Automating the Testing Oracle

Steven J Zeil

February 13, 2013
Outline

1. Automating the Oracle
   - How can oracles be automated?
   - Examining Output
   - Limitations of Capture-and-Examine Tests

2. Self-Checking Unit & Integration Tests

3. JUnit Testing
   - JUnit Basics

4. C++ Unit Testing
   - Google Test
   - Boost Test Framework
Earlier we introduced this model of the basic testing process.

- In this lesson, we turn our attention to the oracle
Outline I

1 Automating the Oracle
   - How can oracles be automated?
   - Examining Output
   - Limitations of Capture-and-Examine Tests

2 Self-Checking Unit & Integration Tests

3 JUnit Testing
   - JUnit Basics

4 C++ Unit Testing
   - Google Test
   - Boost Test Framework
Why Automate the Oracle?

- Most of the effort in large test suites is evaluating correctness of outputs
  - The *oracle* is a decision procedure for deciding whether the output of a particular test is correct.
  - Humans ("eyeball oracles") are notoriously unreliable
    - better to have tests check themselves.
  - Regression test suites can be huge.
- Modern development methods emphasize rapid, repeated unit tests
  - *Test-driven development*: Develop the tests *first*, then write the code.
  - Debugging: if you can’t reproduce the error, how will you know when you’ve fixed it?

But that’s just not going to happen if testing is hard to do.
Output Capture

If we are doing systems/regression tests, the first step towards automation is to capture the output:
If output is to:

- Standard output: diff or cmp on standard output
Output Capture

If we are doing systems/regression tests, the first step towards automation is to capture the output:

If output is to:

- Standard output: `diff` or `cmp` on standard output
- Capture outputs for system-level regression tests
Output Capture

If we are doing systems/regression tests, the first step towards automation is to capture the output:
If output is to:

- Standard output: diff or cmp on standard output
- Capture outputs for system-level regression tests
- Databases
If we are doing systems/regression tests, the first step towards automation is to capture the output: If output is to:

- **Standard output**: `diff` or `cmp` on standard output
  - Capture outputs for system-level regression tests
- **Databases**
  - Clone and compare, or
Automating the Testing Oracle

Automating the Oracle

How can oracles be automated?

Output Capture

If we are doing systems/regression tests, the first step towards automation is to capture the output:

If output is to:

- **Standard output:** `diff` or `cmp` on standard output
  - Capture outputs for system-level regression tests
- **Databases**
  - Clone and compare, or
  - Use queries to search for changes
Output Capture

If we are doing systems/regression tests, the first step towards automation is to capture the output:
If output is to:
- **Standard output**: `diff` or `cmp` on standard output
  - Capture outputs for system-level regression tests
- **Databases**
  - Clone and compare, or
  - Use queries to search for changes
- **Graphics (screen)** – difficult
Automating the Testing Oracle

Automating the Oracle

How can oracles be automated?

Output Capture

If we are doing systems/regression tests, the first step towards automation is to capture the output:

If output is to:

- **Standard output**: `diff` or `cmp` on standard output
  - Capture outputs for system-level regression tests
- **Databases**
  - Clone and compare, or
  - Use queries to search for changes
- **Graphics (screen) – difficult**
  - Screen captures hard to compare, often too finicky
Output Capture

If we are doing systems/regression tests, the first step towards automation is to capture the output:

If output is to:

- **Standard output**: `diff` or `cmp` on standard output
  - Capture outputs for system-level regression tests

- **Databases**
  - Clone and compare, or
  - Use queries to search for changes

- **Graphics (screen) – difficult**
  - Screen captures hard to compare, often too finicky
  - Design for testability
Output Capture

If we are doing systems/regression tests, the first step towards automation is to capture the output:

If output is to:

- **Standard output**: diff or cmp on standard output
  - Capture outputs for system-level regression tests
- **Databases**
  - Clone and compare, or
  - Use queries to search for changes
- **Graphics (screen) – difficult**
  - screen captures hard to compare, often too finicky
  - design for testability
    - capture “events” in a testing log
Output Capture and Drivers

At the unit and integration test level, we are testing functions and ADT member functions that most often produce data, not files, as their output. That data could be of any type. How can we capture that output in a fashion that allows automated examination?

