Announcement: A0 is coming up!

- A0 will be available after class
- Go to Canvas
- Check A0
  - If you will use git to synch your lab/assignment solutions with Vocareum, fill out the GitHub Credentials survey on Canvas
  - We will invite you to the course GitHub org once you do that

JUNIT: MORE ELEGANT ASSERTIONS

Hamcrest matchers make tests more natural

Syntactic sugar and **extensible** pattern matching methods that allow more *literal* assertions/oracles with better diagnostic output

```
assertThat(actual, match-expression)
```

a built-in or custom matcher
instead of an expected value

Without matcher

```
@Test
public void checkMapHasRightKeyAndValue() {
   HashMap myMap = populateMap();
   assertTrue(myMap.containsKey(“bar”));
   assertEquals(“foo”, myMap.get(“bar”));
}
```

With matcher

```
@Test
public void checkMapHasAssociationIAmLookingFor() {
  HashMap myMap = populateMap();
  assertThat(myMap, hasEntry(equalTo(“bar”),
          equalTo("foo")));
}
```

Which one is more readable?

Sample Hamcrest matchers - Quick Reference

Core/Generic
anything - always matches, useful if you don't care what the object is
describedAs - decorator for adding custom failure description/message
is — syntactic sugar, just a wrapper

Logical
allOf - matches if all matchers match, short circuits (like Java &&)
anyOf - matches if any matchers match, short circuits (like Java ||)
not - matches if the wrapped matcher doesn't match and does not match otherwise

Object
equalTo - test object equality using Object.equals
hasToString - test Object.toString
instanceOf, isCompatibleType - test type
notNullValue, nullValue - test for null
sameInstance - test object identity

Collections (really useful)
array - test an array's elements against an array of matchers
hasEntry, hasKey, hasValue - test a map contains an entry, key or value
hasItem, hasItems - test a collection contains elements
hasItemInArray - test an array contains an element

Number
closeTo - test floating point values are close to a given value
greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo — c

Text/String (really useful)
equalToIgnoringCase - test string equality ignoring case
equalToIgnoringWhiteSpace - test string equality ignoring differences in runs of whitespace
containsString, endsWith, startsWith - test string matching

https://code.google.com/p/hamcrest/wiki/Tutorial

Hamcrest examples

```
assertThat(true, hasToString(equalTo(“TRUE”)))

assertThat(new Integer[]{1,2,3},
    is(array(equalTo(1), equalTo(2), equalTo(3))))

assertThat(new String[] {"foo", "bar"},
    hasItemInArray(startsWith("ba")))

assertThat(myMap,
    hasEntry(equalTo("bar"), equalTo("foo")));

assertThat(result,
    describedAs("Not same object”, sameInstance(expectedResult)));
```

Testing Principles & Best Practices

applied in Unit Testing

Unit Testing Best Practices

- Simplicity
- Understandability
- Essentiality
- Single purpose
- Behavior first
- Maintainability
- Determinism
- Independence
- Failability
- Comprehensiveness
- Speed

Simplicity

Keep it simple, keeping it small and neat

Take small steps
Avoid complex fixture code

Image: http://www.delcoalarm.com/3-simple-tips-to-keep-your-home-safe/

Complicated fixture: setup sermon

ARRANGE: Executed *before* each test

Executed *after* each test

Listing 4.21 Setup sermon is a long setup for a simple test

```
public class PackageFetcherTest {
   private PackageFetcher fetcher;
   private Map downloads;
   private File tempDir;

   @Before
   public void setUp() throws Exception {
      String systemTempDir = System.getProperty("java.io.tmpdir");
      tempDir = new File(systemTempDir, "downloads");
      tempDir.mkdirs();
      String filename = "/manifest.xml";
      InputStream xml = getClass().getResourceAsStream(filename);
      Document manifest = XOM.parse(IO.streamAsString(xml));
      PresentationList presentations = new PresentationList();
      presentations.parse(manifest.getRootElement());
      PresentationStorage db = new PresentationStorage();
      List list = presentations.getResourcesMissingFrom(null, db);
      fetcher = new PackageFetcher();
      downloads = fetcher.extractDownloads(list);
   }
   @After
   public void tearDown() throws Exception {
      IO.delete(tempDir);
   }

   @Test
   public void downloadsAllResources() {
      fetcher.download(downloads, tempDir, new MockConnector));
      assertEquals(4, tempDir.list().length);
   }
}
```

Complex fixtures: what to do?

