Module: SV for Verification
Section: Testbench Basics **Task:** Overview

Task

An Overview

---

1) **Why do we need to have a separate class for each functionality like transaction, generator, and driver? What are the issues if we use a single class in the testbench for every functionality?**

   We need to separate functionality into different classes (e.g., transaction, generator, driver) for modularity, maintainability, and reusability. If a single class is used for all the functionalities, it becomes large and complex, making it difficult to develop, debug, and maintain. Each time new behavior is added, the entire class has to be modified, increasing the risk of introducing errors. The flat design also restricts reusability because different testbenches might require the reuse of only specific parts of the code (like the driver or transaction). Additionally, updating the class for a new feature, such as error injection or transaction modification, becomes cumbersome since all functionalities are intertwined in the same file.

   By breaking the testbench into smaller, specialized classes, each class can focus on its specific task (e.g., the transaction class handles data, the generator class creates transactions, and the driver sends them). This division promotes cleaner code, reduces maintenance costs, and allows for easier debugging and extension without affecting unrelated code.

2) **What is the issue with the composition approach in a testbench?**

   The composition approach involves creating separate classes for different functionalities and instantiating them within other classes, which can sometimes complicate code reuse and flexibility.

   The issue with this approach is that it may not allow for easily extending the behavior of an existing class. If the system evolves and new behaviors are needed (such as injecting different types of errors), composition might require reworking multiple parts of the code.

   In contrast, using inheritance allows the creation of new functionality (like error injection) by extending existing classes without needing to modify the original class. This provides greater flexibility and ensures the reuse of the existing infrastructure, minimizing the impact on already developed code.