Day 23: The proxy design pattern

The proxy design pattern is yet another wrapper pattern meant to simplify / control access to a class. The example here shows how access to an existing, eagerly-evaluated class (here, one that loads a potentially huge text file) can be wrapped with a proxy to turn that eager evaluation into on-demand, lazy evaluation.

Say we've got the below interface and expensive, existing implementation:

```
trait FileReader {
  def readFileContents(): String
}

// read `filename` on instantiation and cache it for fetching on [[readFileContents]
class FileReaderReal(filename: String) extends FileReader {
  val contents = {
    val stream = this.getClass.getResourceAsStream(filename)
    val reader = new BufferedReader(new InputStreamReader(stream))
    try {
      reader.lines().iterator().asScala.mkString(System.getProperty("line.separator"))
    } finally {
      reader.close()
      stream.close()
    }
  }

  System.out.println(s"Finished reading the actual file: $filename")

  override def readFileContents(): String = contents
}
```

Now let's add a **proxy** implementation to turn this from eager to lazy:

```
class FileReaderProxy(filename: String) extends FileReader {
  private var fileReader: FileReaderReal = null

  override def readFileContents(): String = {
    if (fileReader == null) {
      fileReader = new FileReaderReal(filename)
    }
    fileReader.readFileContents()
  }
}
```

What have we achieved here? Not a huge amount that couldn't have been done in the original implementation. `contents` could've just been a lazy val, which, heck, if you look at the Scala implementation of `lazy val`, you'll see the pretty immediate similarity:

```
final class LazyCell {
  @volatile var bitmap_0: Boolean = false
  var value_0: Int = _
  private def value_lzycompute(): Int = {
    this.synchronized {
      if (!bitmap_0) {
        value_0 = <RHS>
        bitmap_0 = true
      }
    }
    value_0
  }
  def value = if(bitmap_0) value_0 else value_lzycompute()
}
```

This is a bit fancier and accounts for concurrency on the JVM properly, but it's essentially the same "if not initialized, initialize and return; otherwise return" as the proxy example here.

But we can imagine other use cases, maybe.

- If we can't modify the original implementation for whatever reason (compatibility, breaking change, not our library, etc.), the proxy pattern is allowing us to modify behaviors without modifying the original class.
- This wrapper might've also brought together disparate components, but then we're pretty close to the facade design pattern.
- This also feels similar to the bridge design pattern, but where the bridge pattern allows for deconstructing of many implementations of a component, and the facade pattern allows for the joining of multiple separate components, this pattern is mostly just to wrap and change the behavior of an existing component.
- It was couched as similar to the decorator pattern, and it definitely is. Just not changing methods this time.