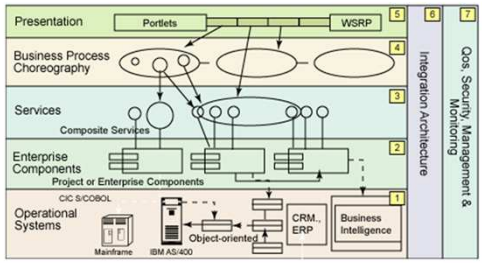
Mock Objects in Functional Testing

Sven Rosvall

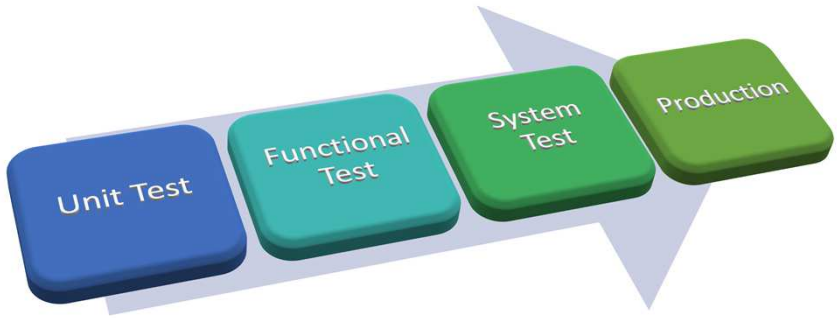
Some slides are animated. They look quite messy unless viewed in slideshow mode.

Dimension Data
Cloud Business Unit

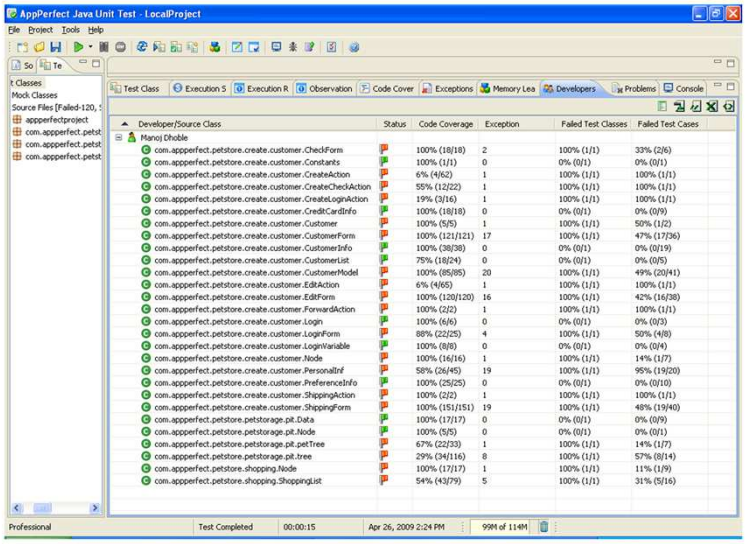


I am working with cloud management systems. This is built as a multi-tier SOA application. This means we have many services that depend on each other which complicates testing.

Testing Lifecycle



Unit Testing



Functional Testing

Put developed classes and any libraries together into a service/link unit/executable. Test this in a test bed where other services are provided as a fixed installation or mocked out.

