ELEGANT OBJECTS 1&2: REVIEW

Stian Z. Vrba, December 2017

Abstract

The author presents his interpretation of OO which he claims stems from a couple of books listed in the bibliography. He did not do his history lessons, however: throughout the book he criticizes Java, C++ and other OO languages, wishing for a pure OO language where, for example, even “if” statement could be an object. Not knowing that such language *already exists* – Smalltalk – he started working on his own pure OO language. In other words, the author is talking about “true” OO programming without *ever* having used or studied a true OO language. This already hurts the credibility of the material.

The author claims that his interpretation of OO makes programs more maintainable and performant, but does not realize that he is reiterating the tension between eager and lazy evaluation. Without realizing it, he is proponent of the latter, and is unaware that precisely lazy evaluation is a source of performance problems and excessive memory usage in Haskell, as well as being a big stumbling block for newcomers. It is problematic to the extent that people are experimenting with Haskell dialects where eager evaluation is the default. Even Simon Peyton Jones has said that “the next Haskell will be strict”.

Even though the book contains a few sound advices (the chapter on using fakes instead of mocks, “fail fast” philosophy), the overall tone of the book is condescending and gives an impression that the author thinks very high of himself because reading a couple of books on OO lead him to some kind of “revelation”. Alas, his (re)interpretations of the books he read seem dubious.

These books are way too expensive and basically a waste of money. I’m sorry I bought them.

POV for this review

This review is based on several (philosophical) beliefs.

First: a class should be designed around one or more *invariants* which define the allowable states of its instances (objects). Constructor establishes the invariants (or throws an exception if it cannot for some reason), public methods must preserve them, while private and protected methods may break them temporarily (during transitions between allowable states). The behavior of the class is described by its invariants. If you cannot clearly state class invariants, then the class is ill-conceived and should be rethought.

Second: any code that you write is a liability. It will almost certainly have bugs, it will have to be updated as requirements change, etc. I operate with a general “liability scale” of code:

1. The not-written code (zero liability),
2. Services provided by the operating system,
3. Services provided by the language run-time,
4. Functionality provided by proven external libraries,
5. The code you write (greatest liability).

Third: I try to automate whatever can be automated, if the cost is not too high. Computers are much better at following instructions than humans are in following prescribed routines (e.g., writing boiler-plate code or keeping it in sync with changes that happen elsewhere).

Lazy Evaluation vs. "fail-fast" vs. "maintainability"

Section 1.1.3[1] sets the stage for the rest of the books which I (somewhat freely) summarize as follows:

1. The author is a proponent of lazy evaluation and claims that it leads to "more maintainable" code because it's "declarative".
2. The author is a proponent of fail-fast error handling, but does not seem to realize that it contradicts the 1st point.
3. Class invariants are nowhere mentioned where object behavior is discussed.
4. Code duplication is better than breaking encapsulation ("taking objects apart").

Here I cite an example that perfectly demonstrates the first three points:

```
class Cash {
       private Number dollars;
       Cash(String dlr) { this.dollars = new StringAsInteger(dlr); }
}

class StringAsInteger implements Number {
       private String source;
       StringAsInteger(String src) { this.source = src; }
       int intValue() { return Integer.parseInt(source); }
}
```

He calls this "declarative style" because objects "do" something only when asked (e.g., converting a string to an integer). Paradoxically, he also claims that this makes programs more maintainable. But think: the time and place of *creation* of a Cash object may be far away from the place of *use* as a Number. If a Cash object is initialized with a non-numeric string, this will be first discovered at the place of use (`parseInt` will throw an exception). How are we supposed to discover when, where, and why the object was created? This "hunt" will be very much alike hunting memory corruption problems in C or "debugging" complex spreadsheets.

With eager evaluation, or at least with constructor establishing invariants ("a Cash object can always be converted to some numeric type"), we have stack back-pointers that allow us to reconstruct the stack trace which immediately relates the failure with its cause. To achieve the same level of "debuggability" with classes like the above, we'd have to introduce in each object a back-pointer that'd point back to its creator. Technically not difficult to do (though tedious), but it'd practically thwart garbage-collection. Indeed, without back-pointers, the creator of an invalid Cash object may have already been garbage-collected by the time the invalid Cash object is used.

