TABLE OF CONTENTS

TEST & BUILD THE APPLICATION

```
$ maven clean test package
```

JavaDoc generated documentation will be available in the following file:

```
./target/apidocs/index.html
```

This application requires Maven 3 and Java 8 (due to the use of streams and method references).

EXECUTE THE JAR IN THE COMMAND LINE

```
java -jar ./target/paintshop-1.0-SNAPSHOT-jar-with-dependencies.jar
    {-i <input file with test cases>}
    [-o <output file with batches solutions>]
```

Note that `-i` (or `--input-file`) is mandatory. If `-o` (or `--output-file`) isn't provided, the output will be printed to the console.

Examples:

```
java -jar ./target/paintshop-1.0-SNAPSHOT-jar-with-dependencies.jar
-i ./src/test/resources/inputs/success_from_specification.txt

java -jar ./target/paintshop-1.0-SNAPSHOT-jar-with-dependencies.jar
-i ./src/test/resources/inputs/performance/large_dataset.txt
-o large_dataset_output.txt

java -jar ./target/paintshop-1.0-SNAPSHOT-jar-with-dependencies.jar
-i ./src/test/resources/inputs/performance/small_dataset.txt
-o small_dataset_output.txt
```

For help execute the following command:

```
java -jar ./target/paintshop-1.0-SNAPSHOT-jar-with-dependencies.jar --help
```

ON THE DESIGN CHOICES

ALGORITHM

TL;DR;

The algorithm used is the following:

- Start with a solution with all colors glossy.
- Iterate through all customers, if it is strictly necessary to swap a matte color in the solution to satisfy the customer do it. If a customer cannot be satisfied by swapping a color to matte return impossible.
- If all customers are satisfied by the solution without any swap to matte return solution, otherwise iterate through all customers again.

In the section that follows the algorithm analysis is available in more detail. In short there are two options:

- Naive approach: Iterate through all possible solution batches, which has a time complexity of `O(M * 2^N)` (not advisable).
- Algorithm first described (with time complexity `O(M * N)`).

ALGORITHM ANALYSIS

A naive solution would be iterating through all possible solution batches, filter the ones that satisfy all batches and then return the one with the fewer number of matte colors.

That would not be advisable though, given that the time complexity of this approach would be:

```
O(M * 2^N)

Where N: Number of Colors.
      M: Number of Customers.
```

From the requirements, N and M could be both as large as 2000. This would require iterating the following number of times:

```
NI = M * 2^N = 2000 * 2^2000 = 229626139054850904846566640235536396
804463540417739040095528547365153252278474062771331897263301253983689192927797492554689423792172611066285186271233330637078259978290624560001377558296480089742857853980126972489563230927292776727894634052080932707941809993116324797617889259211246623299072328443940665362688337817968917011204758969615828117801869553000858005433413251661044016264472562583522535766634413197990792836254043559716808084319706366503081778867804183841109915567179344078320163914433261165510760851167452031056697572838864109017830551567765250350871057601645685541635930907524369702298058752000
```

That's about `2.3 * 10^605` iterations.

Since we want to minimize the number of mattes, nothing more fitting than starting with a solution where all colors are glossy.

While iterating through the customers, if the solution doesn't satisfy a particular customer and this customer can be only satisfied by swapping a color to matte, then we do it.

That will satisfy the current customer, but this swap might have made the solution unsatisfactory for some other customer.

We are required to iterate through all customers again to check if the solution still satisfies all customers because of that.

If the solution cannot satisfy a current customer and this customer cannot be satisfied by swapping a color to matte (the customer doesn't have any matte pair), this means that this customer cannot be satisfied without unsatisfying another customer and therefore there is no solution.

The maximum number of times one can swap a color to matte is equal to the number of available colors, therefore in the worst case scenario, the number of times we would have to iterate through all customers is equal to the number of colors.

The time complexity of this solution is therefore:

```
O(M * N)

Where N: Number of Colors.
      M: Number of Customers.
```

ARCHITECTURE

As a coding test is supposed to serve as portfolio of the candidate's software engineering skills, I'm adopting the mindset that would be fitting for production code.

Production code changes all the time, as well as requirements. Coding tests are usually not representative of production code's demands.

Before anyone accuses me of over-engineering, that's the reason why I'm being strict about following S.O.L.I.D. principles.

Could this application be simpler (less lines of code)? Yes. Would it be open-closed? No.

In order to achieve an open-closed design, the application has been broken into the following basic components:

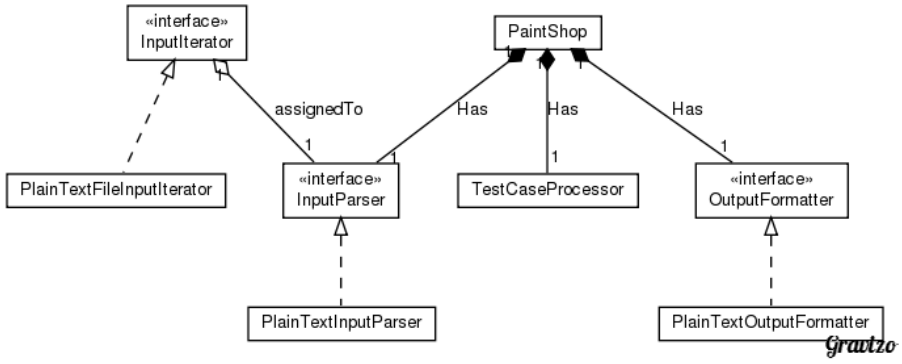

Figure 1: Alt text

`PaintShop` uses `InputParser`, `TestCaseProcessor` and `OutputFormatter` to perform the task of iterating through the input, parsing the input, processing the test cases and outputting the batches solutions respectively.