- Extract the nonessential details from fixture into private methods
- Give things appropriate, descriptive names
- Strive for a single level of abstraction in the fixture code

Setup sermon fixed

Listing 4.22 Extracting details makes the setup easier to grasp

Koskela 2013

```
public class PackageFetcherTest {
   private PackageFetcher fetcher;
   private Map downloads;
   private File tempDir;

   @Before
   public void setUp() throws Exception {
      fetcher = new PackageFetcher();
      tempDir = createTempDir("downloads");
      downloads = extractMissingDownloadsFrom("/manifest.xml");
   }
   @After
   public void tearDown() throws Exception {
      IO.delete(tempDir);
   }

   @Test
   public void downloadsAllResources() {
      fetcher.download(downloads, tempDir, new MockConnector());
      assertEquals(4, tempDir.list().length);
   }

   private File createTempDir(String name) {
      String systemTempDir = System.getProperty("java.io.tmpdir"
      File dir = new File(systemTempDir, name);
      dir.mkdirs();
      return dir;
   }
   ...
```

Understandability

Tests should be readable and what they are testing should be clear

Make tests self-documenting

Image: http://abc-lang.com/translation-service/document-translation/

Self-documenting tests – a lot is in the test name

```
// Name your tests well

@Test
public void testFrameOkInStrikes() {
    ..
    game().roll(new Frame(10, 0)); // a strike
    game().roll(new Frame(10, 0)); // another strike
    game().roll(new Frame(4, 2)); // duh
    assertEquals(24, game().frame(1).getScore());
    assertEquals(16, game().frame(2).getScore());
    assertEquals(6, game().frame(3).getScore());
}

@Test
public void strikeAfterStrikeResultsInTwoAffixedRolls() {
    …
}
```

Understandability

Simplicity will support understandability

To improve readability: Substitute primitive/native assertions with the more natural matchers (e.g., hamcrest library)

Image: http://abc-lang.com/translation-service/document-translation/

Understandability

No obscurity in tests! -- Avoid magic numbers

Magic Numbers

Any **hardcoded constant is potentially a magic number**

```
@Test public void perfectGame() throws Exception {
    roll(10, 12);
    assertThat(game.score(), is(equalTo(300));
```

Which ones are magic numbers?

Why are magic numbers bad?

Magic Numbers

Magic number: **hardcoded value whose meaning is obscure**

```
@Test public void perfectGame() throws Exception {
    roll(10, 12); // magic?
    assertThat(game.score(), is(equalTo(300)); // magic?
```

Vs.

```
@Test public void perfectGameScore() throws Exception {
    final int PERFECT_SCORE = 300;
    roll(pins(10), times(12));
    assertThat(game.score(), is(equalTo(PERFECT_SCORE));

    private int pins(int n) { return n; }
    private int times(int n) { return n;}
}
```

No mystery; self-documenting!

Essentiality

A test should not be overprotective

Redundancy is a V&V principle that promotes application of different testing strategies to test possibly the same behavior from different perspectives, but it's not good within test cases and within a single testing strategy!

➡ *Remove redundant assertions*

Image: http://www.legacydad.com/overprotective-parents-underdeveloped-children-part-1/

Overprotection with redundancy we don't want *or need*


```
@Test
public void testCount() {
    Data data = project.getData();
    assertNotNull(data);
    // next assert would fail if data = null
    assertEquals(4, data.count());
}
```

```
@Test
public void testSummary() {
    Data data = project.getData();
    assertNotNull(data);
    // next assert would fail if data = null, but…
    assertEquals(100, data.getSummary().getTotal());
}
```

Where is the redundancy here?

Example: Koskela 2013

Overprotection with redundancy

```
@Test
public void testCount() {
    Data data = project.getData();
    assertNotNull(data);
    // next assert would fail if data = null
    assertEquals(4, data.count());
}
```

maybe redundant here

```
@Test
public void testSummary() {
    Data data = project.getData();
    assertNotNull(data);
    // next assert would fail if data = null, but…
    assertEquals(100, data.getSummary().getTotal());
}
```

useful?

If thrown, where could the NullPointerException originate from? data or data.getSummary()?

Since data can be null ad data,getsummary can be null, the assertnotNull before eliminate one of the possibility and is thus useful

Check to see if certain assertions are subsumed by others under all conditions, and if extra conditions are worth checking!