The author never concretely clarifies what he means by "more maintainable programs", but I, personally, do not consider programs that are "hard(er) to debug" as "more maintainable". Quite the opposite.

[1]In this numbering scheme, the first number is the *book volume*.

Further, in section 1.3.2.2 he presents an unsubstantiated claim that declarative style somehow results in faster programs. First, this is nonsense: the “speed” of the program is determined by the amount of work the program needs to do to accomplish its task. Second, Haskell is a prime example where naïve use of declarative style (combination of immutability and laziness) leads to *slow* solutions unusable in the real world. Examples: Quick sort looks compelling, but is anything but quick, and sieve of Eratosthenes suffers from the same problem.

References vs objects

Here I summarize problems in the text that stem from the author treating a *reference to an object* as if it were *the object itself* as well as problems arising from disregarding *values*.

On object identity

Already in section 1.2.1 the author states that Java and C++ are flawed because object identity is different from its state in these languages. According to him, this is how a pure OO-language should work:

```
Cash x = new Cash(29, 95, “USD”);
Cash y = new Cash(29, 95, “USD”);
assert x.equals(y);
assert x == y;
```

Now, if you study a bit of history of OO you will learn that Simula, the first OO language, was designed around the idea of *simulating* the real world. West, in his “Object thinking” (also referenced by the author) writes that objects in a well-designed OO program simulate the objects of the real world.

The contradiction here is that the real world is full of examples where objects have the same state but different identity: coins (of the same currency and denomination), books (of the same author, title, edition, publisher), and so on… Object identity allows us to *abstract away* the parts of an object’s state that’s not interesting to model in the program.

The author’s idea of a pure OO language would make it difficult to model the difference between the following two cases:

1. A student *loaning* a book from another student (or a library),
2. A student *buying* a book in a bookstore.

In the first case, the *same* book changes owners (temporarily), while in the second case a *new* book comes to be[2] (the authors getting richer as a side-effect), but the new book is “equal” (value equality) to the original book.[3]

[2] Let’s pretend that in this example the bookstore acts as a “factory” of books. After all, it is we – the programmers and designers – who decide to which level of detail the real world should be simulated for the program to perform a useful function.

[3] However familiar and “obvious” the terms “value”, “identity”, “equality”, “sameness” may seem, the issue at hand is far from simple or resolved and has bothered philosophers for millennia. For a mathematical overview, read an essay “When is one thing equal to some other thing?” by Barry Mazur. For a more accessible presentation of the issue, read about Theseus’s paradox.

On immutability

Discussion on immutability begins in section 1.2.6 which is relatively coherent. Things start to worsen in section 1.3.4 where he writes

> "In a perfect object-oriented world […] In an immutable class, all objects are identifiable by the state they encapsulate. The state is necessary and sufficient to identify an immutable object"

and reiterated in the sentence

> "The main difference between immutable and mutable objects is that an immutable one doesn't have an identity and its state never changes. More precisely, immutable objects' identities are exactly the same as their states."

Well, see the previous section for counter-examples. Yes, we could introduce an artificial[4] "unique id" attribute, but *why should we* when the object's identity gives us this for free?

In short, the discussion of immutability proceeds by "slicing" an object into its constituent data parts and saying that an object is "immutable" if its fields are final; at the same time the analysis *disregards the object's behavior.* (I'm sure the author will disagree with this, but read on.) As an example, section 2.5.8 discusses "gradients" [sic] of immutability where the author states that the following class

```
class GaugeInMemory {
        private final byte[] value = new byte[8];
        void update(double v) { … }
        String html() { … }
}
```

is immutable. However, the object *behaves as if* it were written in what the author calls "evil" way like this:

```
class GaugeInMemory {
        double value;
        void update(double v) { value = v; }
        String html() { … }
}
```

The *behavior* of the class is the same in both cases: how it presents itself in HTML depends on the sequence of previously received update messages, i.e., on the *current value* of the gauge. An immutable Gauge object would *always* respond in the same way to `html()` message, but `update()` method would return a (conceptual)[5] *copy* of the object with value field updated. Here the author has invented own naming and distinguishes between "constant" and "immutable" objects.

