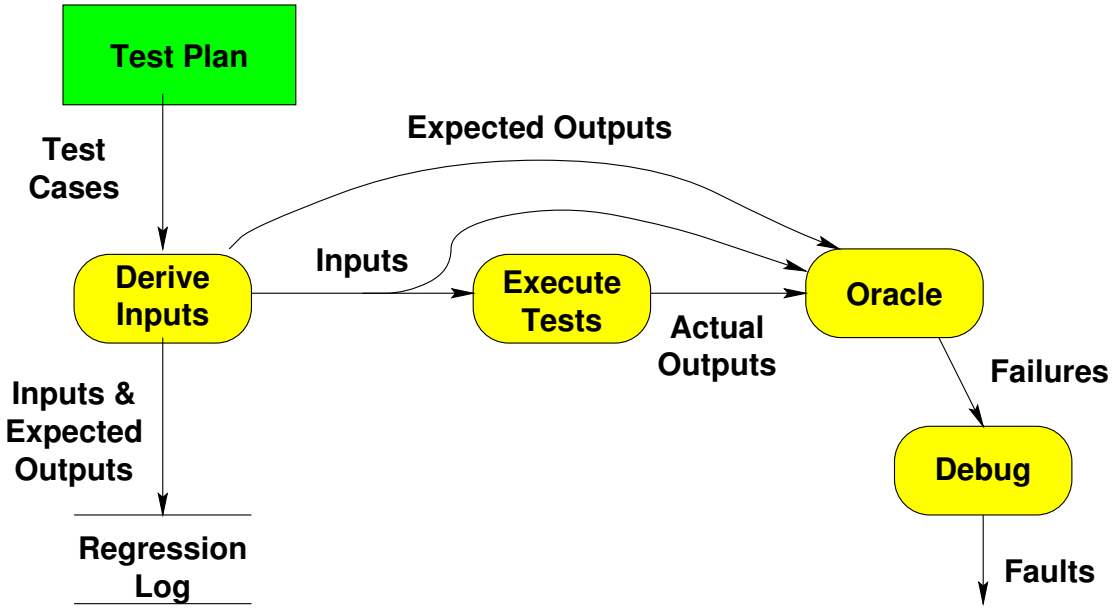
Automating the Testing Oracle

Steven J Zeil

February 13, 2013

Contents

The Testing Process



Earlier we introduced this model of the basic testing process.

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

. . . . . .

1 Automating the Oracle

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.

    Thousands, tens of thousands, even hundreds of thousands of tests are not unheard of. If your idea of testing is running some code and visually inspecting the output, you can see that won't work on this kind of scale. Regression tests have almost always been designed to check themselves. Often this was done by recording, during unit, integration, or systems test, the outputs produced by each test input. During regression testing, the same inputs are rerun, and the outputs compared to the earlier ones that had been recorded. If the outputs change, an alert is printed. If the outputs stay the same, testing moves quietly on to the next test.
- Modern development methods emphasize rapid, repeated unit tests

  It might not be obvious that self-checking is quite valuable for unit and integration testing as well. But if test outputs have to be inspected by a human, then anything more than a very few tests becomes a tedious thing to do. This motivates programmers to write very few tests and to run them infrequently, both of which are very bad ideas.
  - *Test-driven development*: Develop the tests *first*, then write the code.

    In fact, many of the latest trends in software development place a lot more emphasis on almost continual unit testing. In so-called agile or extreme programming, programmers are expected to write tests *before* implementing their functions or ADTs, and to continually rerun the tests as units are added or changed. (In effect, we get a kind of "rolling" integration test.)
  - Debugging: if you can't reproduce the error, how will you know when you've fixed it?

If a bug is reported or a new feature requested, the first step is to add new tests that fail because of the bug or missing feature. Then we will know we have successfully fixed the problem when we are able to pass that test.

But that's just not going to happen if testing is hard to do.

The best way to be sure programmers rerun the tests on a regular basis is to make the test run part of the regular build process (e.g., build the test runs into the project `make` file) and to make them self-checking.

. . . . . .