Essentiality

A test should not be overprotective

Remove redundant assertions

Make brittle tests less brittle

Image: http://www.legacydad.com/overprotective-parents-underdeveloped-children-part-1/

Overprotection with “hyperassertions”


```
public class LogFileTransformerTest {
private String expectedOutput;
private String logFile;

@Before
public void setUpBuildLogFile() {
  StringBuilder lines = new StringBuilder();
  appendTo(lines, "[2005-05-23 21:20:33] LAUNCHED");
  appendTo(lines, "[2005-05-23 21:20:33] session-id###SID");
  appendTo(lines, "[2005-05-23 21:20:33] user-id###UID");
  ... // many more appendTo's here
  appendTo(lines, "[2005-05-23 21:22:48] STOPPED");
  logFile = lines.toString();
}

@Before
public void setUpBuildTransformedFile() {
  StringBuilder file = new StringBuilder();
  ... // many more appendTo's here
  appendTo(file, "screen3###0");
  appendTo(file, "screen4###24");
  appendTo(file, "finished###2005-05-23 21:22:48");
  expectedOutput = file.toString();
}
```

Review this piece of setup code for a minute!

Overprotection with hyperassertions


```
@Test
public void transformationGeneratesRightStuffIntoTheRightFile()
throws Exception {
     TempFile input = TempFile.withSuffix(".src.log").append(logFile);
     TempFile output = TempFile.withSuffix(".dest.log");
     new String expectedOutput = LogFileTransformer().transform(input.file(), output.file());
     assertTrue("Destination file was not created", output.exists());
     assertEquals(expectedOutput, output.content());
}
…
```

Asserting equality is brittle. Should do something else such as checking properties (no. lines, etc.). This will be resilient to any change of the format of the log file

Hyperassertion: an overzealous assertion cursed with details that may change and don't matter for the logic being tested…
to such an extent that the assertion breaks easily (thus making the test *brittle*)

Identify the offending hyperassertion in the above code!

Overprotection with hyperassertions

```
@Test
public void transformationGeneratesRightStuffIntoTheRightFile()
throws Exception {
    TempFile input = TempFile.withSuffix(".src.log").append(logFile);
    TempFile output = TempFile.withSuffix(".dest.log");
    new LogFileTransformer().transform(input.file(), output.file());
    assertTrue("Destination file was not created", output.exists());
    assertEquals(expectedOutput, output.content());
}
…
```

Offending hyperassertion in red!

How would you address it?

A matcher lib like ham crest or regular expression matcher lib would be good

Single purpose

A test should have one reason to fail

A test should have a narrow focus

Avoid testing multiple behaviors and split logic in a single test

Vs.

http://www.recycleaway.com/

A split-personality test


```
public class TestConfiguration {
    @Test
    public void testParsingCommandLineArguments() {
        String[] args = { "-f", "hello.txt", "-v", "--version" };
        Configuration c = new Configuration();
        c.processArguments(args);
        assertEquals("hello.txt", c.getFileName());
        assertFalse(c.isDebuggingEnabled());
        assertFalse(c.isWarningsEnabled());
        assertTrue(c.isVerbose());
        assertTrue(c.shouldShowVersion());
        c = new Configuration();
        try {
            c.processArguments(new String[] {"-f"});
            fail("Should've failed");
        } catch (InvalidArgumentException expected) {
        // this is okay and expected
        }
    }
}
```

2 behaviors

Is this test focused enough? Why? Why not?

Tests with no split personality

```
public class TestConfiguration {
    ...
    @Test
    public void validArgumentsProvided() {
        Configuration c = new Configuration();
        String[] args = { "-f", "hello.txt", "-v",
                "— "    version" };
        c.processArguments(args);
        assertEquals("hello.txt", c.getFileName());
        assertFalse(c.isDebuggingEnabled());
        assertFalse(c.isWarningsEnabled());
        assertTrue(c.isVerbose());
        assertTrue(c.shouldShowVersion());
    }

    @Test(expected = InvalidArgumentException.class)
    public void missingArgument() {
        Configuration c = new Configuration();
        c.processArguments(new String[] {"-f"});
    }
}
```

Check arguments are ok - “happy” path

Check bad arguments throw a descriptive exception - “sad” path

It’s better

Behavior first (*aka* black-box first)

[*often*] 100% test coverage is not the [first] goal

Focus on behavior first, not implementation

Black-box first; white-box next

Image: http://www.dianedokkokim.com/

In black-box testing, we are testing against a contract

Unit testing: testing a method within a class…

- A method's contract is a statement of the responsibilities of that method, and the responsibilities of the code that calls it
  - Analogy: legal contract
    - Spec: If you pay me $20, then I will walk your dog
    - Test: if I walked your dog, then I was paid $20
- All methods have contracts
  - Implicit or explicit
  - Formal or informal
- Can be considered specification, but based on code

Implicit method contract: int division example

- Not documented or described by the programmer
  - Still exists

```
public int divide(int x, int y) {
   return x / y;
}
```

  - What is the contract?