Since behavior is disregarded and copying (Cloneable interface in Java) and value-equality aren't even mentioned, I deem the whole discussion rather pointless. Consider this: how does it help that a Gauge is technically immutable (all fields are final and the behavior – rendering itself as HTML – is always the same) when a faulty `update()` call in another part of the program will cause the gauge to display a wrong value to the user?

[4] Artificial because it does not relate to anything in the real world. Coins do not have serial numbers printed on them, nor do Yegor's printed books.

[5] «Conceptual» because full copying of large data structures (e.g., lists and dictionaries) is impractical. Full copying can be in some cases avoided by using persistent data structures.

In section 1.3.4, the author also erroneously states that in Java, object identity can be overridden by defining an `equals()` method. This is simply **incorrect**: identity is always checked with `==` (equality of references) and *cannot* be overridden, while `equals()` is used to define equality of *values*. Not discussing values, or mixing up values with identity, is symptomatic of both books.[6]

Inconsistent use of "object creation"

In this section I will discuss the author's reinterpretation Law of Demeter from section 2.5.9. Here's the law itself quoted from the book:

> "For all classes C, and all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes: 1) the argument classes of M (including C), 2) the instance variable classes of C. Objects created by M, or by functions or methods that M calls, are considered as arguments of M"

(and can thus be used by M). Given the following snippet

```
class Report {
        void print(Garage garage) {
                int price = garage.getCar("BMW").getPrice();
                System.out.println("price: " + price)
        }
}
```

which violates the Law of Demeter, the author asks

> "And, really, what is the problem with asking a garage to return a car, which is able to do some service for us?"

The question is rhetorical, but I'll try to give an answer: the car could have a `drive()` method that must not be called without coordination with the garage because, e.g., the garage door has to be opened, the car may need to be signed out in some log, etc…

He reinterprets the last sentence of LoD to mean that direct access to object attributes is not allowed, for example `int price = garage.car.price`. The following, however, is allowed in his interpretation because each method call "builds an object for us": `year = garage.car().engine().model().year()`.

The glaring flaw in this section is the author not distinguishing between a method returning a reference to an *existing* object, and a method returning a *new* object. In the first case, M gets a shared reference to an object that is a component of another object (what LoD recommends against), while in the second case M gets the *only* reference to the object. For the author's interpretation of LoD to make sense, every single method returning (a reference to) an object would have to clone it. Often, this is neither possible nor desirable,[7] and even if it were, it would be expensive.

As a side-note: the example code in section 1.2.6.1, which is supposed to be an argument for immutability, "goes wrong" precisely because the author violates Law of Demeter, though in a non-obvious way.

[6] LISP, one of the author's favorite examples of a "good" language, has *four* different equality operators each being more "relaxed" than the previous one (i.e., eq implies eql implies equal implies equalp): `eq`, `eql`, `equal` and `equalp`.

[7] Though I would for sure like to have a garage that builds me a new car every time I ask for it 😉

A nice rationale for the Law can found here (it is similar in spirit to my argumentation here).

Difficult problems

Here I summarize implications of the author's ideas which are not discussed in the text. A common theme here is that the simplest, least error-prone (in my opinion) solution to the problems mentioned here involve getting access to the object's raw data (getters or reflection), or, rather, doing something *to* the object.

Object marshaling

Serialization

Sections 2.5.3 and 2.6.1.1 touch upon serialization and horizontal vs. vertical decomposition: With *horizontal* decomposition, we'd have a class and another serializer class that takes care of converting objects to their external representation (through getters or reflection); with *vertical*, each object knows how to serialize itself to a given stream or other "output" object. The author condemns horizontal decomposition, but admits that vertical will lead to code duplication, which is he claims is the lesser evil because it preserves encapsulation.