1.1 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 a program produces output files, one can self-check by creating a file representing the expected correct output, then running the program to get the actual output file and using a simple comparison utility like the Unix `diff` or `cmp` commands to see if the actual output is identical to the expected output.

For system-level regression tests, this is even simpler. Once we have a program that passes our system tests, we run it on those tests and save the outputs. Those become the expected output files for later regression testing. (Remember, the point of regression testing is to determine if any behavior has changed due to recent updates to the code.)

If the program updates a database, it may be possible to capture entire databases in a similar fashion. Alternatively, we write database queries to check for changes in the records most likely to have been affected by a test.

On the other hand, if the program's main function is to present information on a screen, self-checking is very difficult. Screen captures are often not much use, because we are unlikely to want to deal with changes where, say one window is a pixel wider or a pixel to the left of where it had been in a prior test. Self-checking tests for programs like this either require extremely fine control over all possible interactive inputs and graphics device characteristics, or

they require a careful "design for testability" to record, in a testing log file, information about what is being rendered on the screen.[1]

. . . . . .

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, it ro design these tests to be self-checking.

. . . . . .

1.2 Examining Output

1.2.1 File Tools

File tools

- **diff**, **cmp** and similar programs compare two text files byte by bute
  - 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

[1] We'll revisit this idea later in the semester when we discuss the MVC pattern for designing user interfaces.

    - 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

.....

1.2.2 expect

Expect

expect is a shell for testing interactive programs.

- an extension of **TCL** (a portable shell script).

.....

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.

- Patterns can be regular expressions or simpler “glob” patterns

interact Allow person running expect to interact with spawned program. Takes a similar list of patterns and actions.

.....

Sample Expect Script

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
```

.....

Expect: Testing a program

```
    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\ nl"} \
    timeout {fail "testsets"; puts "timeout\ n"}
}
catch {
    close
    wait
}
```

.....

1.3 Limitations of Capture-and-Examine Tests

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
- significant setup and cleanup effort per test
  - need separate test stores

. . . . . .

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

. . . . . .

2 Self-Checking Unit & Integration Tests

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

. . . . . .

In testing an ADT, we are not testing an individual function, but a collection of related functions. In some ways that makes thing easier, because we can use many of these functions to help test one another.

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.,

```
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?

```
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:

```
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

.....

Not only is this code simpler than writing your own search function as part of the test driver, it continues to work even if the data structure used to implement the ADT should be changed. What's more, it is, in a sense, a more thorough test, since it tests two functions at once. Finally, there's the simple fact that the test with the explicit loop probably won't even compile, since it refers directly to data members that are almost certainly private.

In a sense, we have made a transition from white-box towards black-box testing. The new test case deliberately ignores the underlying structure.

More Thorough Testing

```
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);
}
```

.....

More Tests

```
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.

. . . . . .

3 JUnit Testing

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
- readily integrated into IDEs

. . . . . .

3.1 JUnit Basics

Getting Started: Eclipse & JUnit

Let's suppose we are building a mailing list application. the mailing list is a collection of contacts.

```
package mailinglist;
/**
 * A contact is a name and address.
 * <p>
 * For the purpose of this example, I have simplified matters
 * a bit by making both of these components simple strings.
 * In practice, we would expect Address, at least, to be a
 * more structured type.
 *
 * @author zeil
 *
 */
public class Contact
  implements Cloneable, Comparable<Contact> {

 private String theName;
 private String theAddress;

 /**
  * Create a contact with empty name and address.
  *
  */
 public Contact ()
```


```
{
 theName = "";
 theAddress = "";
}

/**
 * Create a contact
 * @param nm  name
 * @param addr address
 */
public Contact (String nm, String addr)
{
 theName = nm;
 theAddress = addr;
}


/**
 * Get the name of the contact
 * @return the name
 */
public String getName()
{
 return theName;
}

/**
```


```
 * Change the name of the contact
 * @param nm new name
 */
public void setName (String nm)
{
 theName= nm;
}

/**
 * Get the address of the contact
 * @return the address
 */
public String getAddress()
{
 return theAddress;
}

/**
 * Change the address of the contact
 * @param addr new address
 */
public void setAddress (String addr)
{
 theAddress = addr;
}