`InputParser` uses `InputIterator` to iterate through the file.

`InputIterator`, `InputParser` and `OutputFormatter` are abstract given that these are open to changes.

The current concrete implementation of `InputIterator`, `PlainTextFileInputIterator`, is used by the parser to iterate through a plain text file, but we could implement a concrete version that iterates through the standard input for instance if we want an interactive application.

The current concrete implementation of `InputParser`, `PlainTextInputParser`, parses inputs with the syntax defined in the specification, but we could implement another concrete version that parses an input with a different syntax.

The current concrete implementation of `OutputFormatter`, `PlainTextOutputFormatter`, formats solution batches according with the syntax in the specification, e.g.:

```
Case #1: 1 0 0 0 0
Case #2: IMPOSSIBLE
```

But we could create another concrete implementation that formats the output differently.

The `TestCaseProcessor` has no interface on the other hand, as the rule to generate batches doesn't seem to require multiple implementations in any instance. This simplifies the design.

I also would like to be emphatic about the fact that I follow clean code principles:

> "Every time you write a comment, you should grimace and feel the failure of your ability of expression." - Robert C. Martin

Sometimes I got complaints about "not having comments in my code". I try to make my code expressive. If you don't understand what a class does by it's name (the same goes for methods and variables) I accept the criticism, but I'll stick to my guns and rename my classes, methods or variable to better communicate the intention of the code. I avoid using comments, unless they are meant for JavaDoc.

It's also worth to mention that I'm using `BitSet` to store the batches of colors (solutions). In this manner colors and finishes are stored as bits and verifying if a solution satisfies a customer is performed using bitwise operations, which makes it fast.

PERFORMANCE

RANDOM INPUT GENERATOR

To be able to check performance, we need to create the "large data set" and "small data set" mentioned in the specification.

A Python script is available under `scripts/input-generator` for that. Refer to the source code for details. Note that this Python code has automated tests.

To run unit tests, simply execute its components as follows:

```
./scripts/input-generator/test_case.py
./scripts/input-generator/customer.py
```

All customers are generated at random (colors, finishes, as well as number of pairs for each customer), but all test cases have the same number of customers.

The usage is the following:

```
./input-generator.py <number of test cases> <number of colors>
<number of customers> <max number of pairs>
```

```
--output-file <name of the output file>
```

For help execute the following command:

```
./scripts/input-generator/input-generator.py -h
```

Which will output the following:

```
usage: input-generator.py [-h] [--output-file OUTPUT_FILE]
                          num_test_cases num_colors num_customers
                          max_num_pairs

Generate an random paint shop input file.

positional arguments:
  num_test_cases        Number of test cases to generate.
  num_colors            Number of colors in each test case.
  num_customers         Number of customers in each test case.
  max_num_pairs         Max number of pairs in each customer.

optional arguments:
  -h, --help            show this help message and exit
  --output-file OUTPUT_FILE
                        Name of file to output.
```

Examples:

```
./input-generator.py 100 10 10 2
--output-file ../../small_dataset.txt

./input-generator.py 5 2000 2000 3
--output-file ../../large_dataset.txt
```

Every time you run it, even with the same parameters, you will get a different file.

For reference, the generated data sets used for my performance tests are available under the following directory:

```
src/test/resources/input/performance
```

PERFORMANCE RESULTS

Running on my computer, a Lenovo Yoga 2 laptop/tablet running Ubuntu 12.04, I got the following results:

Large data set:

```
03:01:40 {master} ~/workspace/IdeaProjects/Zalando/Java/Paintshop$
java -jar ./target/paintshop-1.0-SNAPSHOT-jar-with-dependencies.jar
--input-file
./src/test/resources/inputs/performance/large_dataset.txt
```

```
--output-file large_dataset_output.txt

Total processing time: 28 ms
```

Small data set:

```
03:01:56 {master} ~/workspace/IdeaProjects/Zalando/Java/Paintshop$
java -jar ./target/paintshop-1.0-SNAPSHOT-jar-with-dependencies.jar
--input-file
./src/test/resources/inputs/performance/small_dataset.txt
--output-file small_dataset_output.txt

Total processing time: 5 ms
```

HOW TO VISUALIZE THIS DOCUMENT

This document is better visualized using IntelliJ's Markdown Plugin. In case it isn't available, there is a PDF version of this document in the same directory.

For my own reference, to convert markdown to PDF use the following command:

```
pandoc README.md -f markdown -t latex -s -o README.pdf
```