I have a hard time buying his argument for two main reasons:

- It is error-prone because it is manual and redundant: whenever a data member is added or removed, the (de)serialization code must also be updated.
- It requires every serializable class to know the low-level details of the format syntax which may be very complex.

Another reason is philosophical: an object *is* the data it encapsulates because the data determines the *run-time* behavior of the class; it is the *data* that changes at run-time, not the *code*. If a data member doesn't affect the behavior in any way, it can be deleted. On the other hand, if a method's actions do not depend on the values of any data members, the method is effectively static (static methods are another item on the author's list of bad techniques).

Therefore, I believe that sometimes it's appropriate to "open" an object to perform tasks like (de)serialization.[8] Here, annotations help with automating the process and telling the serializer which fields should be included in the external representation because some fields cannot be meaningfully serialized, e.g., mutexes.[9]

Deserialization

However, the inverse problem – that of reconstructing an object from its external representation – is only touched upon in section 2.6.4.7 where the author's position is that you should use constructors to restore an object from the stream.

[8] I wonder whether the author has the same objections to hibernating not only objects but the state of the complete operating system, together with running programs.

[9] I'm aware that this can also be used as an argument *against* opening objects. That's why it's important to think about class invariants, which (de)serialization must preserve.

The author overlooks the fact that deserialization is a chicken-and-egg problem. Take a Circle class as an example: according to the author's philosophy, the "right" way of deserializing an object would be to define a constructor like `Circle(InputStream is)`. However, to call this constructor, we must first read some metadata that tells us that the next object in the stream is a Circle (instead of, say, a Square). Only after this metadata is read can an object of the right type be instantiated (Circle) and told to reconstruct itself from the stream. This leads me to the next point.

The whole 2.6.4 chapter tries to tell you why you should not use reflection. Yes, it's a powerful tool that can be misused, but it can also be used to automate otherwise routine and error-prone tasks. In the context of deserialization, he gives no rational explanation of why writing a huge switch-case statement (which would instantiate the correct type with the stream argument) is better than generically creating an instance with reflection (`Class.forName, etc.`).

On decomposition

Object marshaling is just a particular example of a more general problem that the author does not address in the text: that of distributed responsibility for actions, or, rather, modeling of multi-object interactions. Consider a 2D dungeon game consisting of a Map, Player and Monsters (controlled by the computer). A Map consists of a collection of Cells of various types (wall, treasure, trap, teleport, etc.), while the Player knows its position in the dungeon and processes user commands, such as NESW movement.

First variation

The user presses a movement key, the Player asks the Map for permission to move to the new position, and updates its position if the permission is granted. The Map in turn delegates the request for permission to the Cell at the new position, which may refuse the request (e.g., if the Cell is a wall).

But who is responsible for updating the Player's position if the Player has stepped on a teleport Cell? I can think of only a few alternatives:

1. Player implements a `teleport` method that can set its position anywhere on the map (a *setter* with another name).
2. Teleport (a kind of Cell) performs the action *on* the Player.
3. Map queries the teleport for the destination Cell and updates the Player.

I do not believe that the 1st or 3rd alternatives are viable approaches. The 1st alternative "feels" unnatural because if the Player had an innate teleportation ability, we wouldn't need teleports in the Map. Neither the 1st nor the 3rd alternative offer easy extensibility because we may have different kinds of teleports (one that moves the player to a random location, one that moves him to a fixed location, a malfunctioning teleport that takes 150 health points from the Player, potentially killing him, etc.) This leaves us with the 2nd alternative: the teleport must "pry open" (setters and getters) the Player in order to modify its properties (position, health) without its "consent". It does not *ask* the player to teleport itself, it *acts on* it.