/**
```


```
 * True if the names and addresses are equal
 */
public boolean equals (Object right)
{
 Contact r = (Contact)right;
 return theName.equals(r.theName)
    && theAddress.equals(r.theAddress);
}

public int hashCode ()
{
 return theName.hashCode() + 3 * theAddress.hashCode();
}

public String toString()
{
 return theName + ": " + theAddress;
}

public Object clone()
{
 return new Contact(theName, theAddress);
}

/**
 * Compare this contact to another.
 * Return value > 0 if this contact precedes the other,
```


```
  * == 0 if the two are equal, and < 0 if this contact
  * follows the other.
  */
 public int compareTo (Contact c)
 {
  int nmcomp = theName.compareTo(c.theName);
  if (nmcomp != 0)
   return nmcomp;
  else
   return theAddress.compareTo(c.theAddress);
 }
}
```

(source)

```
package mailinglist;

/**
 * A collection of names and addresses
 */
public class MailingList implements Cloneable {

    /**
     * Create an empty mailing list
     *
     */
    public MailingList() {
     first = null;
```


```
    last = null;
    theSize = 0;
   }

   /**
    *  Add a new contact to the list
    *  @param contact new contact to add
    */
v  public void addContact(Contact contact) {
    if (first == null) {
    // add to empty list
    first = last = new ML_Node(contact, null);
    theSize = 1;
    } else if (contact.compareTo(last.contact) > 0) {
    // add to end of non-empty list
    last.next = new ML_Node(contact, null);
    last = last.next;
    ++theSize;
    } else if (contact.compareTo(first.contact) < 0) {
    // add to front of non-empty list
    first = new ML_Node(contact, first);
    ++theSize;
    } else {
    // search for place to insert
    ML_Node previous = first;
    ML_Node current = first.next;
    assert (current != null);
```


```
    while (contact.compareTo(current.contact) < 0) {
    previous = current;
    current = current.next;
    assert (current != null);
    }
    previous.next = new ML_Node(contact, current);
    ++theSize;
    }
  }

  /**
   *  Remove one matching contact
   *  @param c remove a contact equal to c
   */
  public void removeContact(Contact c) {
   ML_Node previous = null;
   ML_Node current = first;
   while (current != null && c.getName().compareTo(current.contact.getName()) > 0) {
   previous = current;
   current = current.next;
   }
   if (current != null && c.getName().equals(current.contact.getName()))
   remove(previous, current);
  }

  /**
   * Remove a contact with the indicated name
```


```
     * @param name name of contact to remove
     */
    public void removeContact(String name) {
     ML_Node previous = null;
     ML_Node current = first;
     while (current != null && name.compareTo(current.contact.getName()) > 0) {
     previous = current;
     current = current.next;
     }
     if (current != null && name == current.contact.getName())
     remove(previous, current);

    }

    /**
     * Search for contacts
     * @param name name to search for
     * @return true if a contact with an equal name exists
     */
    public boolean contains(String name) {
     ML_Node current = first;
     while (current != null && name.compareTo(current.contact.getName()) > 0) {
     current = current.next;
     }
     return (current != null && name == current.contact.getName());
    }
```


```
/**
 * Search for contacts
 * @param name name to search for
 * @return contact with that name if found, null if not found
 */
public Contact getContact(String name) {
 ML_Node current = first;
 while (current != null && name.compareTo(current.contact.getName()) > 0) {
 current = current.next;
 }
 if (current != null && name == current.contact.getName())
 return current.contact;
 else
 return null;
}

/**
 * combine two mailing lists
 *
 */
public void merge(MailingList anotherList) {
 // For a quick merge, we will loop around, checking the
 // first item in each list, and always copying the smaller
 // of the two items into result
 MailingList result = new MailingList();
 ML_Node thisList = first;
 ML_Node otherList = anotherList.first;
```