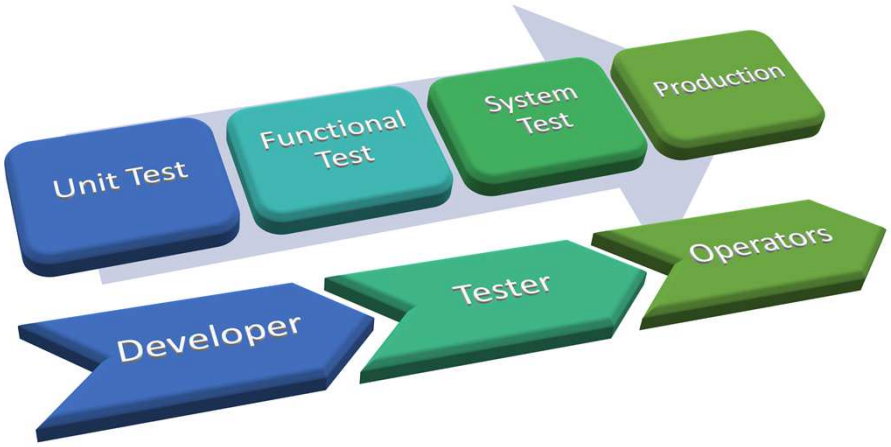
System Testing

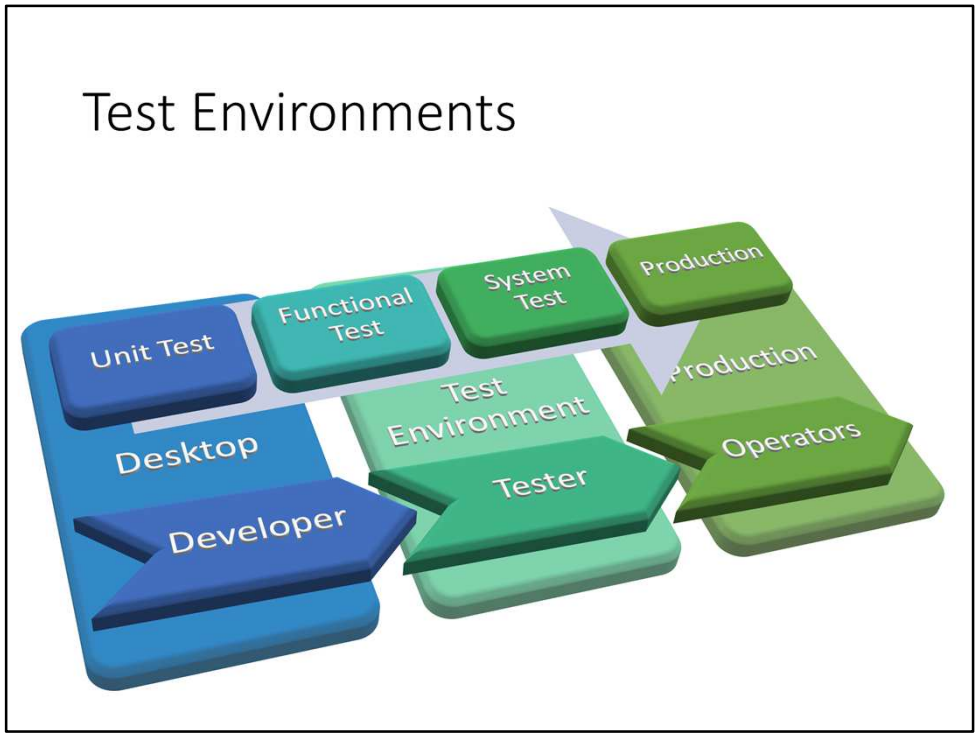
Test several services/executables together as a system as it will be deployed in the production environment.

Production

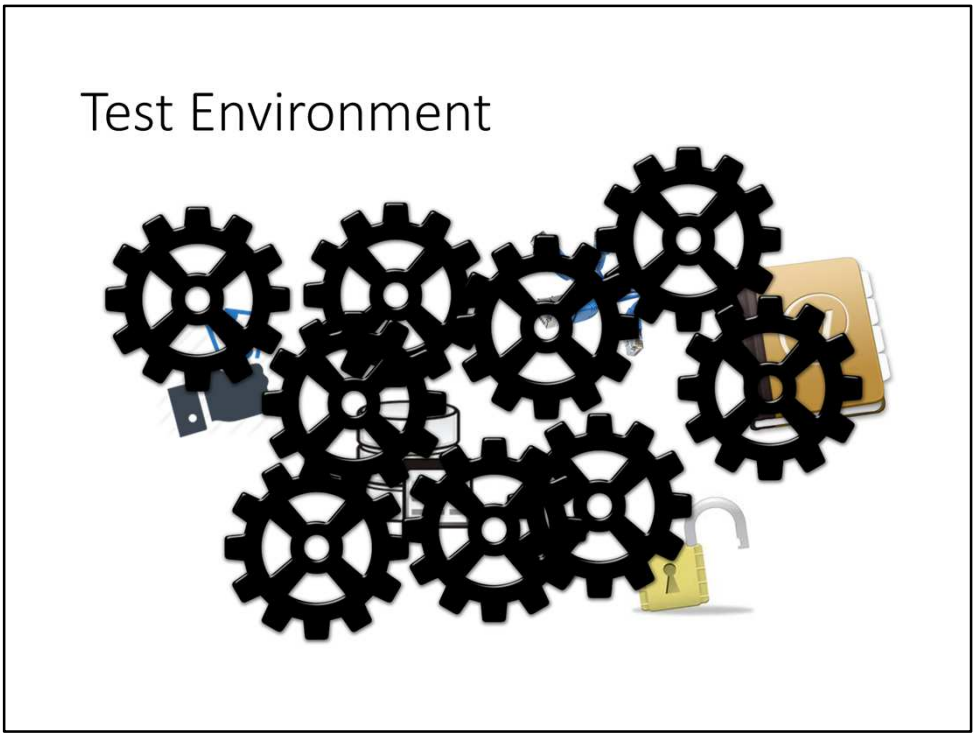
Deliver your work to production.

