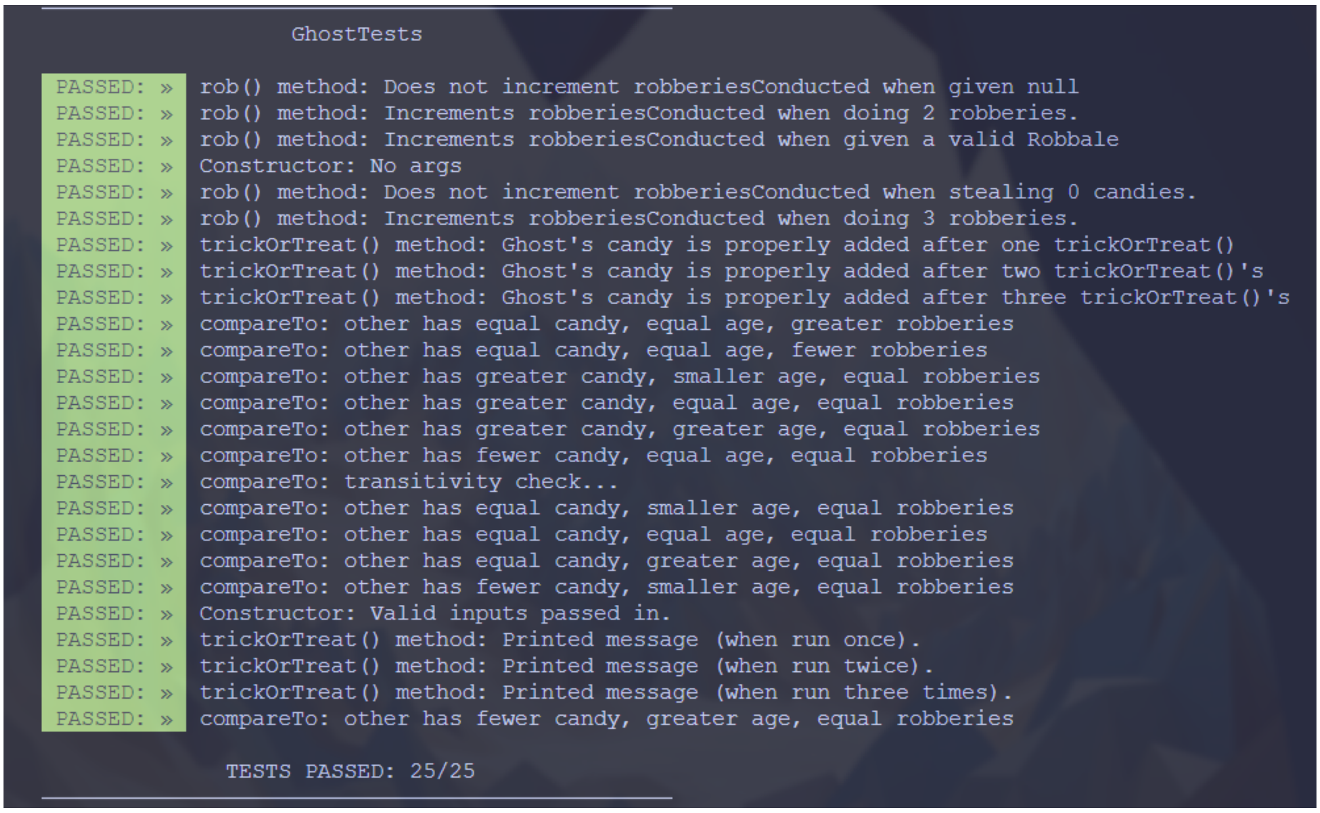
Automated Driver Testing

Main Driver

Written by Ryder Johnson, 11/3/23

Preface

This document is *long*. When I started writing this I really didn't quite grasp how many features and nuances we've built over the last few weeks, but I hope that anyone who was interested in the behind-the-scenes can find a treasure trove of information here. Cheers!



```
                  GhostTests

PASSED: »  rob() method: Does not increment robberiesConducted when given null
PASSED: »  rob() method: Increments robberiesConducted when doing 2 robberies.
PASSED: »  rob() method: Increments robberiesConducted when given a valid Robbale
PASSED: »  Constructor: No args
PASSED: »  rob() method: Does not increment robberiesConducted when stealing 0 candies.
PASSED: »  rob() method: Increments robberiesConducted when doing 3 robberies.
PASSED: »  trickOrTreat() method: Ghost's candy is properly added after one trickOrTreat()
PASSED: »  trickOrTreat() method: Ghost's candy is properly added after two trickOrTreat()'s
PASSED: »  trickOrTreat() method: Ghost's candy is properly added after three trickOrTreat()'s
PASSED: »  compareTo: other has equal candy, equal age, greater robberies
PASSED: »  compareTo: other has equal candy, equal age, fewer robberies
PASSED: »  compareTo: other has greater candy, smaller age, equal robberies
PASSED: »  compareTo: other has greater candy, equal age, equal robberies
PASSED: »  compareTo: other has greater candy, greater age, equal robberies
PASSED: »  compareTo: other has fewer candy, equal age, equal robberies
PASSED: »  compareTo: transitivity check...
PASSED: »  compareTo: other has equal candy, smaller age, equal robberies
PASSED: »  compareTo: other has equal candy, equal age, equal robberies
PASSED: »  compareTo: other has equal candy, greater age, equal robberies
PASSED: »  compareTo: other has fewer candy, smaller age, equal robberies
PASSED: »  Constructor: Valid inputs passed in.
PASSED: »  trickOrTreat() method: Printed message (when run once).
PASSED: »  trickOrTreat() method: Printed message (when run twice).
PASSED: »  trickOrTreat() method: Printed message (when run three times).
PASSED: »  compareTo: other has fewer candy, greater age, equal robberies

             TESTS PASSED: 25/25
```

Introduction

CS1331 Homework assignments can be quite unforgiving. From what feels like hundreds of possible constructors to the many, *many* edge cases, the tests required for each component has exponentially increased with every week--with a desperate need for a streamlined process.

When I initially got together with Justin for HW4, we identified the issues plaguing older iterations of our drivers, and came up with the following restrictions:

