Verify Integrations Without Integration Tests

Contract Test Your Port and Adapters

Problem and Solution Overview

This recipe just states how to execute a solution. Read **DevOps #8 – Find Integration Bugs without Integration Tests** to understand the specific problem we are solving and the solution approach.

Access **DevOps Series** for getting your software clean for the pipeline.

This recipe helps you verify that your dependencies will integrate correctly with your component, without running any integration tests.

Identify One Behavior

Look for a place where your product or test takes an action using the other system and then checks the resulting state. These two operations may be currently performed via the Port, the Adapter but outside the Port, direct calls to the other system's API, or by interacting with the other system's resources.

Steps:

1. Look for one of these four things:
   Integration setup action. Integration test calls an action before calling your component's code. This may be initial test setup or some intermediate step.
   Component action. Your component calls an action, while executing inside an integration test.
   Component state check. Your component checks state in order to decide what to do, while executing inside an integration test.

Integration state check. Integration test checks state in its assertions. These could be final assertions or some intermediate step.

2. Once you have found one half of a pair, find its match as follows. If you find an implicit check or action, take the indicated step in order to make it explicit, then commit.

- Integration setup action
  - Explicit check: integration test verifies state before calling product code.
  - Explicit check: product code checks state to make a decision.
  - Implicit check: integration test is intending that something be true, so that the product code's actions operate in a specific way. Figure out what it expects to be true. Add that as assertions in the integration test.
- Component action
  - Explicit check: assertions in integration test.
  - Implicit check: product code or integration test then takes another action. Determine what the code assumed the first action accomplished. Add that as assertions to the integration test.
- Component state check
  - Explicit action: Integration code took an action before the check.
  - Explicit action: Component took some action before the check.
  - Implicit action: Integration test found some pre-existing state that met its assumptions. Change the test to call an action and create the conditions it expects.
  - Implicit action: Integration test set up shared resources in a way that had a desired side-effect. Figure out the desire and add it to the other system's public API (even if only on a test-only API). Change the integration test to use that.
- Integration state check
  - Explicit action: Component code took an action.

Create a Contract Test for that Behavior

Steps:

1. Create empty contract test classes if you don't already have them.
   - Create an abstract base that contains only an abstract `Port _SupplyTestSubject()` method.
   - Create at least one child class that implements `_SupplyTestSubject()` to instantiate on Adapter for a particular external system and returns the Port from that Adapter.
   - Put the child class into the right test suite. It's a unit test if it operates only on objects in memory and has no shared state between tests. Otherwise put it in a platform test suite for that specific external system.
   - Commit. Merge to main.
2. Create a new empty test.

If both the action and state check are performed through the Port already, put the test into the contract tests.
Otherwise, create the test in the Adapter-specific test child class for the correct other system.
Name it according to the action and the resulting state check.
A good naming convention is `<action>_Should_<resulting state>()`.
Commit.

3. Copy in both the action and the state check that you found.
4. Commit.
5. Convert the state checking code to make assertions.
6. Make the test pass, if it doesn't already.
    Do not try to encapsulate things or make it readable. Just make it pass.
7. Commit. Merge to main.

Encapsulate the Behavior into the Port

This step assumes that the test uses something via a direct call to the Adapter, to the external system's public API, or by modifying one of the external system's resources. If the test already uses only the Port, then skip to extend coverage to one more external system.

This approach fixes encapsulation incrementally. Start by fixing encapsulation violations on the innermost layer; convert each to use encapsulation at that layer but no more. Then look for encapsulation violations at the next layer out, which will include the ones you just created. Working incrementally ensures you don't miss anything. It also allows you to commit after each step so that you always have a working system.

1. Fix Direct Access

Steps:

1. Identify any time you are accessing a foreign system's internal resources. Common examples include:
    A. Modifying data in a database.
    B. Updating or clearing a cache.
    C. Updating a configuration parameter.
    D. Restarting a server or launching a process.
    E. Modifying files on a file system or network share.
2. Decompose your accesses into chunks.
    A. Each chunk should perform one semantic action. For example, one chunk may create a new user, then another chunk may add a loan application for that user.
    B. `Extract Method` each chunk into a well-named method.
    C. Commit.
3. Merge to main.
4. Add any missing capabilities to the foreign system's public API.
    A. You may need to negotiate with another team. Give them your code or a PR as a starting point.