Test Responsibilities

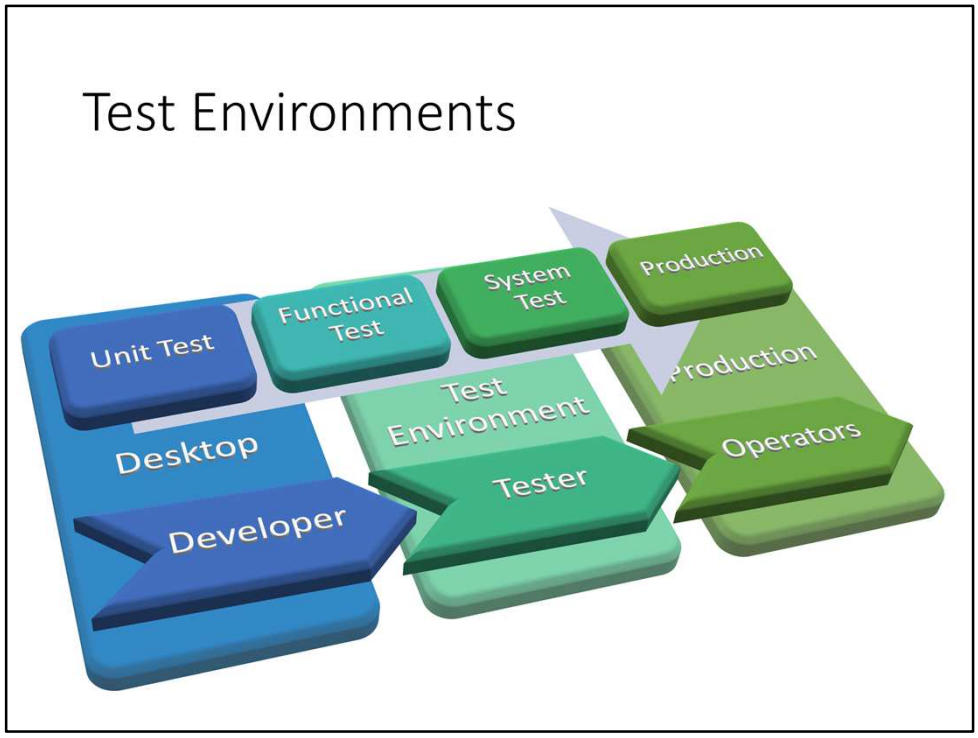




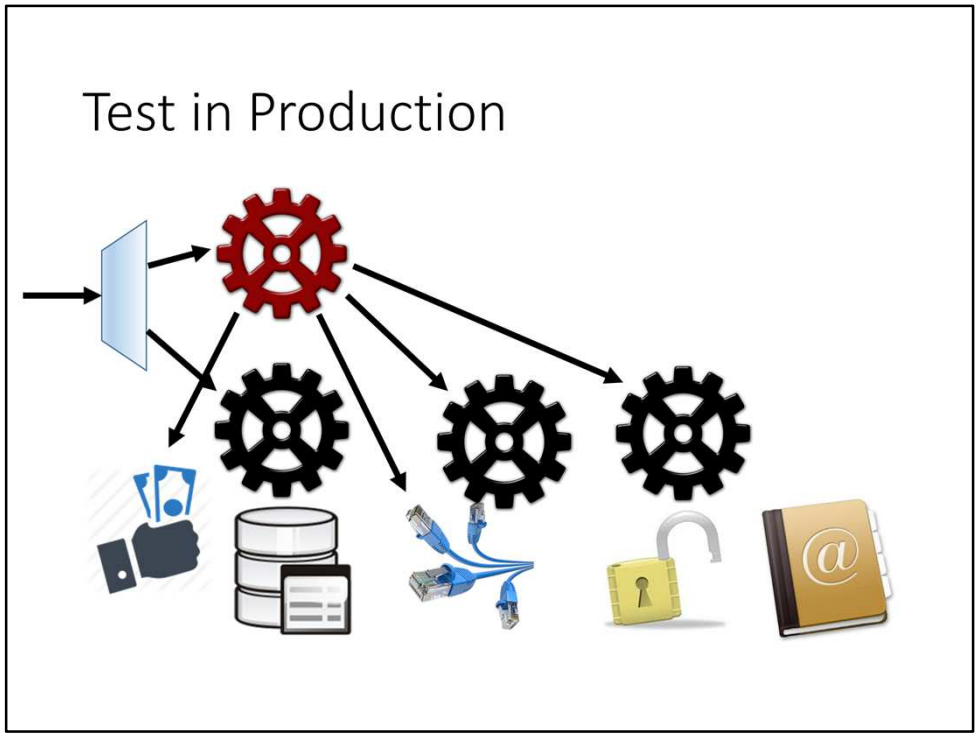
Different environments we use to run our components/systems for the purpose of testing or production.



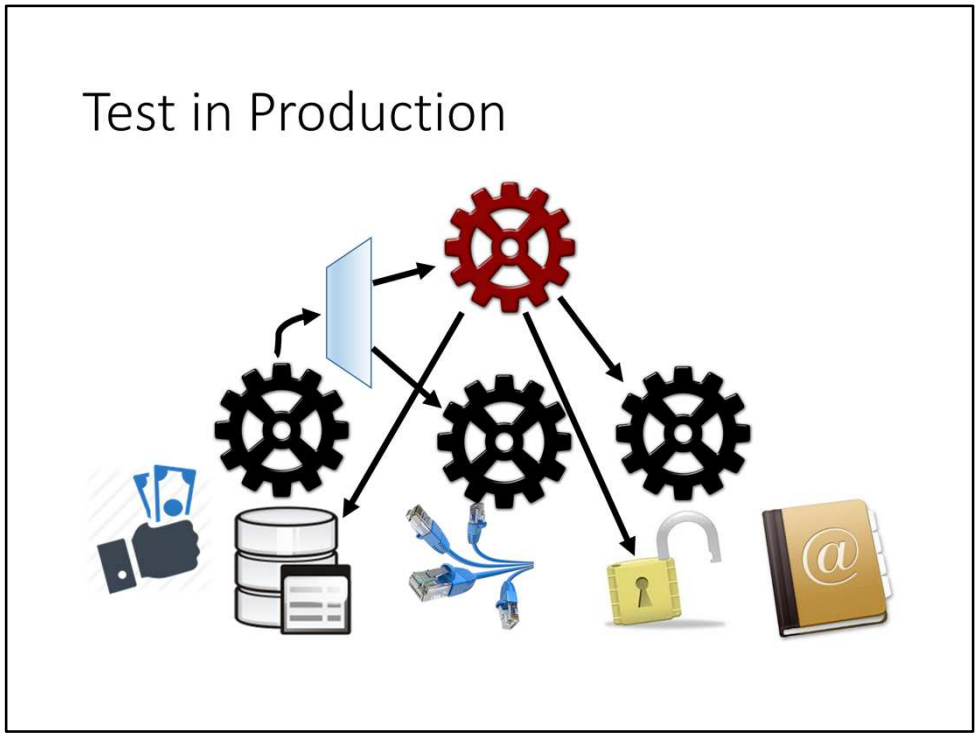
Test environments are complex to set up and maintain. There may be complex external systems such as directories, databases, payment systems, networks, security. There are also many services that must be provided at the correct version. Can be really difficult if many of these services are under development simultaneously. Some companies use many test environments, each with different purposes.



As the test environments are tricky to maintain, we want to reduce the dependency on them. Do this by “test in production” and more developer testing.