- **Portable** - Arguably the largest (and most painful) limitation: everything must be in a single `.java` file. The end user should be able to directly drag the file into their project with no other requirements, leading to the unfortunate complications of:
  - **No more than one file** - EVERYTHING must be contained in a single `.java` file for distribution.
  - **No outside dependencies** - No JUnit, Mockito, Selenium, etc. to use for easy test creation. As seen later on in this article, we essentially wrote our own variant of JUnit.
- **Robust Tests** - Tests must be easy to create, easy to read, and easy to modify if needed.
  - **Feedback-based System** - The tests should notify the developer of specifically *what* they missed, along with the stack trace of where it might have occurred.
  - **Exception Handling** - All sorts of exceptions (some intentional) can be thrown from the end user's code. In order to prevent this from breaking the entire testing environment, these are caught and internally handled.
  - **"No clutter nonsense"** - Tests should be as clear and intuitive to read as possible, requiring a few terminal tricks for formatting.

The problem of portability

The HW5 driver totaled **1984** lines of code. Having a file anywhere near this size is borderline *impossible* to maintain--not to mention it brings even more complications:

What if we had 20+ features where each one would typically have its own class?

If you write to a file and I want to check it against an expected output, what if there was 60+ lines? Do I have to manually write out a String with a billion `\n` characters?

Solving the class problem

This was the very first issue we addressed, and through its iterations has been the sole reason maintaining this project has feasible whatsoever.

The whole point of object oriented programming is in its name: *using objects*. It seems fairly ironic when you're only working with one file and class--right?

...but does it have to be one class?

Let's take a look at the Java specification documentation, specifically §6.6.2:

> **If a class lacks the `public` modifier, access to the class declaration is limited to the package in which it is declared (§6.6). In the example:**

```java
package points;
public class Point {
  public int x, y;
   public void move(int dx, int dy) { x += dx; y += dy; }
}
class PointList {
   Point next, prev;
}
```

two classes are declared in the compilation unit.

Or in other words: **we can put as many non-public classes into one file as we want**.

That's all fine and dandy, but how does that solve dealing with massive driver lengths?