- Traditional answer is to rely on the *scaffolding* to emit output in text form.
- A more sophisticated answer, which we will explore later, is to redesign these tests to be self-checking.
File tools

- **diff, cmp** and similar programs compare two text files byte by byte
  - used to compare expected and actual output
  - useful in *back-to-back* testing of
    - old system to its new replacement
    - system before and after a bug repair
    - but also used with manually generated expected output
  - parameters allow special treatments of blanks, empty lines, etc.
  - some versions can be used with binary files
Alternatives

- More sophisticated tests can be performed via `grep` and similar utilities
  - search file for data matching a regular expression
Custom oracles

- Some programs lend themselves to specific, customized oracles
  - For example, a program to invert a matrix can be checked by multiplying its input and output together — should yield the identity matrix.

- Pipe output from program/driver directly into a custom evaluation program, e.g.,

  testInvertMatrix matrix1.in > matrix1.out
  multiplyCheck matrix1.in < matrix1.out

  or

  testInvertMatrix matrix1.in | multiplyCheck matrix1.in

- Most useful when oracle can be written with considerably less effort than the program under test
expect is a shell for testing interactive programs.

- an extension of TCL (a portable shell script).
Key `expect` Commands

`spawn` Launch an interactive program.
Key expect Commands

**spawn**  Launch an interactive program.
**send**   Send a string to a spawned program, simulating input from a user.
**Key `expect` Commands**

- **spawn**: Launch an interactive program.
- **send**: Send a string to a spawned program, simulating input from a user.
- **expect**: Monitor the output from a spawned program. `expect` takes a list of patterns and actions:
  ```
  pattern1 {action1}
  pattern2 {action2}
  pattern3 {action3}
  ...
  ```
  and executes the first action whose pattern is matched.
Key expect Commands

**spawn**  
Launch an interactive program.

**send**  
Send a string to a spawned program, simulating input from a user.

**expect**  
Monitor the output from a spawned program. Expect takes a list of patterns and actions:

```
pattern1  {action1}
pattern2  {action2}
pattern3  {action3}
```

and executes the first action whose pattern is matched.

**interact**  
Allow person running expect to interact with spawned program. Takes a similar list of patterns and actions.
Log in to other machine and ignore “authenticity” warnings.

```
#!/usr/local/bin/expect
set timeout 60
spawn ssh $argv
while 1 {
    expect {
        eof {break}
        "The authenticity of host" {send "yes\r"}
        "password:" {send "$password\r"}
        "$argv" {break} # assume machine name is in prompt
    }
}
interact
close $spawn_id
```
puts "in test0: $programdir/testsets\n"
catch {
  spawn $programdir/testsets

  expect \
  "RESULT: 0" {fail "testsets"} \\
  "missing expected element" {fail "testsets"} \\
  "contains unexpected element" {fail "testsets"} \\
  "does not match" {fail "testsets"} \\
  "but not destroyed" {fail "testsets"} \\
  {RESULT: 1} {pass "testsets"} \
  eof {fail "testsets"; puts "eof\n"} \\
  timeout {fail "testsets"; puts "timeout\n"}
}
catch {
  close
  wait
}
Structured Output

For unit/integration test, output is often a data structure

- Data must be *serialized* to generate text output and parsed to read the subsequent input
Structured Output

For unit/integration test, output is often a data structure

- Data must be *serialized* to generate text output and parsed to read the subsequent input
- A lot of work
Structured Output

For unit/integration test, output is often a data structure

- Data must be *serialized* to generate text output and parsed to read the subsequent input
- A lot of work
  - Easy to omit details
Structured Output

For unit/integration test, output is often a data structure

- Data must be *serialized* to generate text output and parsed to read the subsequent input
- A lot of work
  - Easy to omit details
  - Can introduce bugs of its own
Structured Output

For unit/integration test, output is often a data structure

- Data must be *serialized* to generate text output and parsed to read the subsequent input
- A lot of work
  - Easy to omit details
  - Can introduce bugs of its own
- Similar issues can exist with the need to supply structured inputs
Repository Output

For system and high-level unit/integration tests, output may be updates to a database or other repository.

