

# CODING VS. SCRIPTING: WHAT IS A TEST CASE?

Sam Bedekar - [sam.bedekar@microsoft.com](mailto:sam.bedekar@microsoft.com)

Julio Lins - [julio.lins@microsoft.com](mailto:julio.lins@microsoft.com)

Microsoft Corporation

# Agenda

- ⦿ Background
  - Definition of a Test Case
  - Test cases for Microsoft Office Communicator
    - Environment
    - Infrastructure
- ⦿ Implementing test cases through scripts
  - SLS: Simple Language for Scenarios
- ⦿ Transitioning to C#
  - Design and implementation
- ⦿ Comparison
  - What we learned from both experiences

# What is a test case?

- A set of instructions that...
  - Executes actions in sequence on a component
  - Verifies actual results against expected results
- Any form factor that meets the above principles
  - Scripts, compiled language, interpreted XML, etc.
- Form Evolves Over Time
  - We have been writing test cases for several versions of Microsoft Office Communicator\*
  - Initially using a script language
  - Recently using C#

\*recently Microsoft [Lync](#)

# Testing Communicator

- Scenarios are distributed across multiple client endpoints/machines
  - Test driver controls multiple endpoints to orchestrate scenarios
- Multi-platform: Desktop, IP Phone, Mobile
- Scenarios are asynchronous
  - Execute action typically sends a request to the server
  - Verify response behavior through events on multiple endpoints
- Typical test case
  - 1. Endpoint-A makes a call to Endpoint-B.
  - 2. Endpoint-B accepts the call.
  - 3. Verify the call is connected on Endpoint-A.
  - 4. Verify the call is connected on Endpoint-B.

# Test code requirements

- Reuse of test code
  - Across different scenarios
  - Performance measurements
  - Across different platforms
  - Test Modes - Different modes of the product code behave slightly differently
- Rapid test case development
  - Easy orchestration of multiple endpoints
  - Reduced redundancy between test cases
- Logging
  - Detailed information of complex events and method calls
- Metrics
  - **Reduce** Mean Time to Write Test Case
  - **Increase** Mean Time Before Failure for a Test Case
  - **Reduce** Mean Time to Fix a Failure in an executed test case

# Infra-structure elements

- Endpoint hosting
  - Remoting Service: listens to a port and bootstraps endpoint host(s)
  - Endpoint host: loads test and product components and binds root object
- Remote communication
  - .Net Remoting for Desktop
  - In-House implementation for Mobile
- Event storage and comparison
  - When an event is raised, data is collected and stored on a queue to be consumed by the test case
- Thread switching
  - Some scenarios require switching to the product code thread
- Ultimately, Individual Test Cases (and their writers) do not need to worry about Remote Endpoint Hosting or Communications, Thread Dependencies, etc.

# Pre and post execution

## ⦿ Pre-Execution

- Logging of execution information (method name, argument values, event data)
- Custom remote communication
- Evaluation in API Testing mode (loop with different argument values)
- Thread synchronization and switch

## ⦿ Post-Execution

- Logging results
- Cleanup code

\*Implementation uses the C# [RealProxy](#) class

# SLS – Simple Language for Scenarios

- ⦿ Allows test driver to control multiple endpoint objects
  - Action execution and event checking
- ⦿ Imperative language
  - Supports Variables, Arrays, Loops, Functions
  - Multiple operation from a single line
- ⦿ Real-time code modification and debugging via intercepting interpreter

# SLS example code

## # Create the Remotes

```
S_OK Client1 Create RTCV2Remote  
(...)  
S_OK Client1 Logon $server$ $user1$ $password1$  
(...)
```

Initialization

## # Create a basic 2 party IM session

```
S_OK Client1 Invite $user2$ Session1  
S_OK Client2 ^CheckSessionEvent SESSION_INCOMING  
S_OK Client2 AnswerCall $user1$ Session1  
S_OK Client1 ^CheckSessionEvent SESSION_CONNECTED
```