Informal method contract

- Described informally in comment

```
/**
 * Divides two numbers. This method assumes
 * that the numbers are greater than 0
 */
public int divide(int x, int y) {
   return x / y;
}
```

Pre/Post conditions and invariants

- More formal and explicit
  - Precondition
    - Things that must be true of parameters and object state for call to be ‘legal.’
  - Post-condition
    - Things this method guarantees will be true of object state and the return value after being called
  - Invariants
    - Something that will always be true
    - Usually describe valid state of an object

Pre/Post condition example

```
public class BankAccount {
  public static final int MAX_BALANCE = 1000;

  // Invariant: The balance will always be greater than
  // zero, but less than MAX_BALANCE.
  private int balance;

  // Precond: amount is greater than zero
  // Postcond: the new balance is set to the
  // old balance plus amount.
  public void credit(int amount) { ... }

  // Precond: amount is greater than zero
  // Postcon: balance set to the old balance minus amount
  public void debit(int amount) { ... }
```

Pre/Post condition example: what to test

```
public class BankAccount {
  public static final int MAX_BALANCE = 1000;

  // Invariant: The balance will always be greater than
  // zero, but less than MAX_BALANCE.
  private int balance;

  // Precond: amount is greater than zero
  // Postcond: the new balance is set to the
  // old balance plus amount.
  public void credit(int amount) { ... }

  // Precond: amount is greater than zero
  // Postcon: balance set to the old balance minus amount
  public void debit(int amount) { ... }
```

Balance is in the right range after each operation

If precond is satisfied, postcond must be true

If precond is not satisfied, what happens?

If precond is satisfied, postcond must be true

If precond is not satisfied, what happens?

Maintainability

Tests should be easy to maintain

Avoid duplication

All the things that we apply to production code apply to test code too to make test code maintainable!

Refactor test code

Image: http://innovativemlmsuccess.com/tag/duplication/

Avoiding duplication

Same principle as production code…

Factor duplicated code out to:

- @Before methods
- @BeforeClass methods
- Private methods in test class
- *Abstract super-test-class if necessary*

Oioooiiiiiooioioi

Example in A1…

Maintainability

Avoid conditional logic in test code

Image: http://innovativemlmsuccess.com/tag/duplication/

Conditional logic in tests

Listing 5.4 Conditional logic in test code is a burden to maintain

```
public class DictionaryTest {
   @Test
   public void returnsAnIteratorForContents() throws Exception {
      Dictionary dict = new Dictionary();
      dict.add("A", new Long(3));
      dict.add("B", "21");
      for (Iterator e = dict.iterator(); e.hasNext();) {
         Map.Entry entry = (Map.Entry) e.next();
         if ("A".equals(entry.getKey())) {
            assertEquals(3L, entry.getValue());
         }
         if ("B".equals(entry.getKey())) {
            assertEquals("21", entry.getValue());
         }
      }
   }
}
```

Method name indicates we're testing the iterator specifically…

1 Populate Dictionary

Loop through entries 2 *using an iterator*

3 Assert value for familiar keys *showing iterator works*

Conditional logic factored out to a custom assertion

Listing 5.5 Extracting a custom assertion cleans up the test

```
@Test
public void returnsAnIteratorForContents() throws Exception {
   Dictionary dict = new Dictionary();
   dict.add("A", new Long(3));
   dict.add("B", "21");
   assertContains(dict.iterator(), "A", 3L);
   assertContains(dict.iterator(), "B", "21");
}

private void assertContains(Iterator i, Object key, Object value) {
   while (i.hasNext()) {
      Map.Entry entry = (Map.Entry) i.next();
      if (key.equals(entry.getKey())) {
         assertEquals(value, entry.getValue());
         return;
      }
   }
   fail("Iterator didn't contain " + key + " => " + value);
}
```

1 Simple custom assertion

2 Found what we we're looking for

3 Fail the test

Determinism

Tests should not fail at random!

Isolate and remove sources of nondeterminism

What kind of sources of nondeterminism can you think of?

1\race condition 2\timestamp or random numbers

Image: http://psmag.com

Determinism

Tests should not fail at random!

