Walter Blair
603 Assignment 4

Command & Adapter Patterns

I got myself into quite a quagmire with this assignment, but I think I made sense of it! The first time I heard the term adapter pattern, I thought of cannoli. Cannoli is open-source software created by the UC Berkeley's AMPLab that fits into their universe of genomics processing tools. Their whole processing platform is based on Apache Spark which I looked at in my first pattern assignment. The anchor of their whole platform is the ADAM framework. ADAM is composed of several keystone processing algorithms that they have reimplemented in a native Spark-ready way as well as some specialized and highly efficient data formats à la Parquet and Avro. This whole ADAM universe is kind of a big deal, and cannoli is a brand new component that is designed to wrap legacy tools that have not been reimplemented in a Spark-native way for use in the Spark-ready ADAM framework.

So what cannoli does is adapt command-line legacy tools that run as stand-alone applications for the in-memory distributed processing of ADAM's Spark application. I thought that surely this is as clear-cut of an adapter pattern as I could find. Famous last words! My attempts to understand cannoli are summarized in Fig. 1 below.

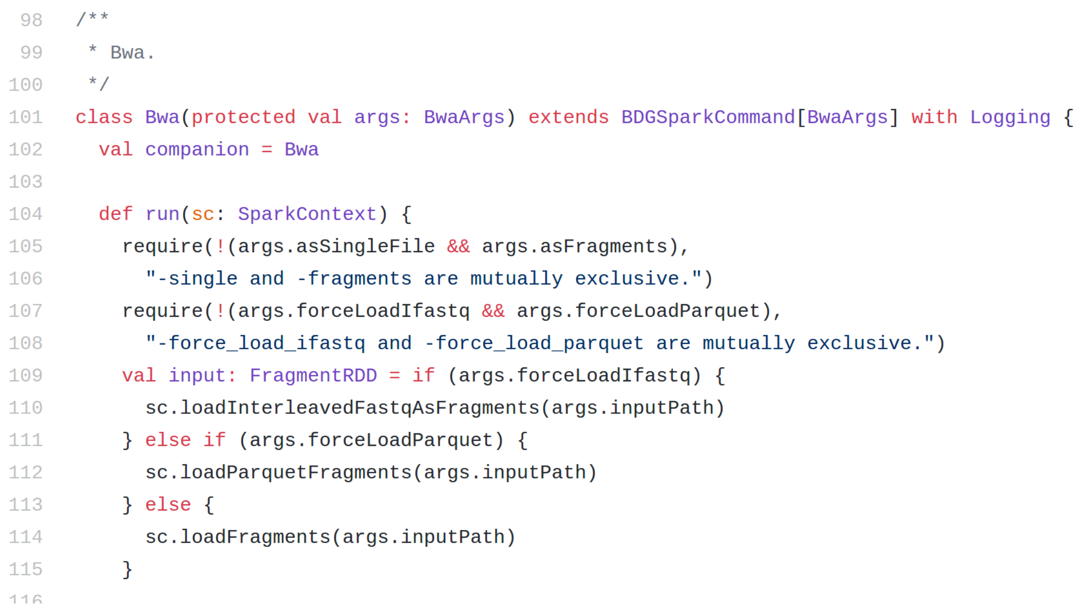
I believe that the adapter pattern is indeed used by cannoli and that the command pattern is a key component of this adapter. The job of cannoli is to adapt any legacy tool to fit into the Spark-ready ADAM framework, so the command pattern emerges as a strategy for parametizing a generic Spark command so that it can wrap any legacy tool we want to use. The cannoli class as the client collects command-line args and creates one of the concrete tool-specific commands that implements the general BDGSparkCommand interface (Snippet 1). The parametized command provides a general Spark-ready action, but the concrete command also contains information on what command line args are needed by the specified legacy tool (Snippet 2).

```scala
trait BDGSparkCommand[A <: Args4jBase] extends BDGCommand with Logging {
  protected val args: A

  def run(sc: SparkContext, job: Job)

  def run() {
    val start = System.nanoTime()
    val conf = new SparkConf().setAppName("adam: " + companion.commandName)
    if (conf.getOption("spark.master").isEmpty) {
      conf.setMaster("local[%d]".format(Runtime.getRuntime.availableProcessors()))
    }
    val sc = new SparkContext(conf)
    val job = HadoopUtil.newJob()
    val metricsListener = initializeMetrics(sc)
    run(sc, job)
    val totalTime = System.nanoTime() - start
    printMetrics(totalTime, metricsListener)
  }
```

Snippet 1. The BDGSparkCommand trait provides a parametized way to execute a Spark job.



```scala
/**
 * Bwa.
 */
class Bwa(protected val args: BwaArgs) extends BDGSparkCommand[BwaArgs] with Logging {
  val companion = Bwa

  def run(sc: SparkContext) {
    require(!(args.asSingleFile && args.asFragments),
      "-single and -fragments are mutually exclusive.")
    require(!(args.forceLoadIfastq && args.forceLoadParquet),
      "-force_load_ifastq and -force_load_parquet are mutually exclusive.")
    val input: FragmentRDD = if (args.forceLoadIfastq) {
      sc.loadInterleavedFastqAsFragments(args.inputPath)
    } else if (args.forceLoadParquet) {
      sc.loadParquetFragments(args.inputPath)
    } else {
      sc.loadFragments(args.inputPath)
    }
```