Action

## # Have the initiator invite a third party

```
S_OK Client1 Invite $user3$ Session1  
S_OK Client3 ^CheckSessionEvent SESSION_INCOMING  
S_OK Client3 AnswerCall $user1$ Session1
```

Event verification

## # Now both client1 and 2 should get a connected event

```
S_OK Client1 ^CheckParticipantEvent PARTICIPANT_ADD  
S_OK Client1 ^CheckParticipantEvent PARTICIPANT_ADD
```

# SLS Applications

- Quick creation & execution of a new test case
  - Reuse of test code through Function calls
  - Test development is flexible with little impact between test cases
- API Testing Mode allowed the reuse of existing test cases in a SDK testing style
  - Test case is used to achieve a scenario in which a method can be called
  - Tags in the script causes the interpreter to vary arguments within a pre-determined range

```
# Have the initiator invite a third party
S_OK Client1 Logon $server$ $user1$ $password1$

@BeginAPI TestMode
S_OK Client1 CreateSession AUDIO_VIDEO Session1
S_OK Client1 Invite $user3$ Session1
S_OK Client3 ^CheckSessionEvent SESSION_INCOMING
S_OK Client3 AnswerCall $user1$ Session1
@endAPI TestMode
```

```
[SLSUnitCategory("uri", "StringSource")]
Invite(string uri, string sessionName);
```

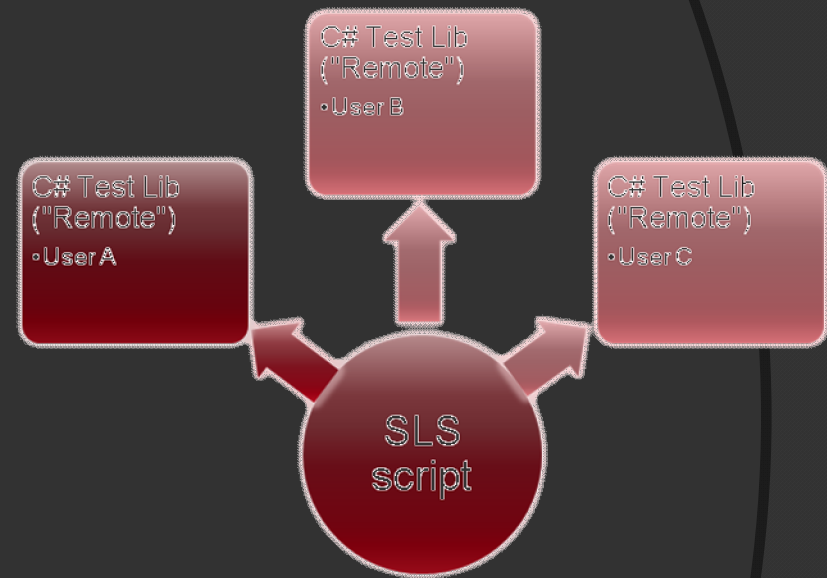
# The C# layer underneath SLS

- ◎ Test Business Layer - “Test Lib”
  - Provides a passthrough on top of the product code with light infrastructure around event queueing and repackaging return values
  - Asynchronous: events checked by test case
- ◎ Container class
  - *Facade*\* to the product object model
  - Set of wrapper methods which represent scenarios
  - Basic type methods with a standard return type
  - Maintains a table of product objects (variables in the script)

\* *Facade design pattern*

# Summary of the Architecture

- Test Lib implements basic reusable methods
- Each test case in the script combines a set of steps to form a test case
- The good
  - Flexible model can reuse the same test for a different test strategy
  - Easy creation and execution of new test cases – simple pattern
  - Common test case code is defined in SLS functions
- The bad
  - Changes in the Test Lib breaks scripts at runtime
  - Test case layer in non-standard language



