PATRICK STEVENS

# IN PRAISE OF `--DRY-RUN`

*Cross-posted at the G-Research official blog; comments at Hacker News.*

One part of my job as a software engineer is writing tools.

Something I always want to see in a tool which does anything non-trivial is a `--dry-run` mode. To be able to know what you're about to do, before you do it, is a great and wondrous thing, helpful to the novice and the experienced user alike. But it's sadly often the case that a tool's `--dry-run` is an unloved second-class citizen, prone to drifting out of date or failing to provide all the information you need.

There's a simple method that keeps you honest, ensures that your `--dry-run` has all the information the user wants to see, and (an unexpected bonus!) helps make the entire tool more maintainable: use the `--dry-run` code path as an input to the "execute" code path.

## The idea

If you have a separate `--dry-run` code path which is to any extent independent of the main execution flow, then you've immediately opted into the huge problem that documentation also tends to suffer: there's now a way to get your explanations out of sync with what actually happens.

So how do we solve this in general? The best answer the world has seen so far is that we need a machine to somehow check whether we've stayed in sync.

### Synchronisation through testing?

One way a machine can check things by testing. For example, the Wolfram Language is documented almost entirely through interactive notebooks written in the Wolfram Language itself, and the documentation itself forms a huge test suite that is evaluated as part of the release pipeline, ensuring that a large number of documentation problems actually turn automatically into *blockers* for release.

### Synchronisation through types

But in many languages (such as F#, the language I use most and the language this post is written in), we have a simpler and more lightweight way to machine-check things: we have the type system. We can use the type system to keep our `--dry-run` and our "execute" flow in sync, by *artificially introducing an API boundary down the middle of*

*our tool*: a boundary which the compiler can check.

The `--dry-run` mode produces a strongly-typed output which is essentially a declaration of what actions we want to carry out. The "execute" mode consumes the `--dry-run` output as its own input.

This way, we're forced to stay relatively honest. If the "execute" flow changes to require different information, you have to make sure it gets that information in the first place, and the easiest way you can do that is by making sure `--dry-run` outputs it.

# An explicit example

Let's take an incredibly simple tool: a command-line utility we'll call `delete`, which takes a collection of globs and deletes every file matching one or more of those globs. Obviously this isn't something you'd write a tool for in practice, since `rm` exists, but it will demonstrate the point.

In F#, one way you might do this could be:

```
type Glob = Glob of string
module Glob =
    let paths (Glob g) : FileInfo list = failwith "(implementation here)"

let deleteFiles (paths : Glob list) : unit =
    paths
    |> List.iter (fun glob ->
        Glob.paths glob
        |> List.iter (fun fileInfo -> fileInfo.Delete ())
    )
```

Now, there are a couple of ways you could make a `--dry-run` for this tool. One way (which is perfectly fine but is not what I'm trying to show here) would be to inject a dependency:

```
let deleteFiles<'ret> (toDo : FileInfo -> 'ret) (paths : Glob list) : 'ret list =
    paths
    |> List.collect (fun glob ->
        Glob.paths glob
        |> List.map toDo
    )
```

Then you'd implement `--dry-run` by passing some variant on `println` as the `toDo` parameter, and you'd implement the real thing by passing `fun fileInfo -> fileInfo.Delete ()`.

But another way to do it, demonstrating the API-boundary technique, is to split up `deleteFiles` into a `gather : Glob list -> FileInfo list`, and an `execute : FileInfo list -> unit`:

```
let gather (paths : Glob list) : FileInfo list =
    paths
    |> List.collect Glob.paths

let execute (files : FileInfo list) : unit =
    files
    |> List.iter (fun fileInfo -> fileInfo.Delete ())
```

In this particular example, there's not much difference between the two styles. But as the tool gets bigger and does more things which are less trivial, it becomes more and more attractive to use the second style and bundle up the `execute` logic together. If nothing else, we want to avoid duplicating work, and the first (dependency-injection) style doesn't make it obvious how.

Of course, there's a judgement you need to make about where to draw the boundary: what is "information retrieval"

and what is "execution"? In this example, `execute` is just a sequence of `doSomething` applied to various parts of its input, but in more complex cases, it's up to you exactly what you consider to be "input" and what you consider merely to be "intermediate processing during execution". But the point is that there's *some* hard boundary drawn between the "find out what you're going to do" phase and the "do it" phase.

