ECE 322 Lab 4 Report
Integration Testing

Diepreye Charles-Daniel

Introduction

The objective of this lab is to become familiar with integration testing techniques in the context of a modular Python application. The system under test is a simple database application composed of seven modules: A, B, C, D, E, F, and G. Module A acts as the command-line interface and delegates operations such as loading records, sorting, editing entries, and exiting the application to lower-level modules. Modules B, C, D, E, F, and G perform specific database operations such as file loading, sorting, displaying, modifying, and updating records.

This lab applies two integration testing methodologies: (1) non-incremental *Big Bang* integration, and (2) incremental *Top-Down* integration. The Big Bang approach integrates all modules simultaneously and tests the full system as a whole, while the top-down approach begins with the highest-level module (Module A) and uses stubs to replace lower-level modules until they are incrementally integrated. These methods allow us to evaluate the system both holistically and in isolated module interactions.

Definitions of Testing Methods

Integration Testing Strategies

Integration testing focuses on verifying interactions between modules after unit testing has confirmed local correctness. In **Big Bang integration**, all modules are combined at once, and the entire application is tested as a whole. This approach is simple to set up but makes fault isolation more difficult.

Incremental integration decomposes testing into steps where modules are added progressively. In **Top-Down integration**, testing begins with the highest-level module and substitutes lower-level modules with stubs until they are ready to be integrated. This enables early testing of control-flow logic. In **Bottom-Up integration**, the lowest-level modules are tested first using drivers; however, this method was not chosen for this lab.

Purpose of Stubs and Drivers

A **stub** replaces a lower-level module during testing, returning controlled, deterministic values to isolate the behavior of the module under test. A **driver** simulates a higher-level

caller to exercise lower-level modules in bottom-up testing. Stubs and drivers are essential in incremental integration because they allow partial systems to be tested without requiring full implementations.

Part 1: Big Bang Integration Testing

Q1: Modules, Order, Stubs, and Drivers Used

Modules	Order	Stubs	Drivers
A, B, C, D, E, F, G	1	—	—

Table 1: Big Bang Integration: Full system integrated with no stubs or drivers.

Q2: Test Case Table for Big Bang Testing

ID	Input	Modules	Description	Expected Result	Actual Result
BB1	parseLoad ("data_copy.txt") then runSort()	A, B, C, F	Load full database and sort records.	Data loads successfully and is sorted alphabetically.	Pass: records sorted and printed in order.
BB2	Sequence of add, update, delete operations via ModuleA.	A, D, F, G	Exercise insert, update, and delete path through modify pipeline.	Data reflects all operations (record added, updated, then removed).	Pass: all operations executed correctly and final data consistent.

Table 2: Test cases for Big Bang integration testing.

Part 2: Incremental (Top-Down) Integration Testing

Q3: Modules, Order, Stubs, and Drivers Used

Modules	Order	Stubs	Drivers
A	1	B, C, D, E	—

Table 3: Incremental Integration: Top-Down approach using stubs for B, C, D, and E.

Q4: Test Case Table for Incremental Testing

Incremental method used: Top-Down Integration

ID	Input	Modules	Description	Expected Result	Actual Result
TD1	`parseLoad ("dummy.csv")`	A + StubB	Ensure ModuleA delegates loading to ModuleB and updates internal data.	StubB.`loadFile` called with filename; A.`data` equals stub data.	Pass.
TD2	`parseAdd ("NewName", "999")` with existing data and filename.	A + StubD	Ensure ModuleA delegates insert to ModuleD with correct parameters.	StubD records call; A.`data` contains old and new entries.	Pass.
TD3	`runSort()` with three unsorted entries.	A + StubC	Verify sorting delegation: A passes data to C and updates data with result.	StubC sees original list; A.`data` updated to reversed list from stub.	Pass.
TD4	`runExit()`	A + StubE	Verify exit delegation from A to E.	StubE. `exitProgram` is called.	Initially fail due to `exitProgam` typo in ModuleA; passes after fix.

Table 4: Test cases for incremental Top–Down integration testing.

Q5: Discussion

Integration testing was effective in validating the interactions between modules and identifying cross-module issues. Big Bang testing confirmed the correct overall behavior of the system but made it more difficult to pinpoint individual module failures. In contrast, the incremental top-down approach enabled focused isolation of Module A. Because stubs were used to simulate lower-level modules, the top-down tests detected a bug in the `runExit()` function of Module A. Specifically, Module A attempted to call `exitProgam()` instead of `exitProgram()`, revealing a spelling error that prevented correct delegation.

Stubs proved to be extremely effective during incremental integration, simplifying the observation of interactions and ensuring deterministic behavior. The differences between Big Bang and incremental testing were evident: Big Bang validated system-wide correctness, while incremental testing gave fine-grained diagnostic feedback. Incremental integration is more scalable for larger systems because adding modules gradually reduces debugging complexity. It is also easier to maintain due to its modular testing design.