Because we're able to store multiple classes in one file, it *also* means we could work in multiple files our end, and have some sort of **script** automatically squash them into one for distribution.

The problem with imports

If you were to directly append all `.java` files, you would run into 3 issues:

1. **Multiple classes with public visibility** - If we were writing `public class ...` in our individual files, this would result in an error when combining. A solution to this was simply search for the visibility modifier of a class and if it exists, remove it.
2. **Imports not at the top** - Java requires **all** imports to be at the top of a file, where simply appending would result in imports being scattered throughout.
3. **Duplicate imports** - Duplicate imports throw a compiler error, and is unfortunately something to account for. If two classes happened to use ArrayList, two ArrayList imports would show up in the resulting Driver file

To solve the last two issues, the script initially used the following methodology:

```
<RESERVED_IMPORT_SPACE>

<CODE_LINES>
```

1. Pass through the entire `src` directory and glob all `.java` files
2. If the class is marked as `public class`, remove the visibility modifier

3. For each line, if it happens to be an import statement add it to a special reserved import space container
4. If the line happens to be code, add it to the code lines list
5. At the end of parsing all `.java` files, write to `Driver.java` using the schema shown above

This ended up working perfectly! ...for HW4 and HW5.

(as of HW6) Data files

As I mentioned above, the second issue was being able to store relevant `.txt` files on our end to cross reference a test's output, e.g. a `PrintWriter`. Once again, the issue is that we *can't* distribute text files.

Another problem is how we locally run the driver code. We don't use the python script to run the driver when developing it, meaning you have the following scenario:

- The driver needs to have the contents of the `.txt` file--sure, we could dump the contents of it during the python script.
- Running it locally *also* needs to have access to the `.txt` file--making this not quite as simple.

To answer this problem, I have to first introduce you to how tests are written.

But what does a test look like?

Take a look at the following example of a test:

```
    @TestCase(name = "Testing that all 7 genres exist in the correct order...")
    @Tip(description = "\nHow many genres should there be? What order should they be in?")
    public void genresExistInCorrectOrder() throws TestFailedException {
        String[] correctGenres = new String[] {"ACTION", "COMEDY", "FANTASY", "HORROR", "MYSTERY", "ROMANCE", "SCI_FI"};
        Genre[] genres = Genre.values();
        for (int i = 0; i < 7; i++) {
            TestFunction.assertEqual(genres[i].toString(), correctGenres[i]);
        }
    }
```

The following was a check for the `Genre` class implemented in HW5 containing all of the correct genres, in the correct order.

Custom Java Annotations

Remember `@Override`? This is an example of a **Java Annotation**, where either a method, class, or field can be marked with a special "tag" of information for later use during runtime. Here's a brief description of our current annotations:

- `@TestCase` - Used for denoting the following method is a test. Takes a `name` field for describing the intent of the test (we show this to the user)
- `@Tip` - If a test fails, we provide a message providing a potential cause as to why their code might've failed (without getting too specific).

New annotations arriving for HW6:

- `@BeforeTest` - Used for common code ran between all tests in a class (e.g. for static classes we have a reset method run before every test)
- `@AfterTest` - Runs after the test, usually for static classes requiring some sort of cleanup function
- `@InjectData` - Injects data from `.txt` and inserts it into a String. This ended up being a massive change **and is the secret sauce to data files**.

Test Functions

In order to actually **assert** whether some sort of snippet has passed or not, we need some sort of way of comparing a multitude of different values, while using that to determine whether a test has been passed or not.

To do this, we utilize custom Test exceptions thrown when a test fails:

```java
// Strings are different values, throw exception
throw new TestFailedException("Strings different! Received " + actualString
+ " but expected " + expectedString);
```

These are then caught later and processed as either failed or passed tests. More on that later!

