Service Programs and Unit Testing – The Perfect Pair

Marina Schwenk

Software Developer

About Me

- Software Developer/IBM i admin at Everbrite LLC, Greenfield WI
- Member of the CAAC. (COMMON America's Advisory Counsel)
- 2019 IBM fresh face
- VP of WMCPA
- COMMON Board Member
- Member of COMMON's Young i Professionals (YiPS) committee

Agenda

- Scenario
- How to start
- Service programs
- Unit testing
- Standards
- Bring it all together
- Final thoughts and take aways

Scenario

- Your monolithic program is over 30+ years old
- The code is unmanageable
- No one wants to own the program
- You don’t know where to begin.

How to start

- Document Business process
- Design
- Modernize
- Develop a plan
- Executive the plan.

Document Business Process

- Document current business processes
- Meet with different departments to confirm business processes
- Document your findings.

Design

- Identify procedures that are needed
- Plan the procedures inputs and outputs
- Design the flow of how the procedures are going to be used.

Develop a plan

- Decide your approach
- Plan the timeline
- Start the project

Modernize Code

- Remove redundancy
- Develop a clear plan on data retrieval that ensures long term success
- Remove outdated code

Service Programs

- **Service Program** (*SRVPGM) can be viewed as a collection of subroutines packaged together and accessible to the outside world.

- Service programs can be thought of like classes, in the open source world.

Service Programs

- Carrying out a routine function.
- External procedures that can be called from other programs.
- You can add/change procedures as needed.

Service Programs

- Single code base.
- Easy to use and reuse.
- The ability to add to the service program without recompiling programs that are using it.

Service Programs

Service programs required objects

*BND

*SRVPGM

Optional

*MOD

Service Programs

- Service programs required source
- Copy file _h
- RPGLE or SQLRPGLE source file.
- Binding source

Copy File

- **free
- dcl-pr getArTransactionId varChar(20);
- releaseNumber packed(7) const;
- releaseSequenceNumber packed(5) const;
- end-pr;

Service Program – Start

- **FREE
- ctl-opt bnddir('EVBLOG’ : 'TEXTUTILS');
- ctl-opt nomain;

Procedure

```
dcl-proc getArTransactionId export;

  dcl-pi *n varChar( 20 );
    releaseNumber         packed( 7 ) const;
    releaseSequenceNumber  packed( 5 ) const;
  end-pi;

  dcl-c PROCEDURE_NAME 'getArTransactionId';

  dcl-s result              varChar( 20 );
  dcl-s customerRelease     char(    20 );
  dcl-s customerReleaseNull  int(     5 );
  dcl-s projectId           packed(   7 );
  dcl-s projectIdNull       int(      5 );

  evblog_entering( RELEASE_APPNAME : psds.PROGRAM_NAME : PROCEDURE_NAME
                 : 'releaseNumber=' + %char( releaseNumber ) +
                   ', releaseSequenceNumber=' + %char( releaseSequenceNumber ) );

  result = '';

  - Continue on next slide
```

Procedure continued

```
exec sql
    select rh.cusrl, rx.prjid
    into :customerRelease:customerReleaseNull, :projectId:projectIdNull
    from rlshdr rh left join rlshdx rx
        on rh.rlsno = rx.rlsno and rh.rlssq = rx.rlssq
    where rh.rlsno = :releaseNumber
    and rh.rlssq = :releaseSequenceNumber;

  if ( isSQLError( sqlstt ) );

    if ( not isSQLRowNotFound( sqlstt ) );

      evblog_log( RELEASE_APPNAME
                : psds.PROGRAM_NAME
                : PROCEDURE_NAME
                : EVBLOG_WARNING
                : 'SQL Error'
                : getSQLStateMessage() );

    endIf;
```

Continue on Next slide

Procedure continued

```
else;

    // information was returned

    if (( projectIdNull = SQL_NOT_NULL ) and ( projectId <> 0 ));
      result = 'P' + %char( projectId );
    elseIf ( customerReleaseNull = SQL_NOT_NULL );
      result = %trim( customerRelease );
    endIf;

  endIf;

  evblog_exiting( RELEASE_APPNAME : psds.PROGRAM_NAME : PROCEDURE_NAME
                : result );

  return result;

end-proc;
```

What is Unit Testing?

- Breaking apart your application and testing each part
- It’s a program that will call your production program/procedure.
- Test its behavior and/or output.
- Separate pieces that gets tested before the final program is completed.

Why Unit Test?

- Peace of mind.
- Good for program modification's.
- Good for defining what your program needs to do, before you write the program.
- Test cases will build over time.