Isolate and remove sources of nondeterminism in the test fixture: concurrency, timestamps, arbitrary delays, external resources/connections/APIs, …

*Use **test doubles in the test fixture** rather than real collaborators/dependencies to make tests deterministic*

More on ***test doubles*** later…

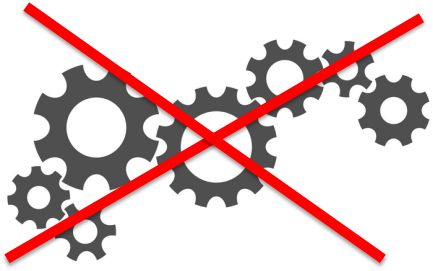
Image: http://psmag.com

Independence

Tests should be able to run in any order,
and give the same results

Tests should not depend on each other

Isolate
each test
from others:
each test should start
with a clean slate!



Be careful when they
share fixtures, and the fixtures are modified by tests!
Use @Before to start each test on a blank slate!

Failability

“Never-failing” tests tests don’t make sense

Make sure that you can make tests fail

Don’t write tests without assertions

Make sure that unfinished tests fail

Can this test fail?

Listing 6.3 Test that never fails

```
@Test
public void includeForMissingResourceFails() {
   try {
      new Environment().include("somethingthatdoesnotexist");
   } catch (IOException e) {
      assertThat(e.getMessage(),
                 contains("somethingthatdoesnotexist"));
   }
}
```

Call to “include” method either succeeds or throws an exception:

- What should happen if action fails and exception is thrown? Is it happening?
- What should happen if action succeeds? Is it happening?

Assume if an exception is thrown inside “include, it’s always the right exception!

Can this test fail?

Listing 6.3 Test that never fails

```
@Test
public void includeForMissingResourceFails() {
    try {
        new Environment().include("somethingthatdoesnotexist");
    } catch (IOException e) {
        assertThat(e.getMessage(),
                   contains("somethingthatdoesnotexist"));
    }
}
```

Call to the include method either succeeds or throws an exception:

- If the production code is correct, action fails, exception is thrown, assertion checks error msg is correct, and test passes!
- If the production code is incorrect, action succeeds, exception is not thrown, and test completes gracefully and passes! Oh no, that's not what I want!
- What's missing here?

Use fail() to… make a test fail

Listing 6.4 Adding the missing `fail()` call makes our test useful

```
@Test
public void includeForMissingResourceFails() {
    try {
        new Environment().include("somethingthatdoesnotexist");
        fail();
    } catch (IOException e) {
        assertThat(e.getMessage(),
                   contains("somethingthatdoesnotexist"));
    }
```

1 Fail test unless exception is thrown

Super important:

- you'll see testing blogs that make this mistake
- you'll find test code in famous OS libraries that makes this mistake!

May use `expected` parameter in test annotation to make a test fail when an expected exception is not thrown OR use assertThrows

Listing 6.5 Declaring an anticipated exception with the `@Test` annotation

```
@Test(expected = IOException.class)
public void includingMissingResourceFails() {
   new Environment().include("somethingthatdoesnotexist");
}
```

Be careful: this does not localize the failure if the test code is complex!

OR

```
@Test
public void thowsAnExceptionOnCallingFoo() {
    assertThrows(AnException.class, () -> { foo();  });
}
```

These features may not be available in other testing frameworks or for other programmimg languages!

Avoiding superfluous & incomplete tests

Trigger failure (intentionally fail)

- Try to make the test fail first by injecting a defect or stubbing out parts of production code
- Once test fails, make it pass by restoring the correct production code

Replace commented out sections with fail()

- To avoid forgetting to replace stubbed out ("to-finish-later") test code or comment

```
@Test databaseConnectionMustBeClosedAfterOp() {
  // todo                                          => fail()
}
```

Comprehensiveness

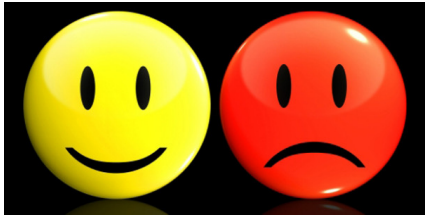
Tests should travel to diverse worlds

*Test all **potentially faulty** behaviors*

*by choosing **representative** cases*

happy and *sad* paths

normal expected behavior
and
expected alternative/exceptional behaviors



boundary cases

Cases that use special values prone to faults at the limits of a data type or resource

corner cases

Unusual, incidental, unexpected, difficult to reproduce, interacting cases that can be pathological, not just any ordinary "sad path" ("what-if" paths)

When stack is empty or one element away from empty are special values.
When stack is full or one element from full, they re special values as well.