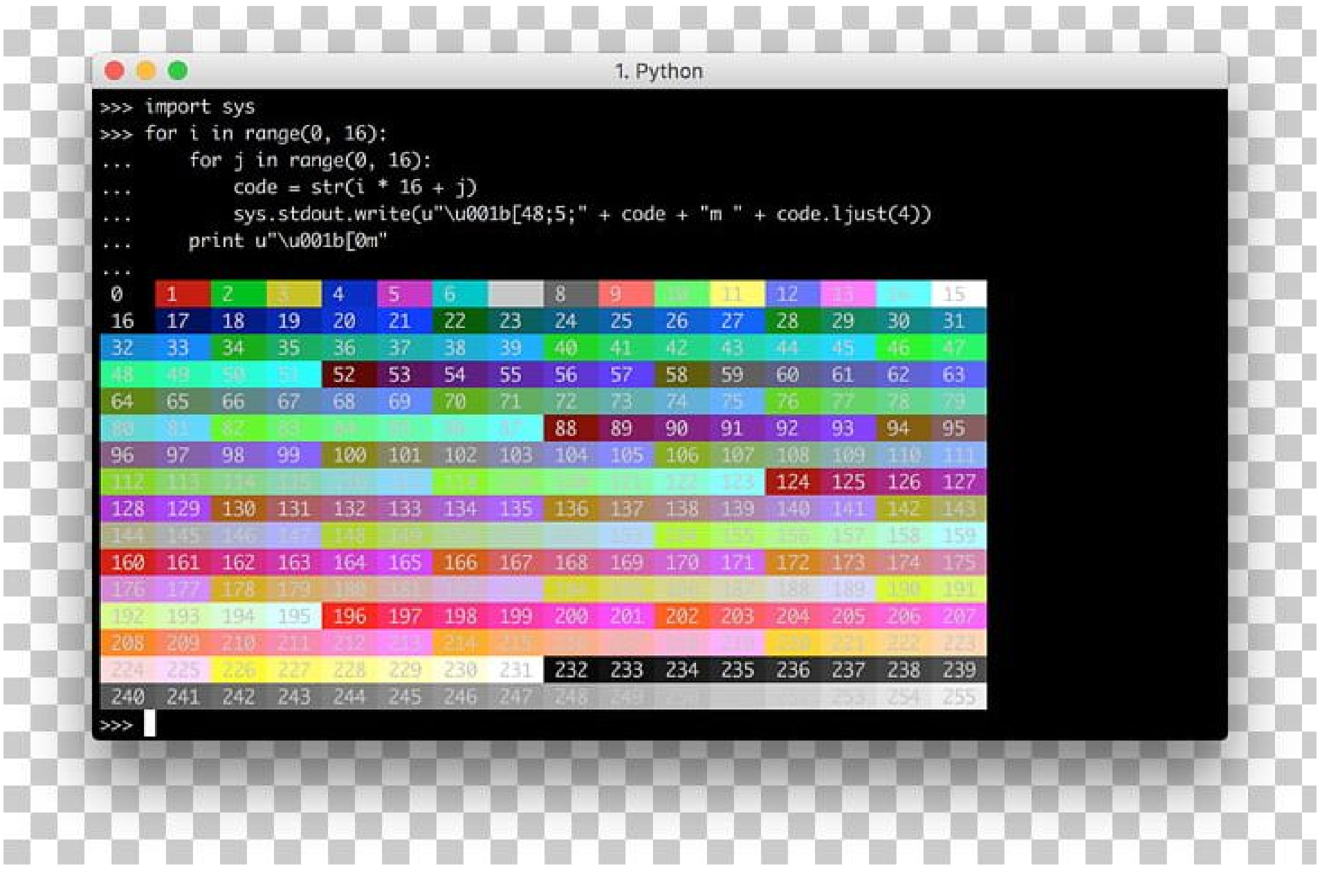
Test black magic, colors, and other neat features

There are some incredibly niche elements the driver does under the hood for formatting the tests, and this section aims to cover a few of them.

Colors

Colors can be *huge* for quickly checking whether a test has failed or not, and your terminal supports them! Somewhat.

