The problem

We were working on a piece of code that orchestrated several side-effects, including on third-party systems. The entire operation needed to happen atomically; it was all or nothing.

Here's what the method looked like:

```
def complex_atomic_operation
  api_1.setup!
  api_2.setup_with_high_chance_of_failing!
  model.save!
end
```

A database transaction could handle the atomicity on our end, but it was off the table because third-party systems could be left in an invalid and unexpected state. We also had *several* other actions that contained this same orchestration problem. I had an idea...

The solution

The idea was: The Rollbacker. It was a tiny class that knew how to rewind itself. It looked something like this:

```
class Rollbacker
  def initialize
    @blocks = []
  end

  def <<(block)
    @blocks << block
  end

  def rollback
    @blocks.reverse.each(&:call)
  end
end
```

And we could use it in our complex atomic operation:

```
def complex_atomic_operation(rollbacker)
  rollbacker << -> { api_1.rollback_setup! }
  api_1.setup!

  rollbacker << -> { api_2.rollback_setup! }
  api_2.setup_with_high_chance_of_failing!

  rollbacker << -> { model.unsave! }
  model.save!
end
```

The caller could now catch whatever went wrong in the operation and roll back as needed. Importantly, it would only roll back the operations that already made changes:

```
def caller
  rollbacker = Rollbacker.new

  complex_atomic_operation(rollbacker)
rescue SomethingWentWrong
  rollbacker.rollback
  raise
end
```

This could even scale out to handle a "chain" of complex operations, still only rolling back things the side-effects that already completed:

```
def caller
  rollbacker = Rollbacker.new

  complex_atomic_operation(rollbacker)
  complex_atomic_operation(rollbacker)
  complex_atomic_operation(rollbacker)
rescue SomethingWentWrong
  rollbacker.rollback
  raise
end
```

I'm a big fan of the Null Object pattern, so we could simply pass a `NoOpRollbacker` in to test objects if we wanted:

```
class NoOpRollbacker < Rollbacker
  def add; end
  def rollback; end
end
```

Of course, this isn't a perfect solution. There are trade-offs:

- rollbacks could also fail.
- it's a synchronous operation.
- we have double the functionality to maintain.

Honestly, the premise of the Problem feels a bit like a code smell; it's a sign that there is a probably a better way to organize the system.

BUT... given the timeline, minimal complexity, and rare chance of failures, it was a great solution that fit our needs at the time.