Conclusion

This lab demonstrated the importance and utility of integration testing in verifying interactions among software modules. Both Big Bang and Top-Down incremental integration approaches were applied successfully. While Big Bang provided confidence in whole-system behavior, the top-down approach allowed early detection of control-flow issues, such as the method name typo in Module A. Overall, integration testing proved essential for ensuring correctness across module boundaries.

Appendix A: Big Bang Integration Test Code

```
import os
import shutil
import unittest

from modules.ModuleA import ModuleA
from modules.ModuleB import ModuleB
from modules.ModuleC import ModuleC
from modules.ModuleD import ModuleD
from modules.ModuleE import ModuleE
from modules.ModuleF import ModuleF
from modules.ModuleG import ModuleG


class BigBangIntegrationTest(unittest.TestCase):
    """
        Non-incremental (Big Bang) integration tests.

        Uses the real modules AG together, exercising:
          - loading from file (B + F)
          - sorting (C + F)
          - insert/update/delete (D + F + G)
    """

    def setUp(self):
        project_root = os.path.dirname(os.path.dirname(__file__))

        # Use a temporary copy of data.txt so tests can freely modify it
        self.original_file = os.path.join(project_root, "data.txt")
        self.test_file = os.path.join(project_root, "data_test_copy.txt")
        shutil.copyfile(self.original_file, self.test_file)

        f = ModuleF()
        g = ModuleG()
        self.app = ModuleA(
            ModuleB(f),
            ModuleC(f),
            ModuleD(f, g),
            ModuleE()
        )
```

```
def tearDown(self):
    # Remove the temporary test file
    if os.path.exists(self.test_file):
        os.remove(self.test_file)

def test_load_and_sort_big_bang(self):
    """Load the DB file and then sort it through the full AG stack."""
    ok = self.app.parseLoad(self.test_file)
    self.assertTrue(ok)

    self.assertIsNotNone(self.app.data)
    self.assertEqual(6, len(self.app.data))

    names_before = [entry.name for entry in self.app.data]

    sort_ok = self.app.runSort()
    self.assertTrue(sort_ok)

    names_after = [entry.name for entry in self.app.data]
    self.assertNotEqual(names_before, names_after)
    self.assertEqual(sorted(names_after), names_after)

def test_add_update_delete_big_bang(self):
    """
        Exercise insert, update, and delete across AG.


        This runs through:
          - D.insertData -> F.displayData -> G.updateData
          - D.updateData -> F.displayData -> G.updateData
          - D.deleteData -> F.displayData -> G.updateData
    """
    self.app.parseLoad(self.test_file)

    # --- Add record ---
    add_ok = self.app.parseAdd("Alice", "9999")
    self.assertTrue(add_ok)
    self.assertEqual(7, len(self.app.data))
    self.assertEqual("Alice", self.app.data[-1].name)

    # --- Update record: change 3rd entry's name to 'James' ---
    third_before = self.app.data[2].name
    update_ok = self.app.parseUpdate(3, "James", "56789")
    self.assertTrue(update_ok)
    self.assertEqual("James", self.app.data[2].name)

    # --- Delete 2nd entry ---
    name_deleted = self.app.data[1].name
    delete_ok = self.app.parseDelete(2)
    self.assertTrue(delete_ok)
    self.assertEqual(6, len(self.app.data))
    self.assertNotIn(name_deleted, [e.name for e in self.app.data])
```

```
if __name__ == '__main__':
    unittest.main()
```