- Must be indirectly “captured” via subsequent query/access
Repository Output

For system and high-level unit/integration tests, output may be updates to a database or other repository.

- Must be indirectly “captured” via subsequent query/access
- Significant setup and cleanup effort per test
Repository Output

For system and high-level unit/integration tests, output may be updates to a database or other repository.

- Must be indirectly “captured” via subsequent query/access
- Significant setup and cleanup effort per test
  - Need separate test stores
Automating the Testing Oracle

Limitations of Capture-and-Examine Tests

**Graphics Output**

- For system and high-level unit/integration tests, output may be graphics
  - very hard to capture
- Similar issues can arise supplying GUI input
  - Supplying a repeatable sequence of input events (key presses, mouse movement & clicks, etc)
  - Sometimes timing-critical
Automating the Testing Oracle

Outline I

1 Automating the Oracle
   - How can oracles be automated?
   - Examining Output
   - Limitations of Capture-and-Examine Tests

2 Self-Checking Unit & Integration Tests

3 JUnit Testing
   - JUnit Basics

4 C++ Unit Testing
   - Google Test
   - Boost Test Framework
Self-Checking Unit and Integration Tests

- Addresses problem of capture-and-examine for structured data
Self-Checking Unit and Integration Tests

- Addresses problem of capture-and-examine for structured data
- Each test case is a function.
Self-Checking Unit and Integration Tests

- Addresses problem of capture-and-examine for structured data
- Each test case is a function.
  - That function constructs required inputs...
Self-Checking Unit and Integration Tests

- Addresses problem of capture-and-examine for structured data
- Each test case is a function.
  - That function constructs required inputs . . .
  - and passes those inputs to the module under test . . .
Self-Checking Unit and Integration Tests

- Addresses problem of capture-and-examine for structured data
- Each test case is a function.
  - That function constructs required inputs . . .
  - and passes those inputs to the module under test . . .
  - and examines the output . . .
Self-Checking Unit and Integration Tests

- Addresses problem of capture-and-examine for structured data
- Each test case is a function.
  - That function constructs required inputs...
  - and passes those inputs to the module under test...
  - and examines the output...
- ...all within the memory space of the running function
First Cut at a Self-Checking Test

Suppose you were testing a SetOfInteger ADT and had to test the add function in isolation, you would need to know how the data was stored in the set and would have to write code to search that storage for a value that you had just added in your test. E.g.,

```c
void testAdd (SetOfInteger aSet)
{
    aSet.add (23);
    bool found = false;
    for (int i = 0; i < aSet.numMembers && !found; ++i)
        found = (aSet[i] == 23);
    assert(found);
}
```
What’s Good and Bad About This?

```c
void testAdd(SetOfInteger aSet)
{
    aSet.add(23);
    bool found = false;
    for (int i = 0; i < aSet.numMembers && !found; ++i)
        found = (aSet.data[i] == 23);
    assert(found);
}
```

- Good: captures the notion that 23 should have been added to the set
- Good: requires no human evaluation
- Bad: relies on underlying data structure
  - Requires the tester to think at multiple levels of abstraction
  - Test is fragile: if implementation of SetOfInteger changes, test can become useless
  - Might not even compile - those data members are probably private
Better Idea: Test the Public Functions Against Each Other

On the other hand, if you are testing the `add` and the `contains` function, you could use the second function to check the results of the first:

```java
void testAdd (SetOfInteger aSet) {
    aSet.add (23);
    assert (aSet.contains (23));
}
```

- Simpler
- Robust: tests remain valid even if data structure changes
- Legal: Does not require access to private data
void testAdd (SetOfInteger startingSet) {
    SetOfInteger aSet = startingSet;
    aSet.add (23);
    assert (aSet.contains(23));
    if (startingSet.contains(23))
        assert (aSet.size() == startingSet.size());
    else
        assert (aSet.size() == startingSet.size() + 1);
}
void testAdd ( SetOfInteger aSet)
{
    for (int i = 0; i < 1000; ++i)
    {
        int x = rand() % 500;
        bool alreadyContained = aSet.contains(x);
        int oldSize = aSet.size();
        aSet.add (23);
        assert (aSet.contains(x));
        if (alreadyContained)
            assert (aSet.size() == oldSize);
        else
            assert (aSet.size() == oldSize + 1);
    }
}
assert() might not be quite what we want

