Java Assignment 3

Implementation Overview

Palette Representation

Initially, I considered representing the color palette with a 3D boolean array of size 256 × 256 × 256 (16,777,216 fields) to account for all possible RGB combinations. However, due to the relatively small limit on the number of colors allowed in the palette (1,024 compared to the total possible combinations), I opted for an ArrayList of hexadecimal string values to store the colors. Eventually, I transitioned to using a HashSet for better performance when handling large palettes and to ensure scalability.

add() Method and Input Validation

To provide flexibility in adding colors to the palette, I implemented two versions of the add() method through method overloading. One version accepts three integers (representing RGB values ranging from 0 to 255), while the other accepts a six-digit hexadecimal string representing the color.

Handling Duplicate Colors

If a color is added more than once, subsequent additions will have no effect on the palette's state. This design choice aligns with real-world applications like Adobe Photoshop or Microsoft Paint, where adding the same color repeatedly doesn't trigger errors.

Additional Features

In addition to the basic requirements, I included several auxiliary methods for user convenience:

- A method to convert RGB values to hexadecimal strings.
- A method to standardize and validate hexadecimal strings, allowing inputs such as fff123, fFf123, or #FFF123 without causing errors.
- Methods to remove colors from the palette, accepting both RGB and hexadecimal inputs.

Furthermore, I introduced the countColours() method to query the number of colors currently in the palette. This functionality also enhanced my ability to write comprehensive tests.

Test-Driven Development Experience

Benefits of TDD

Adopting a Test-Driven Development (TDD) approach offered valuable insights into identifying potential issues early. Writing tests before implementing functionality ensured:

1. **Early Detection of Errors**: Tests helped me quickly identify and address problems, saving debugging time.
2. **Clearer Design**: Thinking about test cases beforehand clarified the expected behavior of the ColourTable, which streamlined the coding process.
3. **Incremental Development**: TDD facilitated isolating parts of the code, focusing on small tasks, and building confidence in the overall correctness of the implementation.

Challenges with TDD

While effective, TDD posed certain challenges:

- Writing tests repeatedly sometimes detracted from the implementation phase, making it easy to lose focus on the assignment's main objectives.
- The bugs identified through TDD were relatively minor, such as correcting an exception message typo and fixing an error in an RGB-to-hexadecimal conversion snippet sourced from StackExchange.

Value of TDD in Large-Scale Projects

While TDD was helpful, its benefits are more pronounced in collaborative, large-scale projects. In such environments:

- The test suite serves as a clear specification for expected behavior, enabling team members to work independently on different components.
- A diverse team can develop higher-quality tests, mitigating the risk of a developer's tunnel vision during test creation.

Challenges Encountered

The main issue I faced was related to the folder structure generated by IntelliJ when using Maven. Switching devices caused discrepancies in folder organization due to different JDK versions. This issue was resolved by standardizing the JDK version across devices.

Reflection

Implementing TDD has been a valuable learning experience, reinforcing the importance of incremental development and thorough testing. Although the process had a learning curve, the resulting code was more reliable, and the structured approach helped streamline development. These insights will undoubtedly benefit me in future projects, particularly in collaborative environments.