## Finishing the example

In real life, you're probably doing something a bit less trivial than just deleting a bunch of files, so you'll have a correspondingly more complex domain model. So naturally you'd introduce a type for the instructions which we previously had implicitly as a `FileInfo list`.

```
type Instructions =
    {
        PathsToDelete : FileInfo list
    }

module Instructions =
    let toString (instructions : Instructions) : string = failwith "(implementation here
```

Then your main program is simply a matter of the following:

```
type DryRunMode = | Dry | Wet

module ArgParse =
    let extractGlobs (args : string[]) : Glob [] =
        failwith "(implementation here)"
    let dryRun (args : string[]) : DryRunMode =
        failwith "(implementation here)"

let main (args : string[]) : int =
    let inputGlobs = ArgParse.parseInputGlobs args
    let instructions = gather inputGlobs

    match ArgParse.dryRun args with
    | DryRunMode.Dry ->
        printfn "Would execute the following:"
        printfn "%s" (Instructions.toString instructions)
        printfn "Halting due to --dry-run."
        0
    | DryRunMode.Wet ->
        printfn "Executing the following:"
        printfn "%s" (Instructions.toString instructions)
        match execute instructions with
        | Result.Error err ->
            eprintfn "%O" err
            1
        | Result.Ok output ->
            printfn "%O" output
            0
```

(Of course, you probably want to be using some more resilient logging than merely `printfn`, you want to be using an actual argument parser rather than my hand-rolled `ArgParse` module, you probably want different return codes based on the different error cases, and so on.)

# An additional benefit: librarification

This "work out what you're going to do, and then do it" idiom has an extra architectural benefit: it really pushes you to make your tool's functionality into a library. If you've got a function which works out what to do, and a function which does it, then you've got two ready-made units of compilation to pull out into the library; additionally (in a statically-typed language) you've got some ready-made types (the `ProgramInstructions`) which serve as places from which to hang docstrings.

In general, a tool which can be used as a library is much more helpful than a tool which can only be used from the command line, and I try to do this with every tool I write - sometimes unlocking unexpected use cases months down the line, when I look at a problem and think "Oh, I've solved that problem already in this other tool - let me just import it"!

If I can import the "work out what to do" component of my tool, then even better, because now I'm not restricted to solving only the problem I originally wrote the tool to deal with. Instead, my tool's library can help me solve any problem for which it's useful to know what the tool knows.

By the way, turning a tool into a library is really helpful whatever language you're using. For example, in Python, there are (often) no static types, but it's still great to be able to import the logic of a tool, even if you're not getting any additional static guarantees by making the `gather`/`execute` split.

# Why the technique works especially well in F#

F# is a strongly- and statically-typed language, so the technique works particularly well here. The API boundary down the middle of our tool is necessarily expressed in terms of some static type: the `--dry-run` mode is really a function like `gather : ProgramArguments -> ProgramInstructions`, and the "execute" mode is a function like `execute : ProgramInstructions -> Result<_, _>`.

We have been forced to model our domain explicitly, by reifying the `ProgramInstructions` type which might otherwise have been implicit (if the tool were all knitted together as a single `doIt : ProgramArguments -> Result<_, _>`).

This is very similar to the idea of "defunctionalisation", by the way, which we've discussed before (see the canonical post for a comprehensive introduction). We get the same benefits, and our code is:

- more readable, because of the explicit domain modelling;
- more easily testable, because we can now test the "information discovery" process independently of the "program execution" process;
- more amenable to optimisation, since (for example) the "execute" phase might be able to realise it can pipeline together two of the things it's being asked to do. It's easier to make this kind of deduction when all your instructions are laid out explicitly.

# Case study

All three of these benefits played out recently for me personally. Some months before, I'd written an internal tool (now used multiple times a day by dozens of developers and quants, as well as at least one automated pipeline) using these principles, and it subsequently came to light that one of the tool's core dependencies actually couldn't do what it said on the tin safely. To work around this, I had to completely rewrite the part of the tool that used this dependency in that particular unsafe way.

But because I'd used the API-boundary technique to structure the tool:

- The tool's statement of intent (the `--dry-run`) was completely unchanged, so immediately the scope of the rewrite was cut by about a third: it was obvious from the type signatures that I wouldn't need to touch the `gather : ProgramArguments -> ProgramInstructions` phase of the program at all.
- A whole raft of tests didn't need to change, because they were testing the `gather` function rather than the `execute` function. It was an unbelievable relief to know that I still had a really solid part of the system to rely on.
- The rewrite was simpler, because of the additional constraints imposed by the fact that my rewritten `execute : ProgramInstructions -> Result<_, _>` could only use the `ProgramInstructions` it was given. I'd essentially already done the work months ago to shrink down the space of possibilities so that the shape of the rewrite was very nearly forced.
- It was extremely easy to test the new `execute` functionality. In some cases, I didn't actually have to go through the effort of setting up all the preconditions for my test cases, because the pre-existing tests for `gather` still proved we could correctly create `ProgramInstructions`. Instead, sometimes, I could simply manually construct a `ProgramInstructions` that would test the `execute` paths, without having to set up the environment and go through the `gather` step first.

- I was also able to optimise away some of the possible instructions by noticing when our core dependency gave us the tools to do some subset of them all in one operation. I may very well not have noticed this if the domain model wasn't sitting there in front of me.

# Conclusion

A `--dry-run` mode is really helpful for the new user who is playing around with your tool, as well as the experienced user who is trying to explore some edge case or who is unsure if their environment is right for the tool to do what they expect. You can help make your `--dry-run` truly meaningful by *depending* on it. This leads to a more modular architecture, with all its benefits.

Published: 2021-05-21
Last modified: 2021-05-24 20:49
2058 words

programming g-research