Our use of assert() in these examples has mixed results
- Good: stays quiet as long as we are passing tests
  - failures easily detected by humans
- Bad: testing always stops at the first failure
  - In a large suite of many such test cases, we may be missing out on info that would be useful for debugging
- Bad: diagnostics are limited to file name and line number where the assertion failed.
Outline I

1 Automating the Oracle
   - How can oracles be automated?
   - Examining Output
   - Limitations of Capture-and-Examine Tests

2 Self-Checking Unit & Integration Tests

3 JUnit Testing
   - JUnit Basics

4 C++ Unit Testing
   - Google Test
   - Boost Test Framework
JUnit

**JUnit** is a testing framework that has seen wide adoption in the Java community and spawned numerous imitations for other programming languages.

- Introduces a structure for a
JUnit

JUnit is a testing framework that has seen wide adoption in the Java community and spawned numerous imitations for other programming languages.

- Introduces a structure for a
  - suite (separately executable program) of
JUnit is a testing framework that has seen wide adoption in the Java community and spawned numerous imitations for other programming languages.

- Introduces a structure for a
  - suite (separately executable program) of
  - test cases (functions),
JUnit is a testing framework that has seen wide adoption in the Java community and spawned numerous imitations for other programming languages.

- Introduces a structure for a
  - suite (separately executable program)
  - test cases (functions),
  - each performing multiple self-checking tests (assertions)
JUnit is a testing framework that has seen wide adoption in the Java community and spawned numerous imitations for other programming languages.

- Introduces a structure for a
  - suite (separately executable program) of
  - test cases (functions),
  - each performing multiple self-checking tests (assertions)
    - using a richer set of assertion operators
JUnit

JUnit is a testing framework that has seen wide adoption in the Java community and spawned numerous imitations for other programming languages.

- Introduces a structure for a
  - suite (separately executable program) of
  - test cases (functions),
  - each performing multiple self-checking tests (assertions)
    - using a richer set of assertion operators
- also provides support for setup, cleanup, report generation
JUnit is a testing framework that has seen wide adoption in the Java community and spawned numerous imitations for other programming languages.

- Introduces a structure for a
  - suite (separately executable program) of
  - test cases (functions),
  - each performing multiple self-checking tests (assertions)
    - using a richer set of assertion operators
- also provides support for setup, cleanup, report generation
- readily integrated into IDEs
Let's suppose we are building a mailing list application. The mailing list is a collection of contacts.

**Contact.java** (source)

**MailingList.java** (source)

- Set up an Eclipse project with these two files
  - Make sure that these compile successfully with no errors.
- In the Eclipse Build Path, add a library: JUnit,
  - Choose JUnit4 if given a choice of versions 3 or 4
A First Test Suite

Right-click on the project and select New...JUnit Test Case.

- Give it a name (e.g., TestMailingList)
  - in the same mailinglist package as the two classes
- For “Class under test”, use the Browse button and select our MailingList class
- Click Next, then select the MailingList() and addContact(...) functions
You’ll get something like this:

```java
package mailinglist;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class TestMailingList {
    @Test
    public void testMailingList () {
        fail("Not yet implemented");
    }

    @Test
    public void testAddContact () {
        fail("Not yet implemented");
    }
}
```

Save this, and run it (right-click on the new file in the Package Explorer, and select Run As...JUnit Test

- You’ll see what failed tests look like.
- Try double-clicking on the failed cases and observe the effects on the source code listing and the Failure Trace
A First Test

Let's implement a test for the mailing list constructor.

```java
/**
 * Create an empty mailing list
 */

public MailingList() {

Glancing through the MailingList interface, what would we expect of an empty mailing list?

- the size() would be zero
- any contact we might search for would not be contained in the list
```
Testing the Constructor I

Change the `testMailingList` function to

```java
@Test
public void testMailingList() {
    MailingList ml = new MailingList();  // ①
    assertFalse (ml.contains("Jones"));  // ②
    assertNull (ml.getContact("Smith"));
    assertEquals (0, ml.size());         // ③
}
```