Why Unit Test?

- Improved Code.
- Validates existing behavior.

What is IBMiUnit?

- RPG open source unit testing framework
- Streamlines unit testing of RPGLE programs and procedures

How to Use IBMiUnit

- Install IBMiUnit Library
- Create test program
- Write one or more tests
- Run the tests

Installation

- Go to https://github.com/MarinaSchwenk/IBMiUnit
- 1st way…
- Download the REPO
- Run the Build file
- 2nd way…
- Download the savf file to IFS
- Create library IBMiUnit
- Restore IBMiUnit library.

Dependencies

- OSSILE (Soon to be obsoleted)
  - Go to https://github.com/OSSILE/OSSILE
  - Download the repo
  - Follow the build instructions listed on the project.

Create Test Program

- No parameters
- `bndDir( 'IBMiUnit' )`
- `/copy IBMiUnit/QRPGLESRC,IBMiUnit_H`
- Main body of the program
  - Call `IBMiUnit_setupSuite()` (one-time)
  - Call `IBMiUnit_addTestCase()` (for each test case)
  - Call `IBMiUnit_teardownSuite()` (one-time)
  - return

IBMiUnit Initialization

- IBMiUnit_setupSuite()
- Initializes the IBMiUnit library to run tests in the program
- Parameters (all optional)
  - Name for the test suite
  - Address of sub-procedure to call before each test
  - Address of sub-procedure to call after each test
  - Address of sub-procedure to call once before any tests in the program are called
  - Address of sub-procedure to call once after all tests in the program are completed
- Example

```
IBMiUnit_setupSuite( 'MathTests' );
```

IBMiUnit Test Case

- IBMiUnit_addTestCase()
- Identifies or ‘links’ a test case into the suite
- Parameters (no return value)
  - Address of a test case sub-procedure
    - No parameters or return values
  - Name of test case; optional but greatly helps you understand the test output and find the problem
- Example

```
IBMiUnit_test( %pAddr( add_twoNumbers )
             : 'add_twoNumbers' );
```

IBMiUnit Test Suite

- IBMiUnit_addTestSuite()
- Adds a set of test cases (suite) to a parent set
- Not all test programs will use this
- Parameters
  - Name of test program
  - Library of test program
    - Optional, defaults to `*LIBL`
- Example

```
IBMiUnit_addTestSuite( 'TEST_ACHAR' );
```

IBMiUnit Teardown

- IBMiUnit_teardownSuite()
- Wraps up test suite
- No parameters
- Every `IBMiUnit_setupSuite()` needs a corresponding `IBMiUnit_teardownSuite()`
- Example

```
IBMiUnit_teardownSuite();
```

Write a Test Case: Interface

- Sub-procedure without parameters or a return value
  - Name the test case with the name of the sub-procedure
  - Example
    - Calling the sub-procedure with positive values
    - Test case named `multiply_twoPositives`
    - Other test case names: `multiply_positiveByZero`, `multiply_zeroByZero`, `multiply_twoNegatives`, …

Write Test Case: Logic

- Call sub-procedure with test data
- Compare actual result with the expected result
  - Trigger failure when they don’t (or do) match

Write Test Case: Failure Detection

- Always fail, i.e. you write the condition and call `fail()`
- Conditionally fail, or test for failure; many possibilities
  - All start with `assert`
  - Indicator tests
    - `On` / `Off`
  - Pointer tests
    - `Null` / `NotNull`
  - Variable tests / comparisons
    - RPG doesn't have overloading so next word is a type
      - `Char`, `Date` (ISO), `Float`, `Numeric`, `Time` (ISO), `Timestamp`
        - Character tests work on values up to a length of 250
        - Numeric is used for non-float numbers; size is 60,25

Write Test Case: Test Procs

- `fail()`
  - Message to display; optional
- `assertOn()`,`assertOff()`,`assertNull()`,`assertNotNull()`
  - Actual value; required
  - Message to display on failure; optional
- `assertFloatXxx()`
  - Expected value; required
  - Actual value; required
  - Delta (leeway; allowable difference); required
  - Message to display on failure; optional
- `assertXxx()` (everything else)
  - Expected value; required
  - Actual value; required
  - Message to display on failure; optional

Write Test Case: Examples

```
if ( %scan( 'TEST' : value ) <> 1 );
   fail( 'value does not start with TEST' );
endIf;

assertOn( rowFound, 'Row not found' );

assertCharEquals( expected, actual, 'Name' );

assertNumericEquals( 12.00, extendedAmount, 'Item price' );

assertDateEquals( today, invoiceDate, 'Invoice date' );
```

