a lot of practical issues already mentioned in respected publications. I can only add a few more.

SQL Is Not Hidden. Users of ORM should speak SQL (or its dialect, like HQL). See the example above; we're calling `session.createQuery("FROM Post")` in order to get all posts. Even though it's not SQL, it is very similar to it. Thus, the relational model is not encapsulated inside objects. Instead, it is exposed to the entire application. Everybody, with each object, inevitably has to deal with a relational model in order to get or save something. Thus, ORM doesn't hide and wrap the SQL but pollutes the entire application with it.

Difficult to Test. When some object is working with a list of posts, it needs to deal with an instance of `SessionFactory`. How can we mock this dependency? We have to create a mock of it? How complex is this task? Look at the code above, and you will realize how verbose and cumbersome that unit test will be. Instead, we can write integration tests and connect the entire application to a test version of PostgreSQL. In that case, there is no need to mock `SessionFactory`, but such tests will be rather slow, and even more important, our having-nothing-to-do-with-the-database objects will be tested against the database instance. A terrible design.

Again, let me reiterate. Practical problems of ORM are just consequences. The fundamental drawback is that ORM tears objects apart, terribly and offensively violating the very idea of what an object is.

SQL-Speaking Objects



What is the alternative? Let me show it to you by example. Let's try to design that class, `Post`, my way. We'll have to break it down into two classes: `Post` and `Posts`, singular and plural. I already mentioned in one of my previous articles that a good object is always an abstraction of a real-life entity. Here is how this principle works in practice. We have two entities: database table and table row. That's why we'll make two classes; `Posts` will represent the table, and `Post` will represent the row.

As I also mentioned in that article, every object should work by contract and implement an interface. Let's start our design with two interfaces. Of course, our objects will be immutable. Here is how `Posts` would look:

```
interface Posts {
  Iterable<Post> iterate();
  Post add(Date date, String title);
}
```

This is how a single `Post` would look:

```
interface Post {
  int id();
  Date date();
  String title();
}
```

Here is how we will list all posts in the database table:

```
Posts posts = // we'll discuss this right now
for (Post post : posts.iterate()){
  System.out.println("Title: " + post.title());
}
```

Here is how we will create a new post:

```
Posts posts = // we'll discuss this right now
posts.add(new Date(), "How to cook an omelette");
```

As you see, we have true objects now. They are in charge of all operations, and they perfectly hide their implementation details. There are no transactions, sessions, or factories. We don't even know whether these objects are actually talking to the PostgreSQL or if they keep all the data in text files. All we need from `Posts` is an ability to list all posts for us and to create a new one. Implementation details are perfectly hidden inside. Now let's see how we can implement these two classes.

I'm going to use jcabi-jdbc as a JDBC wrapper, but you can use something else or just plain JDBC if you like. It doesn't really matter. What matters is that your database interactions are hidden inside objects. Let's start with `Posts` and implement it in class `PgPosts` ("pg" stands for PostgreSQL):

```
final class PgPosts implements Posts {
  private final Source dbase;
  public PgPosts(DataSource data) {
    this.dbase = data;
  }
```

```
  public Iterable<Post> iterate() {
    return new JdbcSession(this.dbase)
      .sql("SELECT id FROM post")
      .select(
        new ListOutcome<Post>(
          new ListOutcome.Mapping<Post>() {
            @Override
            public Post map(final ResultSet rset) {
              return new PgPost(
                this.dbase,
                rset.getInteger(1)
              );
            }
          }
        )
      );
  }
  public Post add(Date date, String title) {
    return new PgPost(
      this.dbase,
      new JdbcSession(this.dbase)
        .sql("INSERT INTO post (date, title) VALUES (?, ?)")
        .set(new Utc(date))
        .set(title)
        .insert(new SingleOutcome<Integer>(Integer.class))
    );
  }
}
```