- ① Before we can test any function, we have to invoke it
- ② Notice the tests using different variations on the idea of an assertion
- assert(...) itself remains a language primitive - we don’t use that but use these JUnit variations instead
Testing the Constructor II

- Use `assertTrue` instead of `assert`
- The JUnit versions don’t shut down all testing the way the primitive `assert(...)` would do
  - The first failed JUnit assertion shuts down the test case (function)
- The order of parameters in `assertEquals` is significant
  - expected value comes first
Running the Constructor Test

1. Run the test suite as before
   - It should succeed (testAddContact will still fail)

2. Change the “0” in the final test to “1”. Run again, and observe the message in the Failure Trace.

3. Restore the “0”. Change the first line to
   ```java
   MailingList ml = null; // new MailingList();
   ```
   and run again.
   - Notice that a “test error” is marked differently than a “failed test”

4. Restore the first line
Testing addContact

Now let’s add a test for addContact:
After adding a contact

- the size() should increase
- the contact name should be contained in the list
- we should be able to find the contact that we just added.
Test your test (run it!)

```java
@Test
public void testAddContact() {
    MailingList ml = new MailingList();
    Contact jones = new Contact("Jones", "21 Penn. Ave.");
    ml.addContact(jones);
    assertTrue(ml.contains("Jones"));
    assertFalse(ml.contains("Smith"));
    assertEquals("Jones", ml.getContact("Jones").getName());
    assertNull(ml.getContact("Smith"));
    assertEquals(1, ml.size());
}
```

It works. But it feels awfully limited.
A Diversion - Test Setup I

Before we try making that test more rigorous, let’s look at how we can organize the set up of auxiliary data like our contacts:

```java
Contact jones;

@Before
public void setUp() throws Exception {
    jones = new Contact("Jones", "21 Penn. Ave.");
}

@Test
public void testAddContact() {
    MailingList ml = new MailingList();
    ml.addContact(jones);
}
```
Fixtures

- The class that contains the data shared among the tests in a suite is called a *fixture*. 
Fixtures

- The class that contains the data shared among the tests in a suite is called a *fixture*.
- A function marked as “@Before” will be run before each of the test case functions.
Fixtures

- The class that contains the data shared among the tests in a suite is called a **fixture**.
- A function marked as “@Before” will be run before each of the test case functions.
  - Used to (re)initialize data used in multiple tests
The class that contains the data shared among the tests in a suite is called a *fixture*.

A function marked as "@Before" will be run before each of the test case functions.
- Used to (re)initialize data used in multiple tests
- Why *every* time?
Fixtures

- The class that contains the data shared among the tests in a suite is called a *fixture*.
- A function marked as “@Before” will be run before each of the test case functions.
  - Used to (re)initialize data used in multiple tests
- Why *every* time?
  - Helps keep test cases independent
Fixtures

- The class that contains the data shared among the tests in a suite is called a **fixture**.
- A function marked as “@Before” will be run before each of the test case functions.
  - Used to (re)initialize data used in multiple tests
- Why *every* time?
  - Helps keep test cases independent
  - During debugging, it’s common to select and run single test cases in isolation
Fixtures

- The class that contains the data shared among the tests in a suite is called a **fixture**.
- A function marked as “@Before” will be run before each of the test case functions.
  - Used to (re)initialize data used in multiple tests
- **Why every time?**
  - Helps keep test cases independent
  - During debugging, it’s common to select and run single test cases in isolation
- Can do cleanup in a similar fashion with “@After”
Automating the Testing Oracle
JUnit Testing
JUnit Basics

Testing addContact - setup

Contact jones;
Contact baker;
Contact holmes;
Contact wolfe;
MailingList mlist;

@Before
public void setUp() throws Exception {
    jones = new Contact("Jones", "21 Penn. Ave.");
    baker = new Contact("Baker", "Drury Ln.");
    holmes = new Contact("Holmes", "221B Baker St.");
    wolfe = new Contact("Wolfe", "454 W. 35th St.");
    mlist = MailingList();
    mlist.addContact(baker);
    mlist.addContact(holmes);
@Test
public void testAddContact() {
    assertTrue(mlist.contains("Jones"));
    assertEquals("Jones", mlist.getContact("Jones").getName());
    assertEquals(4, ml.size());
}