Snippet 2. Concrete object Bwa extends BDGSparkCommand with specific tool adaptee BwaArgs

Armed with the Spark command provided by the interface as well as the necessary tool-specific args, the concrete command (e.g. Bwa or Freebayes) then calls a specific action on the SparkContext to pipe the Spark dataset into the specified legacy tool using RDD.pipe( ), an implementation of Unix named pipes for Spark datasets. This command pattern sequence as described above is illustrated in Fig. 2.

After untangling the command pattern above, and trying my best to ignore the inclusion of companion objects for each concrete command, which I believe is a standard way to implement the singleton pattern in Scala, I was able to focus on the adapter pattern I was originally looking for. A concrete command like Bwa contains the information it needs to execute a Spark command by implementing the BDGSparkCommand interface, but it also contains the information it needs to run the legacy tool's CLI because it contains as an attribute an instance of the tool-specific class BwaArgs (Snippet 2). BwaArgs implements an interface along with a few mixin traits that allow BwaArgs to fit nicely into the legacy tool's CLI. Bwa contains a BwaArgs object and uses all of this information to adapt the legacy tool to the Spark context, so I would call this an example of the object-adapter pattern.

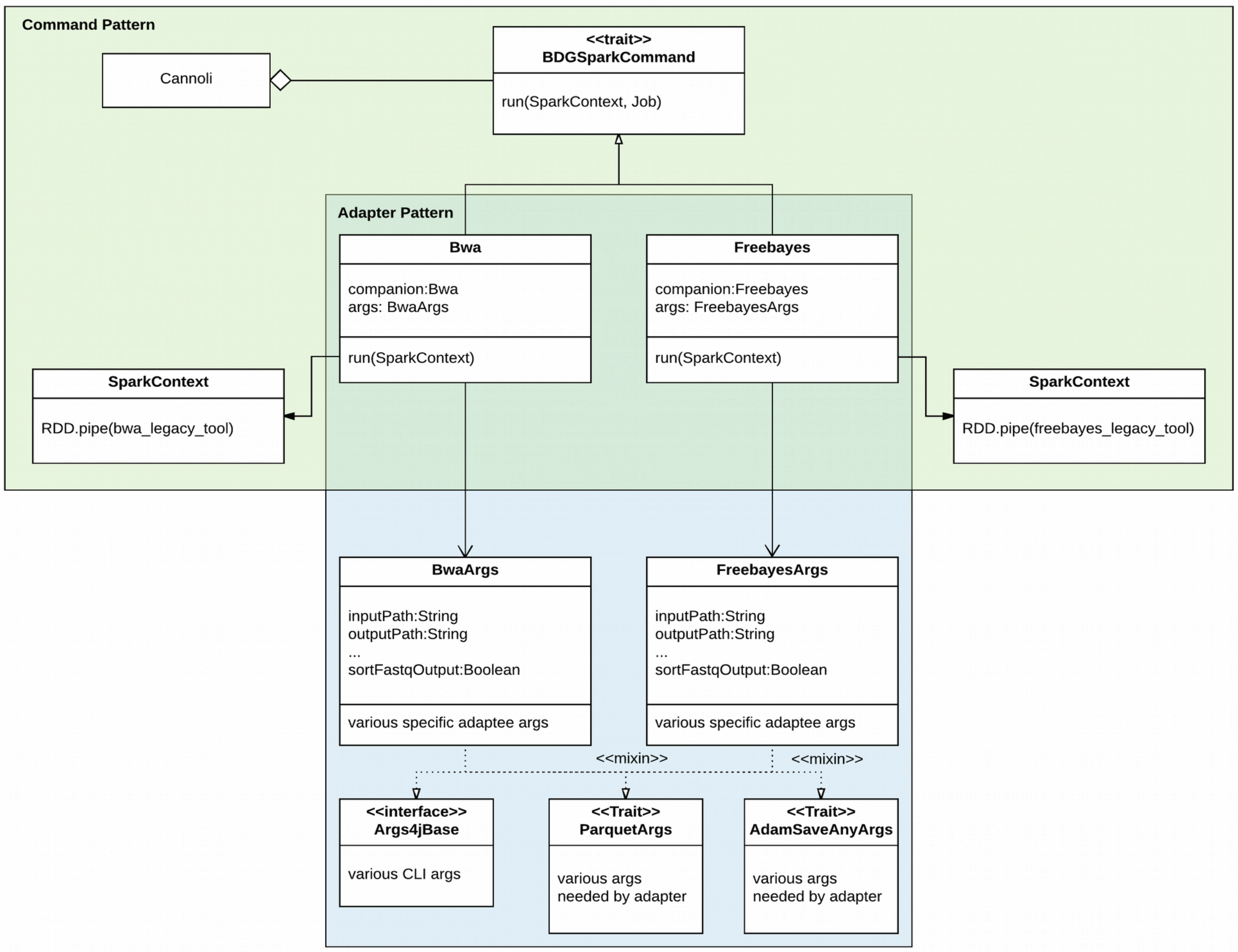


Figure 1. Command pattern and object-adapter pattern diagram of cannoli. Bwa and Freebayes are two examples of concrete commands, each implementing the BDGSparkCommand trait. Each of these concrete commands executes a Spark job that pipes a Spark dataset to the specified legacy tool like bwa or freebayes. Each of these concrete commands contains an object, either BwaArgs or FreebayesArgs, that contains all of the information needed to adapt the command for the CLI of the specified legacy tool.

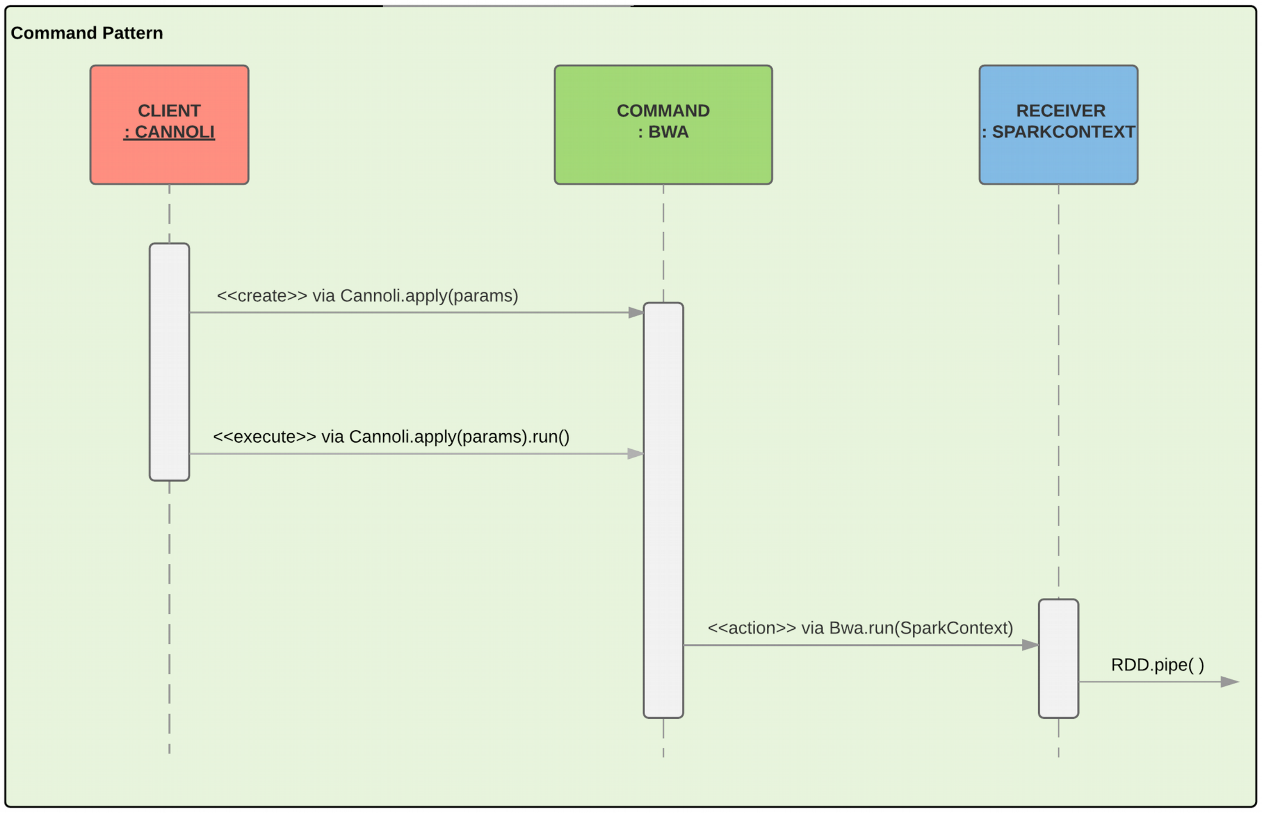

Command Pattern
CLIENT
: CANNOLI
COMMAND
: BWA
RECEIVER
: SPARKCONTEXT
<<create>> via Cannoli.apply(params)
<<execute>> via Cannoli.apply(params).run()
<<action>> via Bwa.run(SparkContext)
RDD.pipe( )

Figure 2. Command pattern sequence diagram. Cannoli parses args and creates a concrete command like Bwa or Freebayes. Cannoli then runs the concrete command that executes a Spark action that pipes the Spark dataset to a particular legacy tool like bwa or freebayes.