# Transition period

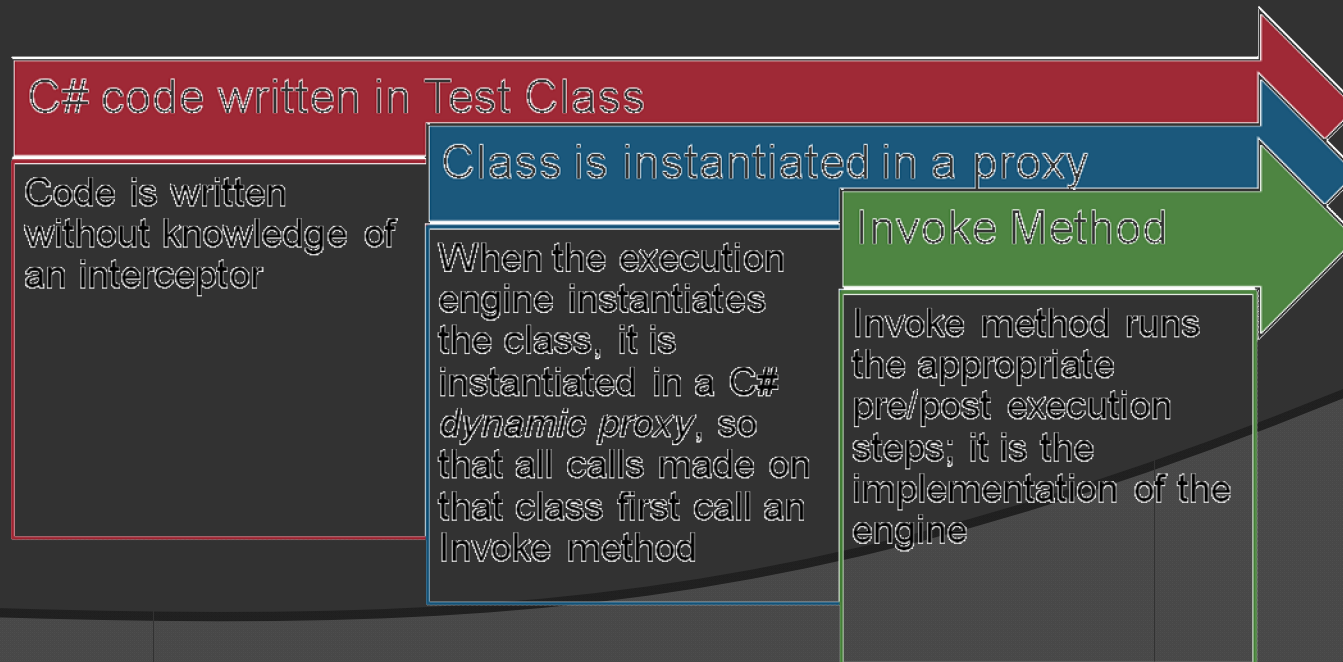
- A redesign of the product code motivated a reassessment of the test case design
  - New Communicator API would be shipped as a product
- Two products sharing similar features
  - Communicator Application
  - Communicator Platform (SDK)
- Team Size & Product Codebase increased
  - How can we share test code across teams and features reliably?
  - Easily reusable libraries
  - Breaks easily detectible

# C# test case approach

- ◎ New “Test API”
  - Object oriented
  - Synchronous view of the product object hierarchy
  - Events handled within action methods
  - Exceptions thrown in error conditions
  - Object model instead of scenario based
- ◎ Infra-structure
  - Desktop: .Net Remoting for OO communication
  - Mobile: in-house solution (similar to .Net Remoting)
- ◎ How about Pre and Post processing?

