# Testing, Part 2: Effects (UNC COMP 523)

Jeff Terrell

2019-10-21

# A motivating example

## creating a Javascript module index

# goal

Generate a javascript module index of *.js files under a root directory
Example:

```
module.exports = {
  AsyncCommand = require('./src/commands/AsyncCommand')
  Command = require('./src/commands/Command'),
  ...
}
```

# what's wrong with this code? (1/2)

```
const getSourceFiles = () => {
  const findCommand = 'find src -name \\*.js'
  const stdout = child_process.execSync(findCommand)
  const lines = stdout.split(os.EOL)
  const nonEmptyLines = lines.filter(line => line)
  const sourceFiles =
        nonEmptyLines.filter(line => !line.match(/spec/))
  return sourceFiles.sort()
}
```

# what's wrong with this code? (2/2)

```
const stream = fs.createWriteStream('index.js', /* options */)
stream.once('open', () => {
  stream.write('module.exports = {\n')
  getSourceFiles().forEach(filename => {
    const exportedSymbol =
          path.basename(filename, '.js')
    const relativePathSansExtension =
          filename.slice(0, -3)
    const requireStatement =
          `require('./${relativePathSansExtension}),'
    stream.write(` ${exportedSymbol} = ${requireStatement},\n`)
  })
  stream.write('}\n')
  stream.end()
})
```

# what's wrong with this code?

How do you test that index generation is successful?

- stub out fs.createWriteStream, to:
  - return an object, which:
  - has once, write, and end methods, and
  - tracks what is passed to its write method

In short: it's hard to test! (And it doesn't need to be.)

# my thesis

Most difficulties with testing are caused by poorly managed *effects*.

# Effects

## what are effects?

# what is an effect? (1/2)

an effect is basically anything *destructive*

- to the contents of a memory cell
- to the screen, e.g. print statements
- to the contents of a file
- to the document of a web page
- to a row in the database
- to the *value* of a *variable*

# what is an effect? (2/2)

an effect is anything that queries global state

- querying a database
- reading from a file system
- reading from the document of a web page
- using the value of a variable not passed in as an argument

# effects are *necessary*

You can't get any work done without effects!

# effects in the example

```
// const stream = fs.createWriteStream('index.js', /* option
// stream.once('open', () => {
  stream.write('module.exports = {\n')
  getSourceFiles().forEach(filename => {
    // const exportedSymbol = path.basename(filename, '.js')
    // const relativePathSansExtension = filename.slice(0, -
    // const requireStatement = `require('./${relativePathSa
    stream.write(` ${exportedSymbol} = ${requireStatement},\
  // })
  stream.write('}\n')
  stream.end()
// })
```

# Effects

## strategies for managing effects

# a caveat

- remember red, green, refactor?
- focus on working code, not simplicity, to start with
- scatter your effects around at first (e.g. printf debugging)
- then refactor and focus on effects
- "effects mindfulness" will become second nature in time

# identifying effects

- is it a known effectful function or operator (e.g. `console.log`)?
- is it clearly affecting an existing value (e.g. assigning to an index of an array)?
- is there a returned value that isn't merely discarded?
- *statements* tend to be effectful; *expressions* tend not to be

# identifying effects: a quiz

1. `const squares = [1,2,3,4,5].map(x => x * x)`
2. `myObject.setField(5)`
3. `return theResult`
4. `print theResult`
5. `myArray.push(7)`
6. `someFunction(1, 2, 3)`
7. `myMap['key'] = newValue`
8. `const result = someFunction(1, 2, 3)`

# strategy 1: move effects to the beginning and end

- effectful utility functions
  - hard to test
  - but broad usefulness
  - often provided by a standard or common library
  - examples from Clojure:
    - `slurp` reads the contents of a file and returns a string
    - `spit` takes a string and dumps it to a file
- ETL: extract, transform, load

# strategy 2: convert external references to be passed as arguments

- usually a fairly simple transformation
- but make sure you're not mutating the passed-in variables–that's an effect!
- you can return a transformed value from the function
- remember that you can return multiple values, e.g. `return [firstThing, secondThing]`
- ultimate result: *functional core, imperative shell*

# strategy 3: use *pure* functions

- what's a pure function?
  - no effects
  - no reading from external state
  - deterministic
  - I/O and randomness are 2 classic impurities
- can a pure function have dependencies?
  - I say yes, if the dependencies are also pure
  - others might disagree
- VIVONE: values in, values out, no effects

# sidebar: what's a value?

- basically anything that's functionally immutable
- a primitive, like numbers and strings
- collections of primitives (so long as they're not mutated)
- collections of collections of primitives, etc.
- in a word: data
- also, functions are values

# sidebar: implications of pure functions

- they're easy to test!
- they're easy to understand!
- they're easy to compose!
- they're easy!
- (because they're simpler than the alternative)

# strategy 4: sanctify thyself

- i.e. make your caller do the dirty work
- you get a function that might be impure, and you just call it at the right time
- (this is actually OK for a pure function to do without compromising purity)
- contrived example: a parameter named `whenGivenArrayIsTooLong`
- Haskell in particular makes an art of this with the `IO` monad

# Effects

## refactoring the example code

# effectful utility function

```
const dumpStringToFile = (file, string) => {
  const stream =
        fs.createWriteStream(file, /* options */)
  stream.once('open', () => {
    stream.write(string)
    stream.end()
  })
}
```

# query external state as the first step

```
const getFilesUnderDirectoryWithExtension = (dir, ext) =>
  child_process.execSync(`find ${dir} -name \\*.${ext}`)
    .split(os.EOL)
    .filter(line => line) // remove empty lines
    .sort()

const getSourceFiles = () =>
  getFilesUnderDirectoryWithExtension('src', 'js')
    .filter(line => !line.match(/spec/)) // remove spec files
```

# the functional core

Notice how testable these functions are!

```
const filenameToRequireLine = filename => {
  const exportedSymbol = path.basename(filename, '.js')
  const relativePathSansExtension = filename.slice(0, -3)
  const requireStatement =
        `require('./${relativePathSansExtension}')`
  return ` ${exportedSymbol} = ${requireStatement},\n`
}

const buildIndexString = filenames =>
  'module.exports = {\n'
  + filenames.map(filenameToRequireLine).join('')
  + '}\n'
```

# the imperative shell

This is simple enough it probably doesn't need to be tested!

```
const main = () =>
  dumpStringToFile('index.js',
                   buildIndexString(getSourceFiles()))

main()
```

# the point, so far

Effects should be a *focal point* of how we approach coding.

# object-oriented programming

## an analysis

# object-oriented programming

- why I don't recommend OOP, in a word: effects
- effects are *everywhere* in OOP code
- this is fundamentally why OOP is more complex and therefore harder
- one way to do OOP and manage effects: immutable "value objects"
  - internal state never changes after constructor
  - any methods that would normally mutate state will instead return a new value object
  - this approach effectively eliminates state (since the internal value never changes, so is decoupled from time)

# exercise

- refactor some Javascript code to manage effects better
- find link to code from `https://comp523.cs.unc.edu/calendar/`