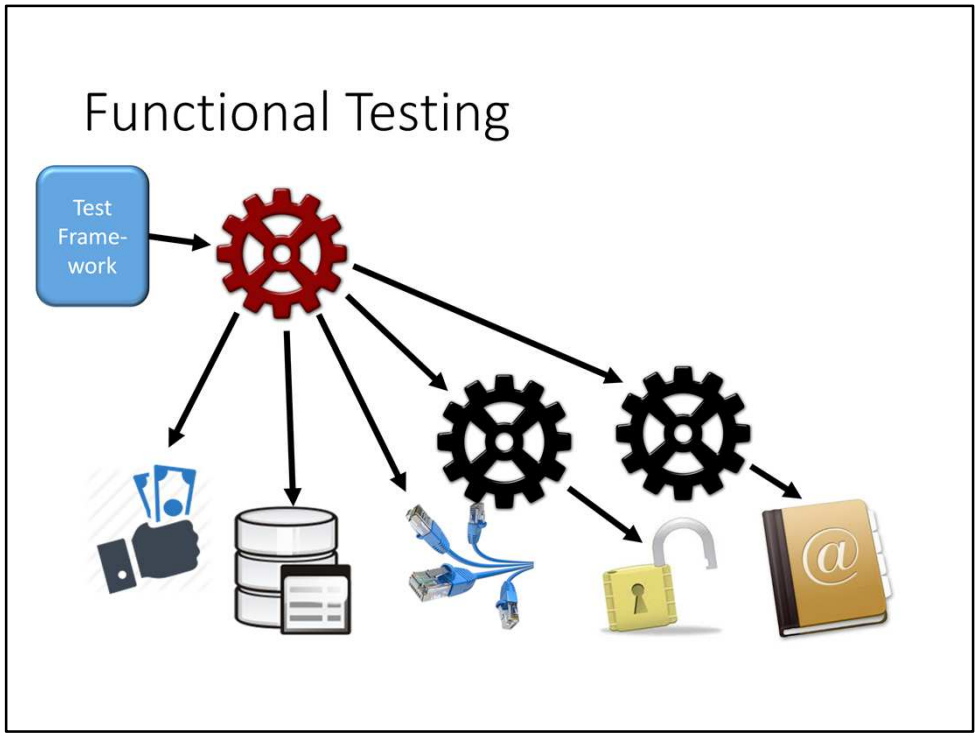


Test in production by introducing your service that uses production services to provide live data support. Test your service manually at first. Then let a load balancer direct some live traffic to the new service.



The service under test can also be introduced at middle-tier level.

Test Environments



Test one service at a time. Provide dependent services. These services may in turn depend on other services. This makes the test environment complex, even if the service under test is small.

Mocking Dependent Services

Goals:

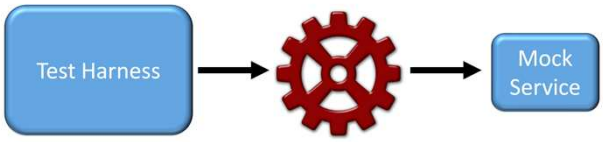
- Mimic required functionality for testing
- Easy to maintain and to deploy
- Verify requests
- Control expected responses
- Remove indirect dependencies

Modes of communication

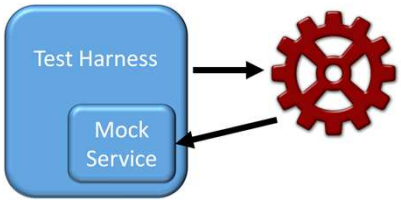
- Synchronous Requests
- Asynchronous Messages
- Files
- Shared memory
- Database updates and triggers