Run Tests

- Call IBMiUnit command.
- Example
  - `IBMiUnit/RUNTEST SUITE(TEXTUTIL_T) UI(*DSPLY)`
  - No feedback if all tests are successful
  - Helps you focus on the problems

Result Status

- Successful
- Failure
  - A state detected by your code
  - From `fail()` or `assertXxx()`
- Error
  - A problem encountered, but not detected, by your code
  - Level check, divide by 0, array index out of bounds, …

Standards

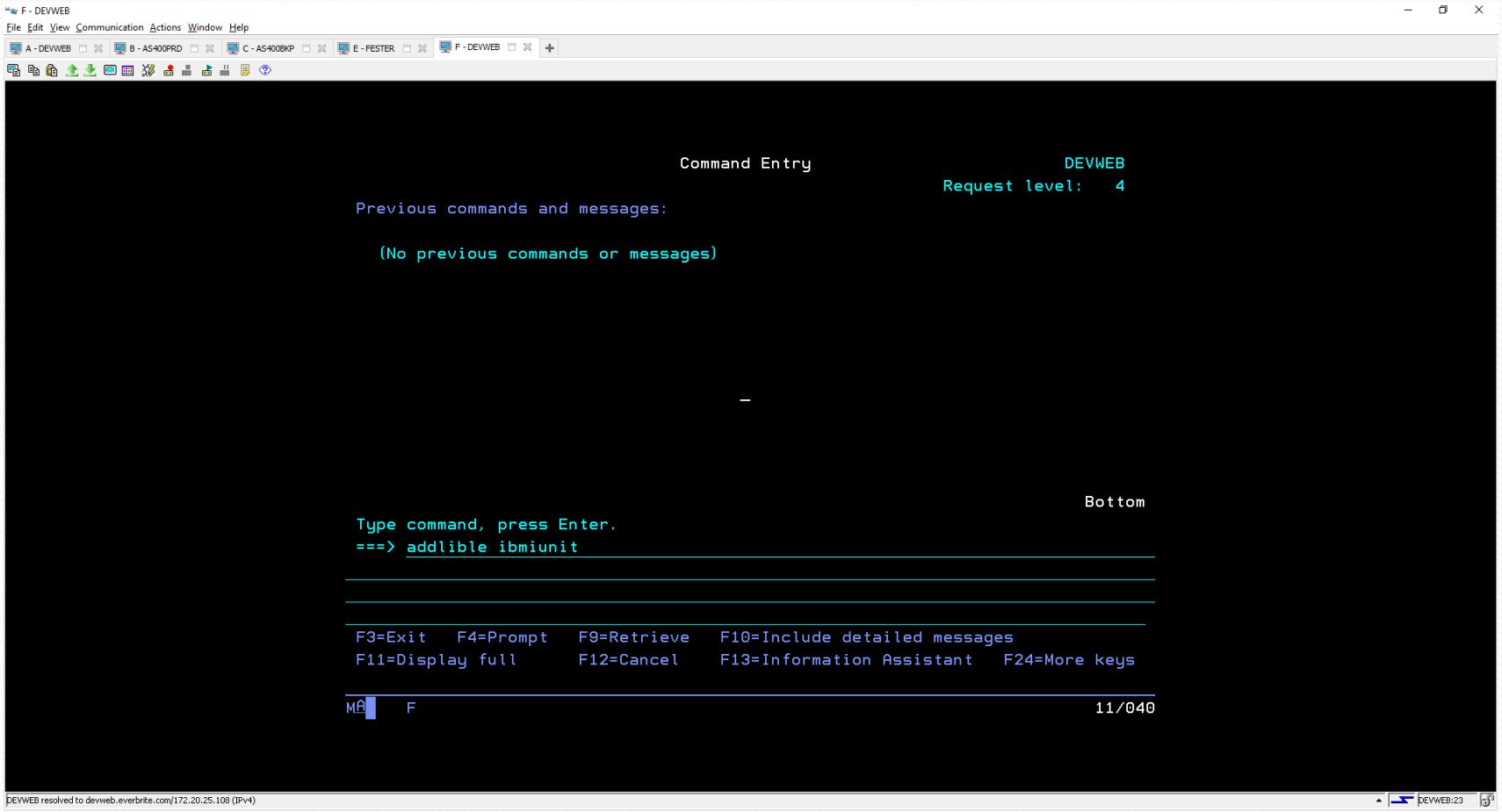
- Keep test code separate from production code.
- Keep Service programs clean
  - Don’t hard code.
  - Keep comments clear and concise
  - Keep code consistent.
- Keep Naming conventions the same
  - Orders Service Program = Orders_T Testing program.

