My primary design goal for this application is simplicity. It was suggested that this app use the javafx library to implement a GUI using the MVC pattern. I found this to be overkill for an application that's primary objectives are:

1. Load in a .jar file
2. Persist metrics of that jar using an object store (MicroStream)
3. Display those metrics to the user

I think these objectives were best achieved by implementing a simple CLI. As for the rest of the design, I focused on adhering to the SOLID principles. For example, the Single Responsibility Principle, no class in my design has more than ~60 lines of code.

Another aspect in which my design excels, in my opinion, is encapsulation. The JarParser class entirely encapsulates and obfuscates all behavior related to parsing the contents of the jar archive. A client user of this class need only instantiate it and pass a fully qualified jar file name as a string to the parse method and be returned a list of all the classes inside the jar. This encapsulation (and its singular responsibility) means that the JarParser class is extremely reusable.

To critique my design, it could be said that the MetricDB is not as generic as it could be. Perhaps with some abstractions it could be used for storing and querying more than just JarMetric instances.

To run the application, you will need to include a class path to both the MicroStream jars (for object persistence) and a class path to the directory containing the jars you wish to load.

Option 1 in the CLI will wipe any jar metrics that have been loaded into the database and replace them with the new metrics. You will need to pass the fully qualified jar file name. The screenshot below demonstrates the application running.

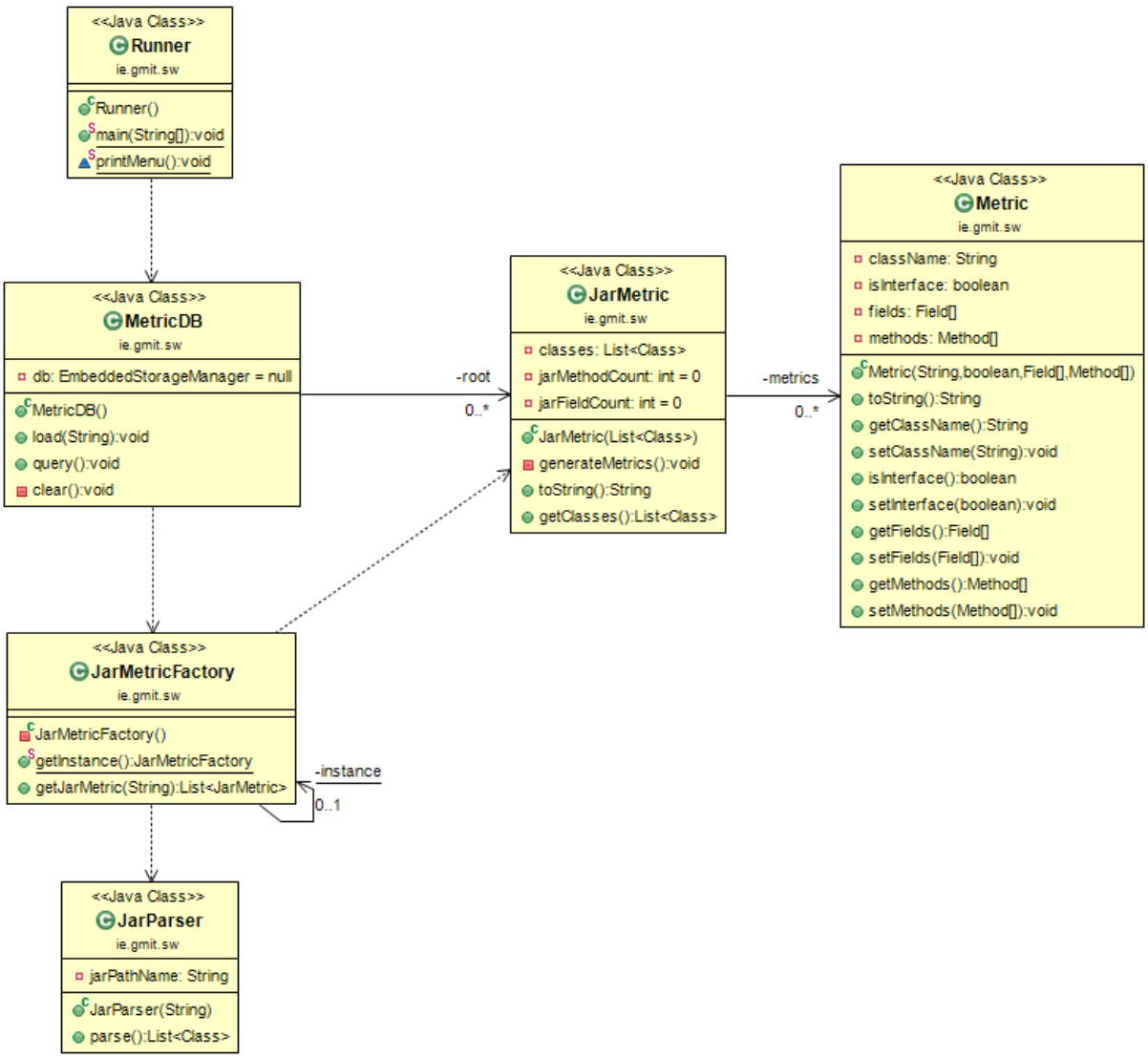
```
C:\Users\Darren\Desktop\G00299944>java -cp .;./lib/microstream/*;./lib/jars/*;./metrics.jar ie.gmit.sw.Runner
--MENU--
1. Load New Jar
2. Jar Metrics
0. Quit
>1
Fully Qualified Jar Name: C:\Users\Darren\Desktop\G00299944\lib\jars\dom.jar
successfully loaded new jar with 62 jar entries...
--MENU--
1. Load New Jar
2. Jar Metrics
0. Quit
>2

Individual Class Metrics: [
org.w3c.dom.Document
Field Count: 18 Method Count: 68,
org.w3c.dom.Node
Field Count: 18 Method Count: 37,
org.w3c.dom.DOMImplementation
Field Count: 0  Method Count: 4,
org.w3c.dom.DocumentType
Field Count: 18 Method Count: 43,
org.w3c.dom.Element
Field Count: 18 Method Count: 57,
org.w3c.dom.DOMException
```

The metrics returned are as follows:

- The name of each class in the jar and the number of fields and methods it has
- The total number classes, fields and methods in the jar altogether.

More jar files can be loaded (this will wipe the currently loaded jar) in by repeating the process, so long as their class path was specified when launching the application.


<<Java Class>>
Runner
ie.gmit.sw
Runner()
main(String[]):void
printMenu():void
<<Java Class>>
MetricDB
ie.gmit.sw
db: EmbeddedStorageManager = null
MetricDB()
load(String):void
query():void
clear():void
-root
0..*
<<Java Class>>
JarMetric
ie.gmit.sw
classes: List<Class>
jarMethodCount: int = 0
jarFieldCount: int = 0
JarMetric(List<Class>)
generateMetrics():void
toString():String
getClasses():List<Class>
-metrics
0..*
<<Java Class>>
Metric
ie.gmit.sw
className: String
isInterface: boolean
fields: Field[]
methods: Method[]
Metric(String,boolean,Field[],Method[])
toString():String
getClassName():String
setClassName(String):void
isInterface():boolean
setInterface(boolean):void
getFields():Field[]
setFields(Field[]):void
getMethods():Method[]
setMethods(Method[]):void
<<Java Class>>
JarMetricFactory
ie.gmit.sw
JarMetricFactory()
getInstance():JarMetricFactory
getJarMetric(String):List<JarMetric>
-instance
0..1
<<Java Class>>
JarParser
ie.gmit.sw
jarPathName: String
JarParser(String)
parse():List<Class>