CASE STUDIES OF SUMMER MODEL-BASED TESTING FRAMEWORK

V. Kuliamin, N. Pakulin, A. Tugaenko
Institute for System Programming, Russian Academy of Sciences
OUTLINE

- Description of the tool
  - Intended domain
  - Main features
  - Similarities and differences with other tools

- Case studies
  - SMPT protocol
  - Part of DOM API
  - Part of Google Web UI
THE TOOL

- Summer
  Test development and execution framework

- Features
  - Java API testing (and where possible to link up)
  - xUnit-like test presentation
  - Unit – Component levels
  - Black-box testing
  - MBT features
    - EFSM-based testing
    - Software contracts as test oracles
@Test
public class TestClass {

    Account target = new Account(); // object under test

    @Test
    public void testDeposit() {
        int oldBalance = target.getBalance();
        int deposited = 5;

        target.deposit(deposited);
        assertEquals(target.getBalance(), oldBalance + deposited, "Balance should grow by deposited sum");
    }
}
@Test
public class TestClass {

    Account target = new Account();

    @State
    public int balance() { return target.getBalance(); }

    int[] sums = new int[]{0, 1, 2, 3, 5, 17, 238};
    public boolean bound() { return balance() < 350; }

    @Test
    @DataProvider(name = "sums")
    @Guard(names = "bound")
    public void testDeposit(int sum) { ... target.deposit(sum); ... }

    @Test
    @DataProvider(name = "sums")
    public void testWithdraw(int sum) { ... target.withdraw(sum); ... }
}
SIMILAR TOOLS AND ADDITIONS

- **TestNG** (one of most powerful xUnit tools, Java)
  - Testware hierarchy: test suites – tests – test classes – test methods
  - Setup-teardown methods on all hierarchy levels
  - Test methods grouping and selection by groups
  - Test methods sequencing
  - Test data providers

- **NModel** (MBT tool extending xUnit, C#)
  - State-based testing
  - Guard conditions
  - Compositions of test classes

- **Additions**
  - Stateful software contracts (described separately of tests)
  - Aspect-based linking of external components: contracts, coverage models, etc.
  - Combinations of data providers for parameters
  - More flexible data providers, guard conditions, state definitions
public class AccountContract {
    int balance;
    int maxCredit;

    public boolean withdrawPost(int sum) {
        if (Contract.oldValue(balance) - sum > maxCredit)
            return assertEquals(Contract.intResult(), sum
            , "Result should be equal to the argument")
            && assertEquals(balance, Contract.oldValue(balance) - sum
            , "Balance should be increased on the argument");
        else
            return assertEquals(Contract.intResult(), 0
            , "Result should be 0")
            && assertEquals(balance, Contract.oldValue(balance)
            , "Balance should not change");
    }
}
<xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." >
  ...
  <bean id="accountContract" class="mbtest.tests.account.AccountContract">
    <property name="checkedObject" ref="accountImpl"/>
  </bean>

  <bean id="accountContractExecutor" class="mbtest.contracts.ContractExecutor">
    <property name="postcondition"
      value="mbtest.tests.account.AccountContract.withdrawPost(int)"/>
    <property name="updater"
      value="mbtest.tests.account.AccountContract.transferUpdate"/>
    <property name="contract" ref="accountContract"/>
  </bean>

  <aop:config>
    <aop:aspect id="accountContractAspect" ref="accountContractExecutor">
      <aop:pointcut id="accoutTransfer"
        expression="execution(* mbtest.tests.account.Account.withdraw(..))"/>
      <aop:around pointcut-ref="accoutTransfer" method="execute"/>
    </aop:aspect>
  </aop:config>
</beans>
CASE STUDIES

- **Testing**
  - SMTP protocol implementations (against SMPT RFC)
  - Part of Xerces DOM implementation (against DOM API standard)
  - Part of Google WebUI (against simple intuition)

- **Main goal**: to check the following ideas
  - **Idea 1**
    Flexibility of component architecture facilitates usage of generic tools in various domains
  - **Idea 2**
    Modular testware (separate components: test oracles, test sequence generator, test data generators, test coverage measurement) helps to achieve comprehensive testing with less effort
SMPT CASE STUDY

- Simple Mail Transfer Protocol
  RFC 5321 [2008]

- Client
  - Basic actions:
    <connect>
    [HELO | EHLO] ...
    ( MAIL FROM: <...> (RCPT TO: <...>)+ DATA (<line>)*)+ QUIT
  - Additional: NOOP, RSET { VRFY, EXPN, HELP }

- Server responses: [2-5][0-2|5][0-9] ...

- Extensions
  RFC 4954 (AUTH) { RFC 1652, 1879, 2034, 2920, 3030, 3207, 3461, 3463, 3865, 3885, 4095, 4405, 4865, 4954, 5336 }
MODULAR TEST MODEL

- Separate CONNECT-DISCONNECT test model
  - Can be used with other over-transport protocols
- Separate basic SMTP test model
- Separate AUTH PLAIN test model
- Possibility to add other extensions
TEST STATE MACHINE (SLIGHTLY SIMPLIFIED)