Second variation

The user presses a movement key and sends the (tentative) new coordinates to the Map. Instead of asking the Cell for permission, Map finds the target Cell and tells it to move the player to its position (`cell.move(player)`). A wall Cell wouldn't do anything to the player, an empty Cell would unconditionally update the player's position, a teleport Cell would perform its own behavior. A Cell containing a Monster would engage the Player in battle. The same mechanics would work for moving the monsters around.

Note that also in this variation we have objects that *act* on other objects instead of asking them to perform an action. Side-effects (teleportation, health points) would still be performed by the Cell itself.

Third variation

To fully uphold the encapsulation principle, which is the author's top priority, we'd have to introduce a Creature class (parent for Player and Monster) which would have to mirror the behavior of every possible type of Cell, to which the Cell would delegate its behavior:

```
abstract class Cell {
        abstract Position position();

        // NOT final, e.g., a trap could trigger only once
        void move(Creature c) { c.moveTo(this); }
}

class FixedTeleportCell extends Cell {
        Position position() { return new Position(10, 10); }
        // …
}

// … etc

abstract class Creature {
        private Position position;

        abstract protected void die();

        void moveTo(WallCell c) { /* do nothing */ }
        void moveTo(EmptyCell c) { position = c.position(); }
        void moveTo(FixedTeleportCell c) { position = c.position(); }
        void moveTo(MalfunctioningTeleportCell c) {
                position = c.position();
                health -= c.healthDeduction(this);
                if (health < 0) die();
        }
}
```

This would be an example of double dispatch if it worked,[10] but let's pretend for brevity and simplicity that it works; I believe that you understand the intention of the code. A truly working version would have to use reflection or the visitor pattern. Its generality allows us to make cells that behave differently depending on the type of Creature that stepped on them (e.g., a Trap wouldn't affect monsters).

But notice what happened: the behavior of (a subclass of) Cell is implemented in (a subclass of) Creature! The Cell class is reduced to an "anemic" dispatcher of messages to Creatures which implement the Cell's behavior. In fact, this implements a *joint behavior* for a given combination of concrete instances of Cells and Creatures. Logically, such behavior does not belong to *either* class. Yes, this design upholds encapsulation, but is it elegant? Well…

[10] The example doesn't work because overloaded methods are resolved at compile-time, not run-time. The code as written would always call `Creature.moveTo(Cell)`.

Lastly, consider Traps that trigger only once as noted in the code comment. The move method on a Trap must record whether it has triggered or not, but for a Creature to correctly implement the behavior of stepping on a Trap, it must use a "getter" to ask the Trap about its state. Thus, not even this solution allows us to completely encapsulate the behavior of the Trap.

Conclusion?

I really don't have any, only an opinion. Main-stream programming languages lack suitable means of expressing joint behavior of objects. Based on the views I expressed in section about serialization and what little I know about multiple dispatch in languages such as Perl 6 and LISP, I dare contradict the author and claim that data is still the king; objects only interpret it.

In examples like the above, I'd pick a class that has the *fewest number of specialized cases* when acting on other objects (probably Cell) and implement all required behavior there. Yes, using `instanceof` operator (another item on the author's "forbidden" list) to discern between the Player and a Monster where it's relevant.

Miscellaneous shortcomings

The problem with null

In section 1.3.3, the author mixes up (again) objects with references, or, rather, takes the object analogy too far by writing, for example:

> "Every time, before working with an object, we have to check his "realness""

Here, he refers to tests of the kind `if (mask == null)`. But mask in this line is not the object itself, it's a *reference* to the object. References are primitive values, like integers, and one of the few operations you can do on references is comparing their values (which amounts to comparing the object's identities).

The case against accepting null arguments in section 1.3.3 is reasonable, but the case against *returning* null values in section 1.4.1 less so. The author proposes three alternatives to returning null: 1) split a method in two (existence check + "find" operation that throws on "not found"), 2) return a collection of objects or 3) "null object" design pattern.