By default, most terminals don't use UTF-8 (unicode) formatting. This unfortunately means the symbols we're able to use is significantly limited but we *do* have one tool at our disposal: ASCII escape codes.



```
>>> import sys
>>> for i in range(0, 16):
...     for j in range(0, 16):
...         code = str(i * 16 + j)
...         sys.stdout.write(u"\u001b[48;5;" + code + "m " + code.ljust(4))
...     print u"\u001b[0m"
...
 0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15
 16   17   18   19   20   21   22   23   24   25   26   27   28   29   30   31
 32   33   34   35   36   37   38   39   40   41   42   43   44   45   46   47
 48   49   50   51   52   53   54   55   56   57   58   59   60   61   62   63
 64   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79
 80   81   82   83   84   85   86   87   88   89   90   91   92   93   94   95
 96   97   98   99   100  101  102  103  104  105  106  107  108  109  110  111
 112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127
 128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143
 144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159
 160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175
 176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191
 192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207
 208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223
 224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239
 240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255
>>>
```

This of course looks like jibberish, but the underlining principle is that there are certain unicode symbols (we type them in java using `\u`) we're able to use to output color in the terminal.

Internally we keep a massive list of different unicode colors to use, with a few of them being listed here:

```
    public static final String BLACK_FOREGROUND = "\033[30m";
    public static final String BLACK_BACKGROUND = "\033[40m";
    public static final String RED_FOREGROUND = "\033[31m";
    public static final String RED_BACKGROUND = "\033[41m";
```

Do note that there are separate color codes for the **background** (behind the text) and the **foreground** (the text itself).

We made a nice little function for writing colored text here:

```
        /**
         * Formats a string to have both an ASCII foreground and background
 in terminal
      *
```

```
     * @param background The ASCII representation of the background color,
pulled from AsciiColorCode
     * @param foreground The ASCII representation of the foreground color,
pulled from AsciiColorCode
     * @param s The string to color
     * @return The colored string
     */
    public static String formatColorString(String background, String
foreground, String s) {

        return foreground + background + s + AsciiColorCode.RESET_COLOR;
}
```

IO Hijacking

Another one of the big features we added was the ability to track `System.out.println` statements. This ended up being a utility class of its own labeled `IOHijacker`.

`System.out` is actually an example of a `PrintStream` variable--in which there are several methods it overrides that determine "when they use `println` do this, or when they `printf` do that"

`System.out` can also be changed! Using the incredibly useful `System.setOut(PrintStream);` method we're able to create our own custom PrintStream and do whatever we want with console output.

Here's the custom PrintStream currently implemented:

```
private PrintStream getRedirectedStream() {
        return new PrintStream(System.out, true) {
                @Override
                public void print(String s) {
                        IOHijacker.appendMessage(s);
                }

                @Override
                public PrintStream printf(String message, Object... args) {
            IOHijacker.appendMessage(String.format(message, args));

                        return this;

                }

                @Override
                public void println(String s) {
```

```
                            IOHijacker.appendMessage(s + "\n");
                    }

            };

    }
```

IO Hijacker - Recording Functionality

Because we only want to hijack the System.out when we explicitly want to, the IO Hijacker uses a sort of recording system, where there is both an `enableRecording` to start recording the inputs, and `stopRecording` to stop and retrieve the output as a String.

To allow the IOHijacker to be used throughout all classes, have its own properties, but without any instance duplication, the IOHijacker is a Singleton.

Back to data files

Now that Java Annotations have been roughly explained, we can go back to the implementation of data files.

Example scenario: The user has code that needs to be tested. The code does the following:

1. Counts from 1 to 10 and puts this into a `List<Integer>`
2. Uses a `PrintWriter` to write this to `Numbers.txt`

The main test case for checking whether the user has successfully put in the correct numbers would be retrieving their output using a `Scanner`, and comparing it to the expected values.

However, this can be incredibly cumbersome if we had to write this in a String, similarly to our other test cases:

```
// This is terrible.
String expectedOutput = "1\n2\n3\n4\n5\n6\n7\n8\n9\n10";
```

Instead, what if we had a way of keeping our own `TestData.txt`, and we were able to compare their output to that text file? Recall we have the single file issue, so this would have to be streamlined in some way.