Mocking an HTTP service

Simple mock HTTP service



Embedded HTTP service within test program



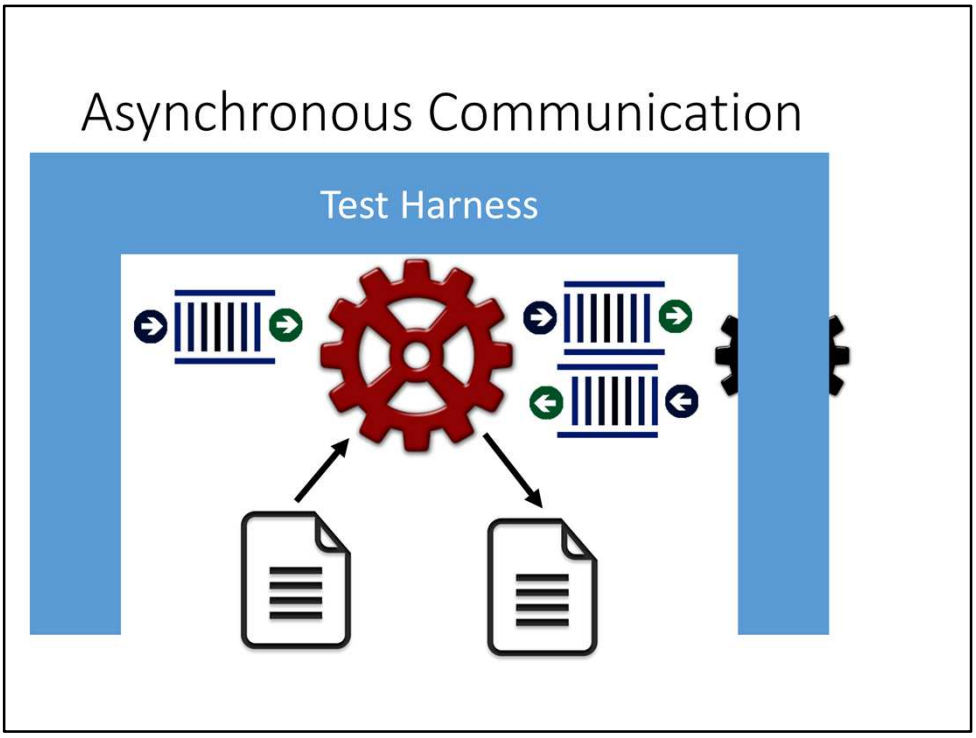
WireMock

```
@Rule
public WireMockRule wireMockRule = new WireMockRule(PORT);


String responsePayload = fromFile("GetDeviceResponse_ok.json");

stubFor(get(urlEqualTo("/api/device?limit=100"
        + "&filter.component_unique_id=" + vmRefId
        + "&filter.component_root_device=" + vcenterDID))
    .withHeader("Authorization", equalTo("Basic "
        + new String(basicAuth, Charset.forName("UTF-8"))))
    .willReturn(aResponse()
        .withHeader("Content-Type", "application/json")
        .withStatus(HttpStatus.SC_OK)
        .withBody(responsePayload)));
```

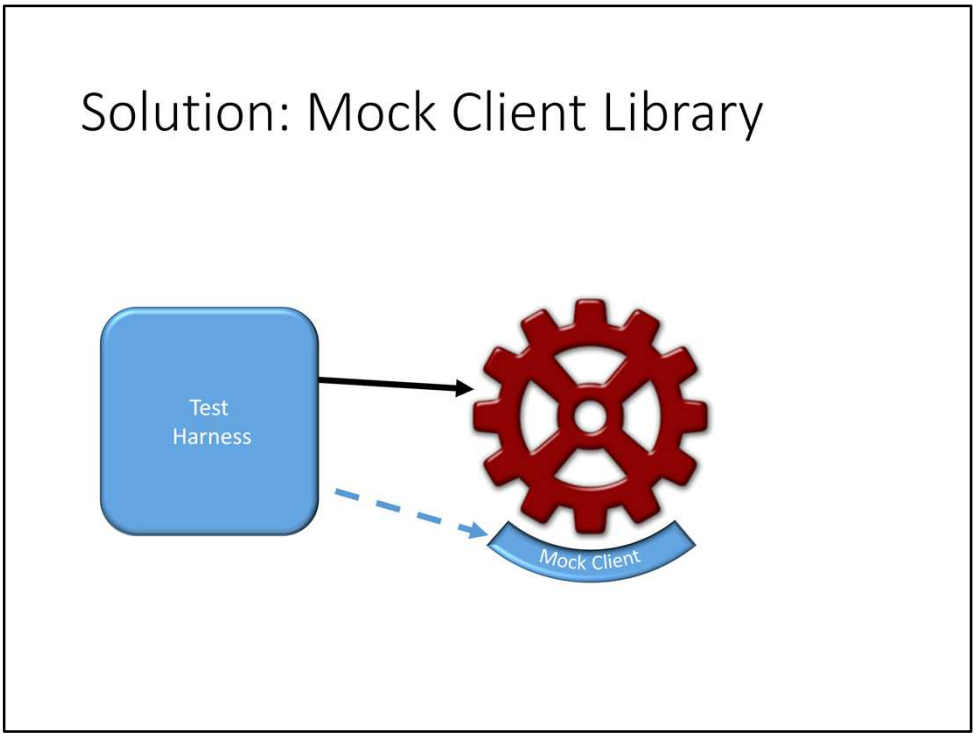
D:\Users\Sven\DiData\code\oec-sl-adapter\trunk\oec-sladapter-facade\src\test\java\com\dimensiondata\sciencelogic\adapter\facade\internal\SlGetDeviceMediatorTest.java