[HELO | EHLO]/250

.../5xx, RSET

MAIL /2xx

RCPT /2xx

DATA /2xx

QUIT /2xx

[HELO | EHLO]/250
TEST STATE MACHINE WITH AUTH PLAIN

[HELO | EHLO]/250

.../5xx, RSET

[HELO | EHLO]/250

MAIL /2xx

RCPT /2xx

MAIL /2xx

. /2xx

RCPT /2xx

DATA /2xx

<line>

<line>

USER/2xx

PASS/2xx

QUIT /2xx

LOGIN/2xx
<table>
<thead>
<tr>
<th>Testware module</th>
<th>Test model</th>
<th>Contract</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect-Disconnect</td>
<td>90</td>
<td>140</td>
<td></td>
<td>230</td>
</tr>
<tr>
<td>Basic SMTP</td>
<td>200</td>
<td>480</td>
<td></td>
<td>680</td>
</tr>
<tr>
<td>Authentication</td>
<td>140</td>
<td>300</td>
<td></td>
<td>440</td>
</tr>
<tr>
<td>Auxiliary</td>
<td></td>
<td></td>
<td>680</td>
<td>680</td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
<td></td>
<td>230</td>
<td>230</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>430</strong></td>
<td><strong>920</strong></td>
<td><strong>910</strong></td>
<td><strong>2260</strong></td>
</tr>
</tbody>
</table>
DOM CASE STUDY

- DOM API Standard
  - Document Object Model – internal representation of web pages in browsers
- Node interface
- SUT – Xerces for Java [xerces.apache.org]
appendChild modified in DOM Level 3

 Adds the node newChild to the end of the list of children of this node. If the newChild is already in the tree, it is first removed.

 Parameter

 newChild of type Node [p.56]

 The node to add.

 If it is a DocumentFragment [p.40] object, the entire contents of the document fragment are moved into the child list of this node.

 Return Value

 Node [p.56] The node added.

 Exceptions

 DOMException [p.31]

 HIERARCHY REQUEST ERR: Raised if this node is of a type that does not allow children of the type of the newChild node, or if the node to append is one of this node’s ancestors [p.265] or this node itself or if this node is of type Document [p.41] and the DOM application attempts to append a second DocumentType [p.115] or Element [p.85] node.

 WRONG DOCUMENT ERR: Raised if newChild was created from a different document than the one that created this node.

 NO MODIFICATION ALLOWED ERR: Raised if this node is readonly or if the previous parent of the node being inserted is readonly.

 NOT SUPPORTED ERR: if the newChild node is a child of the Document [p.41] node, this exception might be raised if the DOM implementation doesn’t support the removal of the DocumentType [p.115] child or Element [p.85] child.
public boolean appendChildNormalPost(Node n) {
    boolean common =
        Contract.assertNotNull(parent, Contract.<Node>oldValue(parent),
            "Parent node should be preserved")
        && Contract.assertNotNull(owner, Contract.<Node>oldValue(owner),
            "Owner node should be preserved")
        && Contract.assertNotNull(Contract.<Node>result), n,
            "[N4]" + " The node added should be returned")
    if (n instanceof DocumentFragment)
        List<Node> appendedNodes = Contract.<List<Node>>oldValue().toList(n.getChildNodes());
    for (int i = 0; i < appendedNodes.size(); i++)
        common &= Contract.assertTrue(target.isSameNode(appendedNodes.get(i).getParentNode())
            , "[N1]" + " This node should become parent for all appended nodes");
    return common
        && Contract.assertNotNull(n.getChildNodes().getLength(), 0,
            "The new contents of the document fragment should be empty")
        && Contract.assertNotNull(children, appendedNodes.size(), appendedNodes
            , "[N1]" + " The entire contents of the document fragment should be moved in the end of the child list of this node");
    common &=
        target.isSameNode(n.getParentNode())
        , "[N4]" + " This node should become appended node parent")
        && Contract.assertNotNull(children, n,
            "[N3]" + " The node added should be appended to the list of children");
    Node oldParent = Contract.<Node>oldValue(n.getParentNode());
    if (target.isSameNode(oldParent))
        int ind = Contract.<Integer>oldValue(children.indexOf(n));
        if (ind == 0)
            return common
                && Contract.assertNotNull(children.subList(0, children.size()-1), Contract.<Object>oldValue(new ArrayList<>(children)
                    , "All other children should be preserved")
            else if (ind == Contract.<Integer>oldValue(children.size()-1))
                return common
                    && Contract.assertNotNull(children, Contract.<Object>oldValue(children.size()-1)
                        , "All other children should be preserved");
    node = target.appendChildNormalPost(n);
    return contract.x"
HIERARCHY_REQUEST_ERR: Raised if this node is of a type that does not allow children of the type of the newChild node, or if the node to append is one of this node’s ancestors or this node itself, or if this node is of type Document or the DOM application attempts to append a second DocumentType or Element node.
WEB APPLICATION CASE STUDY

- Google Web UI

- No ready Java API so, use WebUI Driver
  - Selenium RC
  - http://seleniumhq.org/
@Test public class GoogleTest {
    Selenium sdriver;
    int page = 0;
    @State public int getPage() { return page; }

    @BeforeTest public void init() {
        sdriver = new DefaultSelenium("localhost", 4444, "*firefox", "http://www.google.com");
        sdriver.start();
        sdriver.open("/");
    }

    boolean containsTransitionTo(int n) { return sdriver.isElementPresent("link=" + n); }

    @Test @DataProvider(name = "pageNumber")
    @Guard(names = {"isResultsPage", "containsTransitionTo"})
    public void moveToPage(int n) {
        sdriver.click("link=" + n);
        if(n != 1) sdriver.waitForCondition("selenium.isTextPresent('Page " + n + " of about results')==true", "20000");
        else sdriver.waitForCondition("selenium.isTextPresent('Page " + page + " of about results')!=true", "20000");
        page = n;
        commonCheck();
    }
}
CONCLUSION

- Idea 1
  Flexibility of component architecture facilitates usage of generic tools in various domains
  - Seems to be true: API components, protocols, Web UI can be tested by uniform mechanisms

- Idea 2
  Modular testware helps to achieve comprehensive testing with less effort
  - Not enough data, maybe wrong
  - Note: tests’ maintainability requires additional effort, but modular tests are more maintainable by construction
  - So, such an idea seems to be formulated inadequately
Thank you for attention!