Integration test

1. Grab modules (packages / classes?)and create objects

use Test::MockModule;

    $atxMock = Test::MockModule->new('BERT::ATX', no\_auto => 1);

1. Prepare Tests – create “mocks” of various subs so that if they are called by the function to be tested, they can be set to return fixed info:

$atxMock->mock('milestoneSort' => sub{

    return {

        DATA => {

                'CLOSED' => 0,

                'CRITICAL\_DATE' => '2011-06-06',

                'CRITICAL\_DATE\_TYPE' => 'DD',

                'ESCALATION\_OWNER\_NAME' => 'ANGELICA SCHWARTZ',

                'JEP\_CODE' => ' '

        },

    STATUS => 1

    };

});

This one is also integral to the escalation process – so we’re mocking all subs that are expected to return something during the process

$atxMock->mock(autoEscalationInsert => sub{

    my ($self, $args) = (@\_);

    my $dbh = createMockDatabase();

    my $toReturn = {

        STATUS => 1

    };

    return $toReturn;

});

Some subs don’t need to be called because, like verifyInput, they simply process / make / return decisions based on the inputs recieved – extra calls are not required –

First, mock these apis; that is, create a simulated API that will send back fixed data.

* Sopapmock – allows one to build SOAP clients and servers.
* LWPmock – LWP is a library that lets you write clients and implement simple HTTP servers

And

* Parseominfo – related to IEOL, which verifies escalatable states for orders with USO’s (like atx)

Then the following mocks are created (why these?) they are associated with certain subroutines:

* Notify
  + Notification = updatefollower
  + Notification – Addfollower
  + Gettechdesigner
* Validate
  + Milestonesort –
  + Isactive
  + Getinfo / Efmsapicall
* Escalate
  + Autoescalationinsert

1. Run Tests – tests are held in brackets. They typically consist of these parts:
   1. Calls mocksoap() – in the destructive tests sectin ---
   2. Result – call the REAL sub passing in arguments expected – listed in the ‘expected keys’ of verifyinput. It’s assumed that the data fed actually exists in the DB, so ticket number X will allow a person to access all the details related with it?
   3. Data – what you expect to get back – these details are present in the concluding ‘return’ of the sub
   4. Assert – the comparison

List of tests – each of the following functions are tested for specific abilities – if escalate is called, then along the way, mocks will feed the process certain details needed to finish.

Why did I have to include a mock of autoEscalationInsert and not gettypenamefromtypeidresult?

* + IEOL data-> getinfo() – during this sub, getorderinfo, verifyOCO and \_parseEFMS are called. Hence, the mocks efmsApicall for the latter –
  + Validate – this calls several subs, hence the mocks isactive, milestonesort, getinfo/efmsapicall.
    - No data in db
    - Data is ‘seeded’ – validate, getlastescinfo, getTechAttuid
    - Seeded w/bad time
  + Escalate
    - No data in db – mocks addfolower, storeescalation, notify. Call escalate, unmock.
    - Data present – mock dmaap, addfollower, watsonapicall / call escalate, unmock
    - dmaap fails. – mock pubtodmaaap, call escalate, getidforactiveorder,
    - No notes in arguments
  + To\_type
    - Contacts available
    - No contacts
    - No general and duty cotacts
  + \_updateOrderEsc –
    - no data in db
    - live data
  + updateOrder and closeOrder
    - good value update for atx cron case
  + updateApprovalType – 989 – called by bertAPI and ticketview.pl?

1. destruct tests – helper funtions
   1. seedDB
   2. unseedDB
   3. mockSOAP
   4. seedSpecificEscalateTest
   5. unspecificescalatetest
   6. seedupdateorder
   7. unseedupdateorder
   8. setEFMSforupdate
   9. setefmsfor closed

s

1. finish testing
   1. dbconnectBERT

Questions

1. Why assert at the start of the second test and what is unseeddb?
2. Why these tests?