Let the test harness create messages and input files. It receives messages and files, and may in turn provide new files and messages as response to them.

Proprietary Communication

- Protocol is hidden in a client library.
- Protocol is verbose and complex.



The mock client library is linked in with the service in place of the real client library. It allows for a back door to control its behaviour, i.e. verify calls to the library, generate response values or inject error conditions.

Mock Client Library

- Use a back door connection for:
  - Response data
  - Inspecting request data
  - Control exceptions
- Backdoor runs in its own thread.
- May require care when serializing data.
- Use same technology as is used for the service's inputs.

Serializing: Show AsrCommunicationLogWrapper
Used in MockN2AdapterEndpoint
The mock itself MockAsrFacade

Application state

Stateless	**Stateful**
State kept in some dependent service.	State kept in application.
No need to set up state before each test case.	Run application functionality to set state correctly for each test.
Run test cases in any order.	or ... Run tests in specific order to use state set in previous tests.

We prefer stateless applications. These are much easier to test.

Database Usage

- Database stores state.
- Database as part of the application?
- Resetting content between tests.
- Mocking Database Access Layer

Database and Application

“Database is a part of the application”

ORM make database schema tightly coupled to the application.

Don’t let schema bleed into the test code.

“Database is a dependent service”

Database developed independently of application internal structures.

Test code is allowed full access to database

Database Preparation

- Use a golden image of database for quick start. May contain lots of static data.
- Recreate schema and data
- Recreate data only
- Roll-back data changed during test

- Prepare test specific data
- Tests that don't update database

Reset DB: AbstractApi2FuncTestBase.resetDatabase()

Inputs and outputs

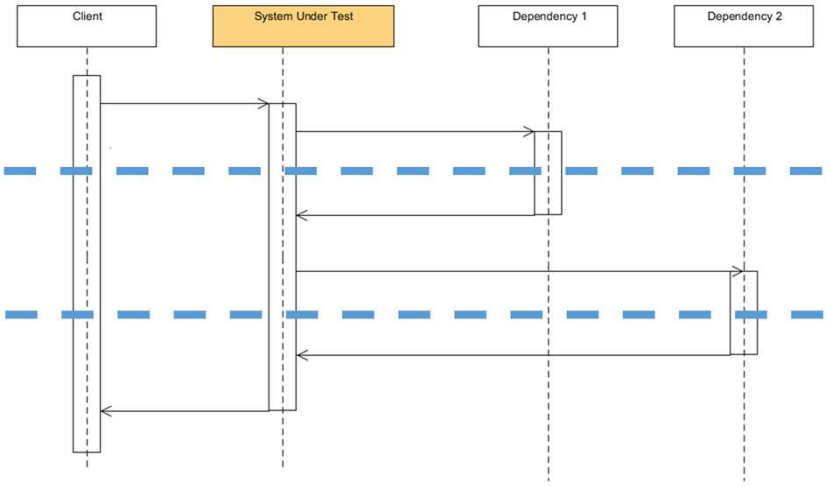
- For a stateless service
- Identify input stimuli and the side effects they cause.
- Split each scenario to include one stimulus only.

See ServerRestApi2FuncTest
deployServerRequest()
sendDeployServerCompletedMessage()

Input Stimuli

- HTTP requests
- Messages
- Database triggers
- File creation

Minimize test scenario scopes



Example bank transactions. A transaction may require several steps for example customer verification.
Split a scenario into sub-scenarios, each starting with a single input stimuli. Test the actions from the system under test.