Appendix B: Top-Down Integration Test Code

```
import io
import contextlib
import unittest

from data.Entry import Entry
from modules.ModuleA import ModuleA
from modules.ModuleB import ModuleB
from modules.ModuleC import ModuleC
from modules.ModuleD import ModuleD
from modules.ModuleE import ModuleE


# ---------- STUBS FOR TOP-DOWN INTEGRATION ----------

class StubB(ModuleB):
    def __init__(self):
        # Don't call super().__init__; we don't need ModuleF here.
        self.loaded_filename = None
        self.data_to_return = [Entry("StubName", "111")]

    def loadFile(self, filename: str):
        self.loaded_filename = filename
        # Return a fresh list each time so tests can safely mutate it
        return list(self.data_to_return)


class StubC(ModuleC):
    def __init__(self):
        # Skip real __init__ (no ModuleF needed)
        self.received_data = None

    def sortData(self, data):
        # Record input and return a deterministic "sorted" result
        self.received_data = list(data) if data is not None else None
        return list(reversed(data)) if data is not None else None


class StubD(ModuleD):
    def __init__(self):
        # No real F/G needed for the stub
        self.insert_calls = []
        self.update_calls = []
        self.delete_calls = []

    def insertData(self, data, name, number, filename):
        # Record the call
```

```
        self.insert_calls.append((list(data) if data else [], name, number, filename))
        # Simulate adding a record
        new_data = list(data) if data else []
        new_data.append(Entry(name, number))
        return new_data

    def updateData(self, data, index, name, number, filename):
        # Record the call
        self.update_calls.append((index, name, number, filename))
        new_data = list(data)
        if 0 <= index < len(new_data):
            new_data[index] = Entry(name, number)
        return new_data

    def deleteData(self, data, index, filename):
        # Record the call
        self.delete_calls.append((index, filename))
        new_data = list(data)
        if 0 <= index < len(new_data):
            del new_data[index]
        return new_data


class StubE(ModuleE):
    def __init__(self):
        self.exit_called = False

    def exitProgram(self):
        # Override real behaviour: record call instead of sys.exit()
        self.exit_called = True


# ---------- TOP-DOWN INTEGRATION TESTS ----------

class TopDownIntegrationTest(unittest.TestCase):
    """
    Incremental Top-Down integration tests.

    We test ModuleA at the top, replacing B/C/D/E with stubs that extend
    the original modules. This demonstrates top-down integration with stubs.
    """

    def setUp(self):
        self.stub_b = StubB()
        self.stub_c = StubC()
        self.stub_d = StubD()
        self.stub_e = StubE()

        # Pass stubs into A; they are subclasses of the real modules
        self.app = ModuleA(self.stub_b, self.stub_c, self.stub_d, self.stub_e)

    def test_parseLoad_calls_B_and_sets_data(self):
        """A.parseLoad should call B.loadFile and store the returned data."""
        self.assertIsNone(self.app.data)
```

```
        ok = self.app.parseLoad("dummy.csv")
        self.assertTrue(ok)

        # StubB should have been called with filename
        self.assertEqual("dummy.csv", self.stub_b.loaded_filename)
        # A's data should now match what StubB returned
        self.assertEqual(self.stub_b.data_to_return, self.app.data)

    def test_parseAdd_delegates_to_D_insertData_and_updates_data(self):
        """A.parseAdd should delegate to D.insertData with correct args."""
        # Simulate file already loaded
        self.app._data = [Entry("Existing", "000")]
        self.app._filename = "dummy.csv"

        ok = self.app.parseAdd("NewName", "999")
        self.assertTrue(ok)

        # One insert call recorded in stub D
        self.assertEqual(1, len(self.stub_d.insert_calls))
        call_data, name, number, filename = self.stub_d.insert_calls[0]
        self.assertEqual("NewName", name)
        self.assertEqual("999", number)
        self.assertEqual("dummy.csv", filename)

        # A's in-memory data should now contain both entries
        self.assertEqual(2, len(self.app.data))
        self.assertEqual("Existing", self.app.data[0].name)
        self.assertEqual("NewName", self.app.data[1].name)

    def test_runSort_uses_C_sortData_and_updates_data(self):
        """A.runSort should call C.sortData and replace A.data with the result."""
        # Start with a known order
        self.app._data = [
            Entry("B", "2"),
            Entry("A", "1"),
            Entry("C", "3"),
        ]

        ok = self.app.runSort()
        self.assertTrue(ok)

        # StubC should have seen the original list
        self.assertEqual(["B", "A", "C"],
                         [e.name for e in self.stub_c.received_data])

        # A.data should now be the reversed list (as defined in StubC.sortData)
        self.assertEqual(["C", "A", "B"],
                         [e.name for e in self.app.data])

    def test_runExit_delegates_to_E_exitProgram(self):
        """
        A.runExit should delegate to E.exitProgram.
```

```
        As provided, ModuleA.runExit calls self._e.exitProgam() (typo),
        so this test will initially FAIL. After fixing runExit to call
        exitProgram(), this test will pass.
        """
        buf = io.StringIO()
        with contextlib.redirect_stdout(buf):
            # Call the public exit function
            self.app.runExit()

        self.assertTrue(self.stub_e.exit_called)


if __name__ == "__main__":
    unittest.main()
```

Appendix C: Test Runner Code

```
# tests/TestRunner.py
import unittest

# Import your test modules (they should be in the same 'tests' package/folder)
import BigBangIntegrationTest    # Non-incremental (Big Bang) integration tests
import TestModuleA# Incremental (Top-Down) integration tests


def main():
    loader = unittest.TestLoader()
    suite = unittest.TestSuite()

    # Add all tests from each module
    suite.addTests(loader.loadTestsFromModule(BigBangIntegrationTest))
    suite.addTests(loader.loadTestsFromModule(TestModuleA))

    runner = unittest.TextTestRunner(verbosity=3)
    runner.run(suite)


if __name__ == "__main__":
    main()
```