### Avoiding Slow Tests – Slow Test Code

- **Avoid Waits**
  - Use Humble Object to avoid Asynchronous Test by testing logic directly

- **Test Less Code**
  - Reduce Test Overlap

- **Set Up Less Fixture**
  - Use a Minimal Fixture

- **Set Up Fixture Less Often**
  - Reuse a Shared Fixture

---

### Shared Test Fixture

**What it is:**
- Improves test run times by reducing setup overhead.
- A “standard” test environment applicable to all tests is built and the tests reuse the same fixture instance.
**xUnit Test Patterns and Smells**

## Shared Test Fixture

### Variations:
- Fixture is shared between some/all the tests in a single test run
- Fixture may be shared across many TestRunners (Global Test Fixture)

### Examples:
- Standard Database contents
- Standard Set of Directories and Files
- Standard set of objects

*Bad Smell Alert: Erratic Tests*

---

**xUnit Test Patterns and Smells**

## Setting Up the Shared Test Fixture

**To share the same fixture instance between tests:**

### Prebuilt Fixture
- Fixture is built ahead of time and reused by many test runs

### Lazy Setup
- First reference causes it to be initialized
- How do you know when to clean up?

### SuiteFixture Setup
- Use Static variables to hold the fixture
- Initialize one before first test; destroy after last!

### Setup Decorator
- Define a Test Decorator that implements Test
- Wrap the test suite with an instance of the decorator

*Unrepeatable Tests*

*Use only when don’t need to clean up the fixture*

*Only supported by NUnit, VbUnit, JUnit 4.0*

*Tests that depend on the decorator cannot be run without it.*
Lazy Setup

- **What it is:**
  - We use Lazy Initialization to construct the Shared Fixture before the first Test Method that needs it.

```
if (!fixture_initialized) {
    initialize_fixture;
    fixture_initialized = true;
}
```

- **How it works:**
  - Hold reference to fixture in a static or global variable
  - Use Lazy Initialization of static variable to set up fixture.
  - Can be done *either* in the Setup method (Implicit Setup):
    ```
    if (not fixture_initialized) {
        initialize_fixture;
        fixture_initialzed = true;
    }
    ```
  - Or, in a finder method (Delegated Setup):
    ```
    Acct findXxAccount() {
        if (not fixture_initialized) {
            initialize_fixture;
            fixture_initialzed = true;
        }
        return xxxAccount;
    }
    ```
**SuiteFixtureSetup**

**What it is:**
- Test Framework support for sharing *test fixtures*.

**How it works:**
- All Test Methods in the Testcase Class share the same test fixture.
- Like a TestSetupDecorator but only for a single Testcase Class.
- TestFixtureSetUp method is called once before first Test Method
- SuiteFixtureTearDown Method is called once after last Test Method

**NUnit Specifics:**
- Indicated by [TestFixtureSetUp] and [TestFixtureTearDown]

**JUnit 4.0+ Specifics:**
- Indicated by the @beforeClass and @afterClass annotations
SetUp Decorator

**What it is:**
- We wrap the Test Suite Object with a Behavioral Decorator that sets up and tears down the fixture

**How it works:**
- Define a Test Decorator that implements the run() method on Test
  » Initializes the fixture
  » Calls basicRun() to run the test
  » Tears down the fixture
- Wrap the test suite with an instance of the decorator
- Decorator class TestSetup (in junit.extensions) does exactly this
  » provides a setUp() and tearDown() method to override
Erratic Tests

- Interacting Tests
  - When one test fails, a bunch of other tests fail for no apparent reason because they depend on other tests’ side effects

- Unrepeatable Tests
  - Tests can’t be run repeatedly without intervention

- Test Run War
  - Seemingly random, transient test failures
  - Only occurs when several people testing simultaneously

- Resource Optimism
  - Tests depend on something in the environment that isn’t available

- Non-Deterministic Tests
  - Tests depend on non-deterministic inputs

Erratic Tests – Interacting Tests

If many tests use same objects, tests can affect each other’s results.

- Test 2 failure may leave Object X in state that causes Test n to fail.

  Symptoms:
  - Tests that work by themselves fail when run in a suite.
  - Cascading errors caused by a single bug failing a single test.
    - Bug need not affect other tests directly but leaves fixture in wrong state for subsequent tests to succeed.
**Erratic Tests – Unrepeatable Tests**

If many test runs use the same objects, test runs can affect each other’s results.

- Test 2 update may leave Object X in state that causes Test 1 to fail on next run.

**Symptoms:**

- First run after opening the TestRunner or re-initializing Shared Fixture behaves differently
  - Succeed, Fail, Fail, Fail
  - Fail, Succeed, Succeed, Succeed
- Resetting the fixture may “reset” things to square 1 (restarting the cycle)
  - Closing and reopening the test runner for in-memory fixture
  - Reinitializing the database

**Erratic Tests – Test Run War**

- If many test runners use the same objects (from Global Fixture), random results can occur.
  - Interleaving of tests from parallel runners makes determining cause very difficult
**Erratic Tests – Non Deterministic Test**

Tests depend on **non-deterministic inputs**.

**Symptoms:**
- Tests pass at some times; fail at other times
  - Lack of control over time/date when system contains time/date logic (addressed by getting control of indirect input via a stub)
  - Tests use different values in different runs

**Erratic Tests – Resource Optimism**

Tests depend on **non-ubiquitous external resources**.

**Symptoms:**
- Tests pass in some environments; fail in others
  - SUT depends on something in the environment that is not always present.
  - Addressed by creating it during the fixture setup phase
Avoiding Erratic Tests - Fresh Fixture

- **What it is:**
  - “Brand new” fixture built for each test
  - Tests are completely independent

- **Variations:**
  - Transient Fresh Fixture
    » Fixture automatically disappears at end of each test
    » e.g. Garbage-collected TearDown
  - Persistent Fresh Fixture
    » Fixture naturally “hangs around” after test
    » Requires extra effort to ensure it is fresh
### Persistent Fresh Fixture

**Two Options:**

1. **Rebuild fixture for each test and tear it down**
   - **When**
     - At end of this test (just in case)
     - At start of next test that uses it (just in time)
   - **How**
     - Hand-coded Tear Down
     - Automated Tear Down

2. **Build different fixture for each test**
   - Use a Distinct Generated Value for any unique Id’s
   - Makes tear down necessary

### Reducing Erratic Tests - Shared Fixture

- **Avoid Interactions between Test Runners**
  - Give each developer their own Database **Sandbox**.
    - Avoids Test Run Wars but not Interacting Tests, etc,

- **Don’t Change Shared Fixture**
  - **Immutable** Shared Fixture avoids Interacting Tests
  - Create Fresh Fixture for objects to be changed
    - (See Persistent Fresh Fixture)
  - Challenge: What constitutes a “change” to a fixture?
    - Change existing objects / rows -> YES!
    - Add new objects related to existing objects -> SOMETIMES!