Next, let's implement the `Post` interface in class `PgPost`:

```
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 Date date() {
```

```
    return new JdbcSession(this.dbase)
      .sql("SELECT date FROM post WHERE id = ?")
      .set(this.number)
      .select(new SingleOutcome<Utc>(Utc.class));
  }
  public String title() {
    return new JdbcSession(this.dbase)
      .sql("SELECT title FROM post WHERE id = ?")
      .set(this.number)
      .select(new SingleOutcome<String>(String.class));
  }
}
```

This is how a full database interaction scenario would look like using the classes we just created:

```
Posts posts = new PgPosts(dbase);
for (Post post : posts.iterate()){
  System.out.println("Title: " + post.title());
}
Post post = posts.add(new Date(), "How to cook an omelette");
System.out.println("Just added post #" + post.id());
```

You can see a full practical example here. It's an open source web app that works with PostgreSQL using the exact approach explained above — SQL-speaking objects.

What About Performance?

I can hear you screaming, "What about performance?" In that script a few lines above, we're making many redundant round trips to the database. First, we retrieve post IDs with `SELECT id` and then, in order to get their titles, we make an extra `SELECT title` call for each post. This is inefficient, or simply put, too slow.

No worries; this is object-oriented programming, which means it is flexible! Let's create a decorator of `PgPost` that will accept all data in its constructor and cache it internally, forever:

```
final class ConstPost implements Post {
  private final Post origin;
  private final Date dte;
  private final String ttl;
  public ConstPost(Post post, Date date, String title) {
    this.origin = post;
    this.dte = date;
    this.ttl = title;
  }
  public int id() {
    return this.origin.id();
  }
  public Date date() {
    return this.dte;
  }
  public String title() {
    return this.ttl;
  }
}
```

Pay attention: This decorator doesn't know anything about PostgreSQL or JDBC. It just decorates an object of type `Post` and pre-caches the date and title. As usual, this decorator is also immutable.

Now let's create another implementation of `Posts` that will return the "constant" objects:

```
final class ConstPgPosts implements Posts {
  // ...
  public Iterable<Post> iterate() {
    return new JdbcSession(this.dbase)
      .sql("SELECT * FROM post")
      .select(
        new ListOutcome<Post>(
          new ListOutcome.Mapping<Post>() {
            @Override
            public Post map(final ResultSet rset) {
              return new ConstPost(
                new PgPost(
                  ConstPgPosts.this.dbase,
                  rset.getInteger(1)
```

```
                ),
                Utc.getTimestamp(rset, 2),
                rset.getString(3)
              );
            }
          }
        )
      );
  }
}
```

Now all posts returned by `iterate()` of this new class are pre-equipped with dates and titles fetched in one round trip to the database.

Using decorators and multiple implementations of the same interface, you can compose any functionality you wish. What is the most important is that while functionality is being extended, the complexity of the design is not escalating, because classes don't grow in size. Instead, we're introducing new classes that stay cohesive and solid, because they are small.

What About Transactions?

Every object should deal with its own transactions and encapsulate them the same way as `SELECT` or `INSERT` queries. This will lead to nested transactions, which is perfectly fine provided the database server supports them. If there is no such support, create a session-wide transaction object that will accept a "callable" class. For example:

```
final class Txn {
  private final DataSource dbase;
  public <T> T call(Callable<T> callable) {
    JdbcSession session = new JdbcSession(this.dbase);
    try {
      session.sql("START TRANSACTION").exec();
      T result = callable.call();
      session.sql("COMMIT").exec();
      return result;
```

```
      } catch (Exception ex) {
        session.sql("ROLLBACK").exec();
        throw ex;
      }
    }
  }
```

Then, when you want to wrap a few object manipulations in one transaction, do it like this:

```
new Txn(dbase).call(
  new Callable<Integer>() {
    @Override
    public Integer call() {
      Posts posts = new PgPosts(dbase);
      Post post = posts.add(new Date(), "How to cook an omelette");
      posts.comments().post("This is my first comment!");
      return post.id();
    }
  }
);
```

This code will create a new post and post a comment to it. If one of the calls fail, the entire transaction will be rolled back.

This approach looks object-oriented to me. I'm calling it "SQL-speaking objects", because they know how to speak SQL with the database server. It's their skill, perfectly encapsulated inside their borders.