Still seems a bit limited - we'll address that later.
Outline I

1. Automating the Oracle
   - How can oracles be automated?
   - Examining Output
   - Limitations of Capture-and-Examine Tests

2. Self-Checking Unit & Integration Tests

3. JUnit Testing
   - JUnit Basics

4. C++ Unit Testing
   - Google Test
   - Boost Test Framework
Google Test

Google Test, a.k.a. gtest, provides a similar *Unit environment for C++

- Download & follow instructions to prepare library
  - gtest will be added to your project as source code
  - easiest is to copy the files from fused-src/gtest/ files into a subdirectory gtest within your project.

- For Eclipse support, add the Eclipse CDT C/C++ Tests Runner plugin
Example

This time, the C++ version of the mailing list.

1. Source code is here.
   - Unpack into a convenient directory.

2. With Eclipse create a C++ project in that directory.

3. Compile

4. Run the resulting executable as a local C++ application

5. Run as a *unit test:
   - Right-click on the binary executable, select “Run as...Run configurations”.
   - Select C/C++ Unit, click “New” button
   - On the “C/C++ Testing” tab, select Tests Runner: Google Tests Runner
   - Click “Run”
Examining the ADT

`mailinglist.h` Should look familiar after the Java version
Examining the Tests I

MailingListTests.cpp

- Roughly similar to the JUnit tests

1. The class provides a fixture where data can be shared among test cases
   - This is optional, but common

2. Test cases are introduced with TEST_F
   - First argument identifies the suite (same as fixture name)
   - Second argument names the test case
   - The combination must be unique
   - Use TEST if you don’t provide a fixture class

3. The test cases feature assertions similar to those seen in JUnit
   - EXPECT assertions allow testing to continue after failure
   - ASSERT variations also exist that shut down testing on failure
Examining the Tests II

4 Public members of the fixture class will be visible to all test cases
5 and are assigned the values during SetUp, run before each test case
6 You can see the fixture members used here
Boost is a well-respected collection of libraries for C++.  

- Many of the new library components of C++11 were distributed in Boost for “beta test”.
- Other, more specialized libraries will remain separately distributed.
  - This include another popular *Unit framework for C++, the Boost Unit Test Framework (UTF).
- Basic principles are similar to Google Test
  - Also has some support for building trees of test suites.
The UTF can be added as a single header `#include` or, for greater efficiency when dealing with multiple suites, compiled to form a static or dynamic library.

Easiest to start with the single header approach. Download and unpack the Boost library.
- Add the Boost include directory to your C++ project’s search path for system headers (`#include <...>`)
  - In Eclipse, this is Project...Properties...C/C++ General...Includes.
  - Be sure to add to C++ language, all configurations

For Eclipse support, use the same [Eclipse CDT C/C++ Tests Runner plugin](https://eclipse.org/cdt/)
Example

This time, the C++ version of the mailing list.

1. Source code is here.
   - Unpack into a convenient directory.

2. With Eclipse create a C++ project in that directory.
   - Add path to Boost include directory

3. Compile

4. Run the resulting executable as a local C++ application

5. Run as a *unit test:
   - Right-click on the binary executable, select “Run as...Run configurations”.
   - Select C/C++ Unit, click “New” button
   - On the “C/C++ Testing” tab, select Tests Runner: Boost Tests Runner
   - Click “Run”
Examining the Tests I

MailingListTests2.cpp

1. This names the suite.

2. The class provides a fixture where data can be shared among test cases
   - Optional, but common
   - Simpler in Boost than in Google
     - Can be any class.
     - Initialization is done in the class constructor instead of in a special function.

3. Test cases that don’t need anything from a fixture are introduced with BOOST_AUTO_TEST_CASE
   - Argument names the test case

4. The assertions have different names but are similar in function and variety to JUnit and GTest
   - A full list is here.
Examining the Tests II

Test cases that need a fixture are introduced with

`BOOST_FIXTURE_TEST_CASE`

- Second argument identifies the fixture class
- Public members of the fixture class will be visible to