Chapter 9 – Unit Tests – Summary

Unit Tests should:

- Be written before the corresponding production code (First Law of TDD).
- Fail initially and only pass when the production code is implemented correctly (Second Law of TDD).
- Be focused on testing a single concept or behavior in each test function.
- Follow the F.I.R.S.T. principles (Fast, Independent, Repeatable, Self-Validating, Timely).

Unit Tests to avoid:

- Tests that are slow and take too much time to run.
- Tests that depend on the order or execution of other tests.
- Tests that are not reproducible in different environments.
- Tests that require manual validation or reading of log files to determine success.

Must Do:

- Keep tests clean, readable, and well-structured.
- Use expressive and clear naming for test functions and variables.
- Create a domain-specific testing language to make tests more concise and readable.
- Maintain a dual standard for test code, allowing it to be less efficient but still clean.
- Focus on one assert statement per test while minimizing the number of asserts per concept.
- Write unit tests just before writing the corresponding production code.

Must Not Do:

- Write production code without corresponding unit tests.
- Allow unit tests to become messy, cluttered, or unreadable.
- Depend on other tests or external factors for test success.
- Write tests that are slow to execute and hinder development speed.
- Skip writing tests or delay test writing until after the production code is complete.

Best Practices:

- Follow the Three Laws of TDD for effective test-driven development.
- Use meaningful and descriptive test case names.
- Maintain a balance between test coverage and test execution speed.
- Use testing APIs and utilities to simplify test setup and verification.
- Continuously refactor and improve test code alongside production code.
- Keep unit tests as an integral part of the development process.

Chapter 10 – Classes – Summary

Classes should:

- Be organized following the standard conventions (variables, functions).
- Prioritize encapsulation to maintain privacy, unless necessary for testing.
- Emphasize small size and single responsibility.
- Have names that clearly describe their responsibilities.
- Be described briefly in about 25 words without conjunctions.
- Be designed to adhere to the Single Responsibility Principle (SRP).
- Maintain high cohesion, with methods manipulating related variables.
- Be organized to support change and minimize dependencies.
- Depend upon abstractions (interfaces) rather than concrete details.
- Embrace the Open-Closed Principle (OCP), allowing extension without modification.

Classes To Avoid:

- God classes that do too much and violate the SRP.
- Classes with ambiguous names that hint at multiple responsibilities.
- Large, multipurpose classes that hinder code understanding.

Must Do:

- Consider class organization as a higher-level code organization.
- Prioritize smaller classes with single responsibilities.
- Define clear and concise class names.
- Adhere to SRP by ensuring classes have one reason to change.
- Maximize cohesion within classes.
- Organize systems with many small classes.
- Use interfaces and abstract classes to isolate dependencies.
- Embrace the DIP by depending on abstractions rather than concretions.
- Design classes to be open for extension but closed for modification.

Must Not Do:

- Overlook higher-level code organization.
- Create large, multifunctional classes.
- Use ambiguous class names.
- Violate the SRP by having multiple reasons to change.
- Allow classes to have low cohesion.
- Rely heavily on large, multipurpose classes.
- Depend directly on concrete implementation details.