Enter: The new `@InjectData` annotation:

```java
class TxtTestData {
    @InjectData(name = "TestData.txt")
    public static String DATA = "";
}
```

`TxtTestData` is an example of what I now call a **Data class**. A data class is a class explicitly registered to the `TestManager` (we'll get to it, don't worry) that contains one or more data inputs (we might rename `@InjectData` to `@DataInput`, not too sure yet)

The `TestManager` allows a data class to be registered using the following method:

```java
public static void registerDataClasses(Class<?>... classes);
```

During runtime, it does the following:

1. For each data class, scan for all fields with the `InspectData` annotation
2. For each field, enforce public visibility, and start a new Scanner with the path being `name`
3. Loop through and convert the scanner's output to a single string with `\n` delimiter
4. Replace the value of the field with the scanned output--in the example above, `DATA` would be set to whatever contents `TestData.txt` contained
5. Run the tests with the new pulled data

This now allows us to *really* easily compare test cases to a `PrintWriter`:

```java
TestFunction.assertEqual(output, TxtTestData.DATA);
```

where `DATA` would be populated using Reflection (more on that in the Reflection section).

That still doesn't solve the one file issue 2: Electric Boogaloo

...because there's always something else!

The issue is that if we export this to a single `.java` file we quite literally cannot export a `.txt` with it--as that would be violating one one of our restrictions.

Just a tad ago I mentioned this was really bad to write:

```java
// This is terrible.
String expectedOutput = "1\n2\n3\n4\n5\n6\n7\n8\n9\n10";
```

...but what if we ALSO got the python script to do this for us?

```
class TxtTestData {
    @InjectData(name = "TestData.txt") <-- InjectData found with
"TestData.txt", signal injection
    public static String DATA = ""; <-- Injection prepared, fire injection
sequence into DATA
}
```

The process is more or less the following:

1. While parsing each line in a `.java` file, attempt to find a line with the `@InjectData` annotation
2. If the `@InjectData` annotation is found, override default behavior and parse the value between the quotation marks using regular expressions
3. Set the injection path to be the `name` value, in this case being "TestData.txt"
4. On the next line, the system knows the injection path has been populated and attempts to do the following:
    1. Open the specified path -> `TestData.txt`
    2. Pull all lines of data from the txt and combine into one string with `\n` delimiter
    3. Parse the line using regular expressions to retrieve `public static String DATA =`
    4. Append `\" + fileContents \";` to the parsed string
    5. Set the current code line for writing to be the implemented string

This results in the following when converted to a single file:

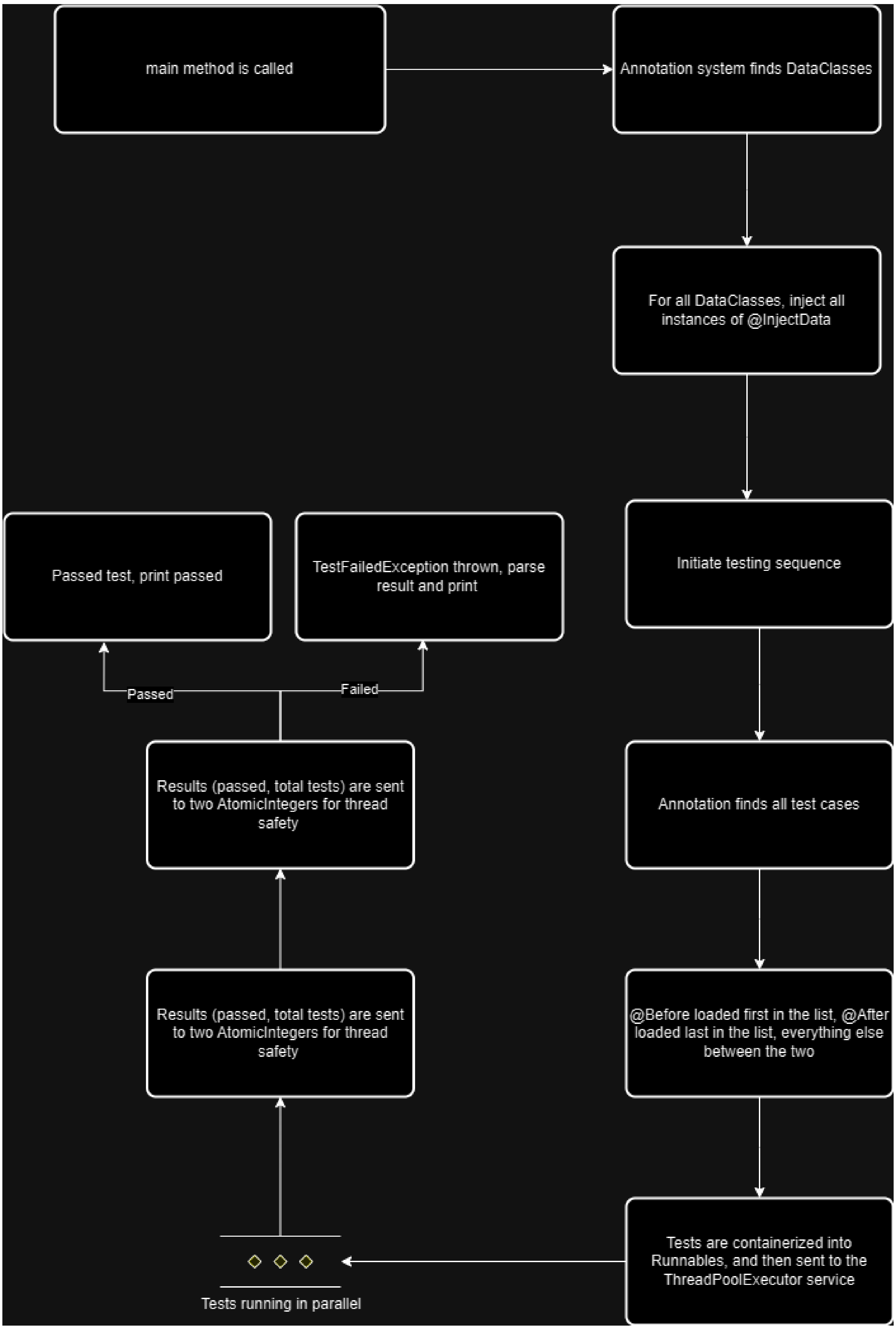
```
//AUTOGENERATED FROM ../src/TxtTestData.java

class TxtTestData {
    public static String DATA = "1\n2\n3\n4\n5\n6\n7\n8\n9\n10";
}
```