```
while (thisList != null && otherList != null) {
int comp = thisList.contact.compareTo(otherList.contact);
if (comp <= 0) {
result.addContact(thisList.contact);
thisList = thisList.next;
/* if (comp == 0)
otherList = otherList.next;
                          [Deliberate bug ]  */
} else {
result.addContact(otherList.contact);
otherList = otherList.next;
}
}
// Now, one of the two lists has been entirely copied.
// The other might still have stuff to copy. So we just copy
// any remaining items from the two lists. Note that one of these
// two loops will execute zero times.
while (thisList != null) {
result.addContact(thisList.contact);
thisList = thisList.next;
}
while (otherList != null) {
result.addContact(otherList.contact);
otherList = otherList.next;
}
// Now result contains the merged list. Transfer that into this list.
first = result.first;
```


```
    last = result.last;
    theSize = result.theSize;
  }

  /**
   * How many contacts in list?
   */
  public int size() {
    return theSize;
  }

  /**
   * Return true if mailing lists contain equal contacts
   */
  public boolean equals(Object anotherList) {
    MailingList right = (MailingList) anotherList;
    if (theSize != right.theSize) // (easy test first!)
    return false;
    else {
    ML_Node thisList = first;
    ML_Node otherList = right.first;
    while (thisList != null) {
    if (!thisList.contact.equals(otherList.contact))
    return false;
    thisList = thisList.next;
    otherList = otherList.next;
    }
```


```
 return true;
 }
}

public int hashCode() {
 int hash = 0;
 ML_Node current = first;
 while (current != null) {
 hash = 3 * hash + current.contact.hashCode();
 current = current.next;
 }
 return hash;
}

public String toString() {
 StringBuffer buf = new StringBuffer("{");
 ML_Node current = first;
 while (current != null) {
 buf.append(current.contact.toString());
 current = current.next;
 if (current != null)
 buf.append("n");
 }
 buf.append("}");
 return buf.toString();
}
```


```
/**
 * Deep copy of contacts
 */
public Object clone() {
 MailingList result = new MailingList();
 ML_Node current = first;
 while (current != null) {
 result.addContact((Contact) current.contact.clone());
 current = current.next;
 }
 return result;
}

private class ML_Node {
 public Contact contact;

 public ML_Node next;

 public ML_Node(Contact c, ML_Node nxt) {
 contact = c;
 next = nxt;
 }
}

private int theSize;

private ML_Node first;
```


```
    private ML_Node last;

    // helper functions
    private void remove(ML_Node previous, ML_Node current) {
     if (previous == null) {
     // remove front of list
     first = current.next;
     if (last == current)
     last = null;
     } else if (current == last) {
     // remove end of list
     last = previous;
     last.next = null;
     } else {
     // remove interior node
     previous.next = current.next;
     }
     --theSize;
    }

}
```

(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

.....

A First Test Suite (cont.)

You'll get something like this:

```
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.

```
/**
 * 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

Change the `testMailingList` function to

```
@Test
public void testMailingList() {
    MailingList ml = new MailingList();  ❶
    assertFalse (ml.contains("Jones"));  ❷
    assertNull (ml.getContact("Smith"));
    assertEquals (0, ml.size());         ❸
}
```

- ❶ Before we can test any function, we have to invoke it
- ❷ 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
    - 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)
- ❸ 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

   ```
   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.

. . . . . .

Testing addContact - first pass

Try this test (run it!)

```
@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

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:

```
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*.
- 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"

. . . . . .

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 (holmed);
    mlist.addContact (baker);
}
```

Create a series of contacts

. . . . . .

Testing addContact - improved

```
@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.

. . . . . .