Bring it all together

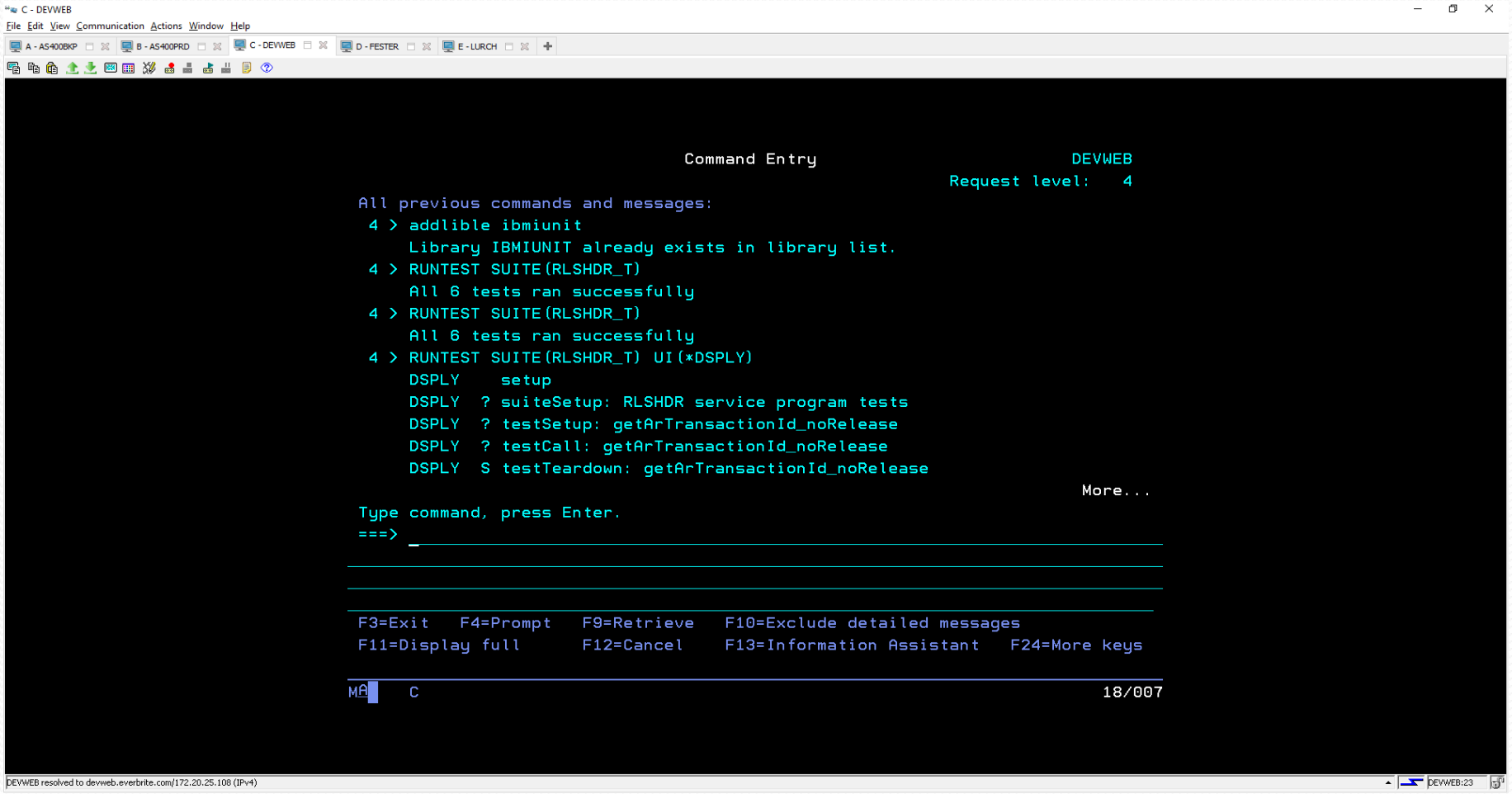
- Old code has been modernized and brought into a service program.
- Created new methods and new program to call the methods.
- Created testing program to test new methods.
- Replace the calling programs with a call to the method or the new program.

Demo Time!

Add IBMiUnit library



Run IBMiUnit interactive



Final Thoughts..

- Service programs are very easy to adopt.
- By having service programs you can easily incorporate unit testing.
- It’s a process to modernize, its worth it in the end.

Thank you!!

My contact Info

Marina Schwenk

- marinaschwenk23@gmail.com
- @marinaschwenk26 on twitter