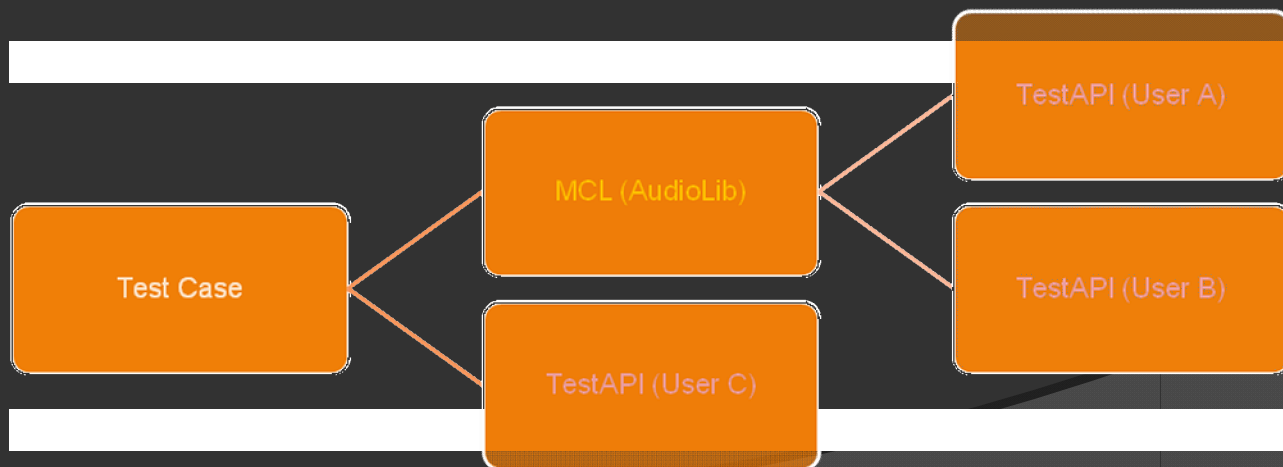
# Porting some interpreter features

- RealProxy C# class provides an intercepting mechanism
- Given that pre and post execution are available, C# test code can be written agnostic of environment details



# C# test structure

- New MCL layer (Multi-Client Library)
  - Scenario based (ex: EstablishAudioCall)
  - May involve multiple endpoints
- Test cases uses both MCL and Test API



# C# test programming model

- Three levels of reusable code
  - Test API, MCL and Test Case Helpers
- Test cases vastly simplified
  - One line to establish an audio call
- Syntactic breaks in the code are detected at build time
  
- How about semantic breaks?

# Test code as a large scale system

- Large amount of code being written
- Sharing code brings up issues characteristic to a large scale system
- Configuration management becomes a mission critical part of the lifecycle
  - Code reviews
  - Strong build system
  - Smoke Tests (testing the test cases)
  - Pre-check-in bench
    - Automatic smoke tests for both dev and test

# Comparing strategies

- ⦿ Each strategy was used at different points in time
  - This is our own experience while testing Communicator
- ⦿ Configuration management
  - Same source control in both cases
  - Automatic builds
    - SLS: only Test Lib was built
    - C#: All code is compiled (Test Case / Libraries)

# Productivity

- Given a pre-existing set of scenarios
  - SLS has the advantage of writing independent and concise test cases, and running them in an isolated fashion
  - C# requires more time to write a single test case given all its dependencies and build process
- Fully new scenarios
  - C# has the advantage of having an object model that is reusable across scenarios at each layer
  - SLS Test Lib is lightweight so most re-use is done at the test case layer

# Portability and Integration

- SLS had limited portability since it run on a in-house interpreter
  - Lack of integration with execution frameworks
  - No integration if C# components wanted to reuse SLS code
- New team members had to learn the SLS programming language
- C# framework requires shorter ramp up time
  - Standard programming language
  - Components to be learned, but no paradigm shift
- C# Integration
  - Test cases are debugged using well-known tools
  - Robust programming environment (Visual Studio)

# Conclusion

- A script language will better allow for rapid test case development while a compiled language is better suited for a long term development of reusable test code
- Pre and post processing brought numerous benefits since it brings aspects of script languages into C#
- The language is one item in the development lifecycle
  - Code and design reviews, configuration management, quality gates all play an important role in reducing the overall maintainability costs
- Ultimately, the choice to use script vs. code is based on the test situation
  - Scripting for fast lower scale solutions
  - C# for more elaborate applications

# Questions?

## ◎ Some references

- RealProxy C# class  
<http://msdn.microsoft.com/en-us/library/system.runtime.remoting.proxies.realproxy.aspx>
- Aspect Oriented Programming  
[http://en.wikipedia.org/wiki/Aspect-oriented\\_programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming)
- Facade design pattern  
[http://en.wikipedia.org/wiki/Facade\\_pattern](http://en.wikipedia.org/wiki/Facade_pattern)