Speed

Tests should provide fast feedback (run fast)

*Use **test doubles** if tests rely on expensive resources*

Image: http://images.businessweek.com/ss/10/04/0406_most_expensive_watches/18.htm

Speed

Tests should provide fast feedback (run fast)

*Use **test doubles** if tests rely on expensive resources*

Review and optimize your tests if they are sluggish

Watch for "sleeping snails"

Homework: review the "sleeping snail" code smell and its fix in this deck

Image: http://images.businessweek.com/ss/10/04/0406_most_expensive_watches/18.htm

Sleeping Snail

- Pervasiveness of sleep calls in multi-threaded code (in Java: calls to `Thread.sleep()`)

- Fix: *get rid of over-cautious, wasteful idle waiting!*

  … by using thread synchronization facilities

  (e.g., in Java: `java.util.concurrent` package)

Homework: review and understand this code smell

Sleeping snail: example

Homework: review this code on your own, think about why it's problematic
(it's bad in more than one way, violates multiple principles)

Answer is in the corresponding Koskela 2013 chapter – check Canvas materials if you need to….

Listing 5.10 Testing multithreaded access to a counter

```
@Test
public void concurrentAccessFromMultipleThreads() throws Exception {
   final Counter counter = new Counter();

   final int callsPerThread = 100;
   final Set<Long> values = new HashSet<Long>();
   Runnable runnable = new Runnable() {
      @Override
      public void run() {
         for (int i = 0; i < callsPerThread; i++) {
            values.add(counter.getAndIncrement());
         }
      }
   };

   int threads = 10;
   for (int i = 0; i < threads; i++) {
      new Thread(runnable).start();
   }

   Thread.sleep(500);

   int expectedNoOfValues = threads * callsPerThread;
   assertEquals(expectedNoOfValues, values.size());
}
```


Homework: review and understand this code smell

Sleeping snail: example

Listing 5.10 Testing multithreaded access to a counter

```
@Test
public void concurrentAccessFromMultipleThreads() throws Exception {
   final Counter counter = new Counter();

   final int callsPerThread = 100;
   final Set<Long> values = new HashSet<Long>();
   Runnable runnable = new Runnable() {
      @Override
      public void run() {
   };
   int
   for (int i = 0; i < threads; i++) {
      new Thread(runnable).start();
   }

   Thread.sleep(500);

   int expectedNoOfValues = threads * callsPerThread;
   assertEquals(expectedNoOfValues, values.size());
}
```

1 Threads increment counter in a loop

2 Start threads

3 Wait for threads to finish

4 Check values' uniqueness

- Is 500 msec long enough for all threads to finish?
- Is it guaranteed that they will finish?
- Could there be any race conditions?
- Will this test always pass?
- Is it deterministic?

Sleeping snail: fix (using the right concurrency construct)

Listing 5.11 Testing multithreaded access without sleeping

```
@Test
public void concurrentAccessFromMultipleThreads() throws Exception {
   final Counter counter = new Counter();

   final int numberOfThreads = 10;
   final CountDownLatch allThreadsComplete =
         new CountDownLatch(numberOfThreads);

   final int callsPerThread = 100;
   final Set<Long> values = new HashSet<Long>();
   Runnable runnable = new Runnable() {
      @Override
      public void run() {
         for (int i = 0; i < callsPerThread; i++) {
            values.add(counter.getAndIncrement());
         }
         allThreadsComplete.countDown();
      }
   };

   for (int i = 0; i < numberOfThreads; i++) {
      new Thread(runnable).start();
   }

   allThreadsComplete.await(10, TimeUnit.SECONDS);

   int expectedNoOfValues = numberOfThreads * callsPerThread;
   assertEquals(expectedNoOfValues, values.size());
}
```

1. **Synchronization latch**
   Main thread will be notified when this latch expires
2. **Mark thread completed**
3. **Wait for threads to complete**
   Or timeout in 10 secs

Key Messages

- Unit testing is applied throughout construction.
- Good unit tests involve *simple*, *understandable*, *meaningful*, *deterministic*, *independent* tests that have a *single purpose*, can demonstrably *fail*, provide *fast feedback*, are *easy to maintain*, focus on *behavior first*, and are *comprehensive* without being *overprotective*.