Booleans Are a Trap

BY PAWEŁ ŚWIĄTKOWSKI
09 Nov 2024

As developers, we love our booleans. They map perfectly into how computers work at a low level and play nicely with `if` statements, our primary control-flow tool. They are simple to reason about. What's not to like?

We actually like them so much that we use them for domain modeling. And that's where things get problematic. I would like to show you some examples of a mess we land in because of using booleans and to offer a better approach.

Expectations vs Reality

Domain modeling is a core responsibility of software engineers. It has many definitions, but in short it is taking some real-world problem and representing it using code, databases, network calls etc. If there's one thing we know about the "real world", it's that it's complex, messy, sometimes unpredictable and unfitting our nice deterministic models. On top of that there is one thing we know for sure about requirements in software development: they *change*.

How does this relate to booleans? Let's say that, for some reason, you have to model a door. Seems straightforward, right?

```
ass Door {
public isOpen: boolean;
```

Wow, was that easy? You will likely have methods like `open` and `close` on top of that, but generally there's nothing complicated here. It was really nice and we used boolean exactly as intended - the door can either be open or closed. What else?

You happily committed the code and few weeks down a great discovery comes. One of your customers suggested, that their doors aren't just opened or closed. They also have a lock on them! The product owner added a ticket for you to implement that. Fortunately, that's still easy.

```
ass Door {
public isOpen: boolean;
public isLocked: boolean;
```

We used boolean again, as intended (door could be locked or not, what else?), the ticket is done[1].

Let me stop you right here. Let's look at our model again. It has two properties. Each of the properties can have two values. Quick math tells us that our door can be in one of four computed states. Wait, four?

- closed and locked
- closed and unlocked
- open and unlocked
- open and locked

Wait—that last state doesn't make sense. With a real door, you can technically turn the key while it's open, but does that meaningfully change its state? Yet our model allows this impossible combination. I can assure you that at some point you will inevitably have an object where `isOpen: true, isLocked: true`. Even if you are careful and the code does not allow the official path to get there, perhaps you back your doors by a database and someone recklessly added a migration setting all the doors in a specific building to `open`?

Since this *will* happen, you need the code to be ready for it. You should have tests for that situation. This all adds code, adds complexity, add time to running the test suite (even if milliseconds) etc.

Beyond the Door

Okay, but it was a stupid example. Easy to dismiss.

Let me tell you a little story based on my actual professional experience. The project was (still is) a B2B SaaS. The main entity was a `Company`. We didn't really have a pricing. All the contracts were individually negotiated outside of the system. The only representation **in** the system was `isPaying: boolean`. Some companies were still evaluating, i.e. not paying, some were already contracted. This worked well, as the fact of paying or not was the only differentiator, there was no standard trial time or anything. For paying companies some extra features were enabled, invoices were shown etc. Simple boolean saved the day.

A few quarters later a new business situation appeared. We now had "partner companies": they were getting a full package, but for free - or rather, in exchange for non-monetary service, like featuring our company somewhere or offering their services for free to us. Since they were not paying, showing invoices section was misleading. Our original boolean no longer sufficed, so we added another: `isPartner: boolean`.

As you can probably see, a similar situation happened as with our doors. The partner was never paying, yet the system allows such combination on booleans and we needed to handle it, otherwise risking nasty exceptions. But the story did not end there...

With time we got more additions. For example: AI features. Some contracts got them, some did not, but they were never allowed during the trial. So we got another boolean: `isAIEnabled`, leading to impossible state of non-paying company with AI enabled. The list went on and on, at some point I counted 12 different boolean flags. That's 4096 possibilities, out of which maybe 20 were valid. All this because we initially bet on a simple boolean and nobody pulled a brake on adding new when the time was right.

A Better Approach

If I convinced you that using boolean might lead to trouble, you might ask yourself "what is this guy's solution?". Indeed, I have one. It's enums and enum sets.

Let's start the show with fixing the door situation. Consider this:

```
um DoorState {
Open,
Closed,


ass Door {
public state: DoorState;
```

Might seem like a bit of overengineering if you haven't heard the story, but you did. So you can probably guess that adding `Locked` to `DoorState` will be our next step, leaving us – correctly – with three possible states of the door.

With the companies we can have similar solution for the contract state.

```
um ContractStatus {
Trial,
Paying,
Partner,


ass Company {
public contractStatus: ContractStatus;
```

With this, for features available in a full package we would check if the status is in one of `Paying, Partner`, but for showing invoicing page we would just check for `Paying`.

This way we fixed the first problem, but what about extra-extra-features, available for paying companies, but not all? We need another enum and a set.

```
um PremiumFeature {
Ai,
ApiAccess,
Sso,
SalesforceIntegration,
// ...


ass Company {
public premiumFeatures: Set<PremumFeature>;
```

An avid reader probably noticed that combining the two still leads to unreachable states. And that's true. It is sometimes really hard to avoid. But we could reason about it that there is far less states in general than 4096 and that only one state is unreachable: the one when company is in `Trial` and has any of the premium features in the set. This is essentially just one case to handle, much easier to test and guard against. I would definitely argue that it's better than an explosion of booleans.

Are Booleans Inherently Evil?

After reading this you might wonder if you should completely avoid booleans, because they are just bad. The answer is: of course not! Booleans absolutely have their place. My suggestions is to keep booleans usage to technical parts, not to business logic or domain modeling.

For example, we mentioned sets above. Let's say you write your own implementation of a set. You will need a `hasElement` method on it. Could you use enums here as a return value? For sure. But actually using boolean here would be completely fine and more natural. It's not like you can get a third value, like "maybe" or "most likely". This is a lower-level technical concept and boolean is absolutely fine here.

Bonus: Foundation for State Machines

When you figure our your possible states with an enum, it's a great foundation to use another useful tool: a state machine. In case of our door it could look like this:

```
from    | event  | result  |
------- | ------ | ------- |
open    | close  | closed  |
closed  | lock   | locked  |
closed  | open   | open    |
locked  | unlock | closed  |
```

I have to say that sometimes I have my reservations about using state machines for business logic. After all in the end business demands all transitions between every step possible - because people make mistake and there needs to be a way to fix them. But if you want to use them, starting with enums will be much easier than starting with a bunch of boolean flags.

Closing note

"The boolean trap" is just one example of how seemingly simple modeling decisions can have unexpected consequences as systems grow. While booleans are perfect for their intended purpose – representing `true/false` technical states – they often fall short when applied to a domain. By choosing enums and enum sets we create code that is better prepared for the "real world"' of the problem domain. Sometimes it's better to rethink how we represent the state instead of blindly adding another boolean flag.

END OF THE ARTICLE

Tags:

This article was written by me – **Paweł Świątkowski** – on **09 Nov 2024**. I'm on Fediverse (Ruby-flavoured account, Elixir-flavoured account). Let's talk.

Loading comments…