4 C++ Unit Testing

4.1 Google Test

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

```
#ifndef MAILINGLIST_H
#define MAILINGLIST_H

#include <iostream>
#include <string>

#include "contact.h"

/**
   A collection of names and addresses
*/
class MailingList
{
public:
  MailingList();
  MailingList(const MailingList&);
  ~MailingList();
```



```
  const MailingList& operator= (const MailingList&);

  // Add a new contact to the list
  void addContact (const Contact& contact);

  // Does the list contain this person?
  bool contains (const Name&) const;

  // Find the contact
  const Contact& getContact (const Name& nm) const;
  //pre: contains(nm)

  // Remove one matching contact
  void removeContact (const Contact&);
  void removeContact (const Name&);

  // combine two mailing lists
  void merge (const MailingList& otherList);

  // How many contacts in list?
  int size() const;


  bool operator== (const MailingList& right) const;
  bool operator< (const MailingList& right) const;

private:
```


```
  struct ML_Node {
    Contact contact;
    ML_Node* next;

    ML_Node (const Contact& c, ML_Node* nxt)
      : contact(c), next(nxt)
    {}
  };

  ML_Node* first;
  ML_Node* last;
  int theSize;

  // helper functions
  void clear();
  void remove (ML_Node* previous, ML_Node* current);

  friend std::ostream& operator<< (std::ostream& out, const MailingList& addr);
};

// print list, sorted by Contact
std::ostream& operator<< (std::ostream& out, const MailingList& list);


#endif
```


Should look familiar after the Java version

Examining the Tests

```
#include "mailinglist.h"

#include <string>
#include <vector>

#include "gtest/gtest.h"

namespace {

using namespace std;

// The fixture for testing class MailingList.
class MailingListTests : public ::testing::Test {  ❶
 public:
  Contact jones;                                    ❹
  MailingList mlist;

  virtual void SetUp() {
    jones = Contact("Jones", "21 Penn. Ave.");      ❺
    mlist = MailingList();
    mlist.addContact (Contact ("Baker", "Drury Ln."));
    mlist.addContact (Contact ("Holmes", "221B Baker St."));
```


```
    mlist.addContact (Contact ("Wolfe", "454 W. 35th St."));
  }

  virtual void TearDown() {
  }

};

TEST_F (MailingListTests, constructor) {
  MailingList ml;
  EXPECT_EQ (0, ml.size());                ❷
  EXPECT_FALSE (ml.contains("Jones"));
}


TEST_F (MailingListTests, addContact) {
  mlist.addContact (jones);
  EXPECT_TRUE (mlist.contains("Jones"));          ❸
  EXPECT_EQ ("Jones", mlist.getContact("Jones").getName());
  EXPECT_EQ (4, ml.size());
}




}  // namespace
```


- Roughly similar to the JUnit tests

❶ The class provides a fixture where data can be shared among test cases

  – This is optional, but common

❷ 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

❸ 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

❹ Public members of the fixture class will be visible to all test cases

  ❺ and are assigned the values during `SetUp`, run before each test case

  ❻ You can see the fixture members used here

4.2 Boost Test Framework

Boost UTF

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.

. . . . . .

Boost Test

- 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 wit hthe 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

. . . . . .

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

```
#define BOOST_TEST_MODULE MailingList test

#include "mailinglist.h"
```


```
#include <string>
#include <vector>

#include <boost/test/included/unit_test.hpp>

using namespace std;

// The fixture for testing mailing lists.
class MailingListTests  {                          ❶
 public:
  Contact jones;                                   ❹
  MailingList mlist;

  MailingListTests() {                             ❺
    jones = Contact("Jones", "21 Penn. Ave.");
    mlist.addContact (Contact ("Baker", "Drury Ln."));
    mlist.addContact (Contact ("Holmes", "221B Baker St."));
    mlist.addContact (Contact ("Wolfe", "454 W. 35th St."));
  }

  ~MailingListTests() {
  }

};

BOOST_AUTO_TEST_CASE ( constructor ) {
```


```
  MailingList ml;
  BOOST_CHECK_EQUAL (0, ml.size());                ❷
  BOOST_CHECK (!ml.contains("Jones"));
}


BOOST_FIXTURE_TEST_CASE (addContact, MailingListTests) {
  ml.addContact (jones);
  BOOST_CHECK (ml.contains("Jones"));              ❸
  BOOST_CHECK_EQUAL ("Jones", ml.getContact("Jones").getName());
  BOOST_CHECK_EQUAL (4, ml.size());
}
```

❶ This names the suite.

❷ 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.

❸ Test cases that don't need anything from a fixture are introduced with `BOOST_AUTO_TEST_CASE`

- Argument names the test case

❹ The assertions have different names but are similar in function and variety to JUnit and GTest

- A full list is here.

❺ 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

. . . . . .