and everything works! In this case, because we are technically doing this at a "psuedo compile time" the `InjectData` is no longer used in the compressed Driver output.

The dreaded TestManager

This document is currently at 2500 words, and we haven't even gotten to the core Test loader!

Here's a rough flow diagram of what happens when you run the Driver:


main method is called
Annotation system finds DataClasses
For all DataClasses, inject all instances of @InjectData
Initiate testing sequence
Annotation finds all test cases
@Before loaded first in the list, @After loaded last in the list, everything else between the two
Tests are containerized into Runnables, and then sent to the ThreadPoolExecutor service
Tests running in parallel
Results (passed, total tests) are sent to two AtomicIntegers for thread safety
Results (passed, total tests) are sent to two AtomicIntegers for thread safety
Passed
Failed
Passed test, print passed
TestFailedException thrown, parse result and print

How on earth does the annotation system work?

The entirety of annotations is built on something known as Java reflection. Quite frankly, you *shouldn't* know or *ever* need to know it, but reflection gives your code the ability to look at other code... **ALL** of the other code. It doesn't matter if it's private, static, final, not even ran anywhere, reflection allows you to retrieve it.

Reflection sends you down a *massive* can of worms that would require several essays on its own (this "overview" is already too long), but we can use it to **scan** a class for all fields or methods that might have a particular annotation.

Here's a portion of the Test scenario:

```java
for (Method m : clazz.getMethods()) {
        TestCase testCase = m.getAnnotation(TestCase.class);
        Tip tip = m.getAnnotation(Tip.class);

        if (testCase != null) {
                try {
                        // Attempt to invoke method
                        m.invoke(instance);
                        // ...
```

In this case, we're looping over a class that happens to be registered at the start (using TestManager), where all methods are looped through and checked if they have the `TestCase` annotation. If they do, we try to **invoke** (run) the method, and catch for any errors. We catch for many different potential exceptions, but the main one is the `TestFailedException` previously mentioned.

How tests accidentally got multithreaded

It all started when someone left an Ed Discussion post regarding the tests "only working half of the time" where it wouldn't show any result whatsoever. The issue ended up being an infinite while loop in their code, resulting in nothing ever being able to be thrown.