Yes, null is overused and abused, but it certainly has a semantic meaning that's reflected in the real world: the *absence* of an object. Imagine that you want to borrow a pen from a friend. Normally you ask, "Can I borrow your pen?",[11] and you expect to get either a pen or an answer like "I don't have one." The author's proposed alternatives would in the real world amount to:

1. First asking "Do you have a pen?",[12] followed by "Give me the pen." if the answer to the 1st question was "yes". If you dared say "Give me a pen" when your friend didn't have any, he'd slap you in the face (exception).

[11] Note that this is not a yes/no question, it's a request phrased as a question for politeness. You'd be baffled if you got a simple "yes" instead of a pen.

[12] Human language is weird: even on this question, I expect people to offer a pen if they have it instead of saying "yes". I cannot think up a question that would provoke a pure yes/no answer from humans that'd also always be correct. Probably something very formal like "Are you in possession of a pen?", but this risks a false positive answer

2. You ask for a pen, but your friend always gives you a box containing a pen or not.
3. If your friend didn't have a "real" pen, he'd somehow conjure up a "default non-pen". I can't even imagine how would this work in the real world.

By thinking of a program as a simulation of the real world, returning null seems like the most "natural" thing to do. Regarding the 1st alternative, the author offers no compelling argument for why writing

```
if (db.exists(userName)) {
       User j = db.user(userName);
       …
}
```

is better than writing

```
User j = db.user(userName);
if (j != null) {
       …
}
```

I can also think of at least two reasons why the 1st example is *worse.* The first is redundancy and, yes, maintainability: the userName variable is used twice; when you modify the program (e.g., user name has to be replaced with a user id at a later point), you have to update the program at two places. It's trivial in a toy example like this, but it can be easy to overlook when the check and use are farther apart. The second reason is correctness: the first snippet has an obvious race condition when used with a concurrent database. Returning null is both simpler – the code is shorter and we don't need a transaction – *and* correct.

On annotations

In section 1.1.1 the author tells us that we can think of a class as "a factory of objects". In chapter 2.6.1 he criticizes annotations as "bad" because they are applied to the class, so the objects of the class are manipulated without them knowing it. But of course! With annotations we are modifying the *factory itself* and instructing it to create objects that behave differently than they otherwise would. Instead of manually writing boilerplate code, we use annotations to automate the process.

A funny example

Throughout the text, the author reiterates his point that objects should "do" instead of being "done to". Section 2.7 contains the following little gem:

```
Employee jeff = department.employee("jeff");
jeff.giveRaise(new Cash("$5000"));
if (jeff.performance() < 3.5)
       jeff.fire();
```

Now, why would somebody give himself a low performance score, and subsequently also fire himself? Can you imagine this happening in the real world? No, in the real world – that OO is supposed to simulate

(they have a pen at home, not with them). On the other hand, asking something very specific like "Do you have a pen in your pocket?" could provoke a yes/no answer, but also a false negative (the pen is in the person's back-pack).

– objects (and humans!) *both do and are being done to*. You don't fire yourself,[13] your boss fires you. This is just another example of a joint behavior.

Technical errors

The text is full of various technical errors, which I think is inexcusable given the price of the books. A short, incomplete summary in random order:

- The author writes that the doubly-recursive calculation of Fibonacci numbers has linear ($O(n)$) complexity, whereas it's actually *exponential*.
- C++ examples are written in an extremely bad C++ style (e.g., using `calloc` instead of `new`).
- LISP is repeatedly used as an example of a declarative language, but LISP is anything but declarative: it's eagerly evaluated and very imperative. In fact, it offers all the features that the author advocates *against*, and that proficient LISP programmers cite as its *strengths*: reflection, manipulating code as data (macros), annotations, and more.
- The first page (before ToC) of volume 1 **misappropriates** the quote about lobotomy to David West. The quote is not **by** West, it is **criticized** by him; here's the full context from the book:

  > "Many of the original proponents of the object ideas overdramatized the need for a mental change. ("Step one […] is a lobotomy.") Predictably, this alienated potential converts to the new ideas more often than it convinced them."

[13] Though exceptions happen: in the company where I had worked, and which got dissolved, the CEO *did* formally fire himself.