    B. This may not be a capability they want to expose on their public API.
        1. Ask for a second, test-only API.
        2. Have them disable the test-only API in production.
        3. Run your platform tests against their pre-production code and environment.
5. Migrate each chunk to use the public API. In parallel, one at a time:
    A. Wait for the other team to extend the public API, if needed.
    B. Replace the extracted method with a call to the public API.
    C. Run your test. It should still pass.
    D. Commit.
6. Merge to main.

2. Fix API Access

Steps:

1. `Extract Method` each call or sequence of calls to the public API.
2. `Move Method` those calls into the Adapter.
3. Commit.
4. Clean up each method. Do this incrementally with many commits along the way.
    A. Group new methods into the Adapter's existing objects.
    B. `Rename` methods to match the metaphors used elsewhere in the Adapter.
    C. Commit.
5. Merge to main.

3. Fix Adapter Access

Steps:

1. Encapsulate details that your component doesn't want to deal with. Work incrementally and commit often.
    A. Encapsulate any concerns that your Adapter has encapsulated in other methods, in the same way.
    B. Encapsulate concerns that are specific to this external system.
    C. Encapsulate concerns that you wish your component didn't have to deal with.
2. Merge to main.
3. Refactor each method, one at a time, to the Port's existing metaphors.
    A. `Rename` methods appropriately.
    B. Align parameter types and dispatch types (events, async, etc).
    C. Commit.
4. Add each method, one at a time, to the Port.
    A. Add the method to the Port.
    B. Implement that method on each other Adapter to `throw new NotImplementedException()` or the equivalent. Make it compile.
    C. Commit.
    D. Convert your test to call through the Port instead of the Adapter.

    E. Run all tests. They should still pass; this verifies that nothing is calling the non-implemented methods in other Adapters.
    F. Commit.
5. Merge to main.

Extend Coverage to One More External System

This part may require waiting on external teams. Start one at a time, and start another one if you end up blocking on an external team. Try to minimize work in progress; it is usually better to have a contract test suite that has a complete test for some adapters and nothing for others than one with in-progress tests for several at once.

Steps:

1. If you are expanding from one Adapter to 2, then:
    A. The test definition will currently be in the Adapter-specific test child class.
    B. `Extract Method` the entire body of the test.
    C. `Pull Member Up` to move the extracted method to the Contract Test base class. Leave it as a regular method, not a test.
    D. Commit and merge to main.
2. Copy the test from one existing Adapter-specific test class to the new Adapter-specific test class.
    A. Its body will be just a call to the definition in the base class.
    B. It will probably fail, usually due to unimplemented methods on the Port.
    C. If it happens to pass, then commit and skip to checking for final cleanup — you are done extending the test to cover this external system.
3. Mark the test on this child class to be ignored.
4. Commit. Merge to main.
5. Run the test, find the first non-implemented method or other failure.
6. TDD that method into existence.
    A. This may require asking another team to extend the public API for their external system. Approach this the same way as in fix direct access, above.
    B. Implement the method from the outside in. Create new `NotImplemented` methods on inner layers as you need them. For example, implement the Port methods using Adapter methods you have. And add new `NotImplemented` Adapter methods as needed. Do the same to implement Adapter methods in terms of the public API.
    C. Work in small steps and commit often. Merge to main often as well.
    D. Avoid mocks if you have already used better techniques to isolate your dependencies. If not then use mocks so that your tests pass even though the next layer hasn't implemented the methods yet.
    E. Make clear requests for public API methods only after you have implemented the Adapter.
7. Respond to completed requests for public API methods.

A. Perform the cleanup steps from the fix API access part and then the fix Adapter access part.
B. Remove any mocks you used as workarounds in step 6. Use your other mechanisms to handle the dependencies without the ongoing integration problem caused by mocks.
C. Commit. Merge to main.

Check for Final Cleanup

Have you extended the test to now cover every Adapter (every child test class)? Perform final cleanup if so.

Steps:

1. Find the test definition method in the Contract Test base class. It's the method called by all the tests in the child classes.
2. Make the test definition method into a regular test.
3. Delete all the tests in the child test classes.
4. Commit. Merge to main.
5. Reduce the scope of any integration test you can.
   Remove anything that duplicates the new contract test if you can.
   Comment it as duplicate if you can't delete it.
   Delete any test that is now empty or entirely duplicated (as you have extracted several Contract Tests and Simulator Tests).