So how do you check if there's an infinite loop and prevent it? You can't. There isn't any way of systematically detecting 100% that the code they run is in a while loop, but what we *can* do is check if their code takes a significant amount of time to run. This inconveniently requires the code to be on a separate thread, requiring some sort of threaded solution where we can figure out how long a thread might be running.

How tests are containerized

Each class is thrown into what is known in Java as a `Runnable` -- essentially a way of calling a method without necessarily knowing what that method might do.

The `TestContainer` is a custom type of `Runnable` with the following constructor:

```
/**
 * Initializes a new TestContainer. A TestContainer is used to prevent
certain edge cases such as infinite loops.
 *
 * If an infinite loop occurs, the test will timeout and notify the user
accordingly.
 * @param clazz The class containing the tests
 */
public TestContainer(Class<?> clazz) {
        this.clazz = clazz;
}
```

As the test has to have access to the provided Class methods, it is a required argument to initialize a TestContainer.

We then can create a list of TestContainers form all of the registered Tests:

```
        for (Class<?> testClass : testClazzes) {

                runnables.add(new TestContainer(testClass));

        }
```

For actually sending these runnables to their own threads, we can make use of Java's `ThreadPoolExecutor`:

```
ExecutorService executor = Executors.newFixedThreadPool(5);
```

For submitting these, I've gone ahead and annotated the following code:

```
for (Runnable r : runnables) {
        // Future<T> is a way of saying "I want the result in X amount of
time"
        Future<?> future = executor.submit(r);

        try {
                // We try to get the result within 10 seconds. If you DON'T
give it in 10 seconds, throw a TimeoutException
                future.get(10, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
                e.printStackTrace();
        } catch (ExecutionException e) {
                e.getCause().printStackTrace();
        } catch (TimeoutException e) {
                // Test took too long, tell them
                future.cancel(true);



System.out.println(ColorUtils.formatColorString(AsciiColorCode.BRIGHT_RED_BA
CKGROUND,

                                AsciiColorCode.BRIGHT_WHITE_FOREGROUND, "
FAILED: \u00BB ")

                                + " A test failed by exceeding the time
limit. You likely have an infinite loop somewhere.");


                // Intentionally crash the system to prevent them from doing
more harm
                System.exit(-1);

        }

}
```

Thread safety

One of the elements we have to be careful about when using multiple threads is thread safety. Put simply: multiple threads accessing or writing to the same information at the same time is really not a good idea.

The solution: `Atomics`.

Think of atomics as a way to synchronize the retrieval and setting of values. In this scenario, we use `AtomicInteger` numbers for the test cases, where each test uses methods such as these when updating the values:

```
// set is a method of AtomicInteger
// get is a method of AtomicInteger
TestManager.classTests.set(TestManager.classTests.get() + classTests);
```

Printing the end result

Thankfully due to how we use the `ThreadPoolExecutor` in this scenario, we know exactly when the tests will end. All that is left is actually printing it to the terminal:

```
StringUtils.printTextCentered("Test Results");

System.out.println();

StringUtils.printTextCentered(

                String.format("TOTAL TESTS PASSED: %d/%d", classTests.get()
- classTestsFailed.get(),

                                classTests.get()));

StringUtils.printHorizontalLine();
```

String Utilities

Arguably the most straightforward part of the program, this contains a bunch of utilities we use everywhere in order to handle formatting, such as centering text:

```
/**
 * Prints centered text to the terminal.
 *
 * @param text The text to be centered
 */
public static void printTextCentered(String text) {
        // HORIZONTAL_LINE_LENGTH is maximum width
        // if line is
        // xxxxxxxxxxxx
        // We can pad the start of the string with half of the horizontal:
        // xxxxxhelloxx
        // The problem is that this doesn't account for the length of the
string.
        // We can add half of the string length as well to correct it:
        // xxxxhelloxxxx = CENTERED! (or at least as close as it can get)
        System.out.printf("%" + (HORIZONTAL_LINE_LENGTH / 2 + text.length()
/ 2) + "s%n", text);
}
```

Horizontal Lines

In order to print completely straight lines, there's a symbol we can use for it:

```
/**

 * The ASCII character to use for horizontal lines

 */

public static final String HORIZONTAL_LINE_CHARACTER = "\u2500";
```