Miscellaneous

1) Testing against Non-Determinism
2) Mutation Testing
3) Test Patterns and Smells

Testing against Non-Determinism

https://blogs.msdn.microsoft.com/ploeh/2007/05/11/testing-against-non-determinism/

Outline

1) Testing Against Randomness

2) Testing Against GUIDs

3) Testing Against The Current Time

4) Testing Against The Passage of Time

Testing against randomness

```
public bool DoRandomStuff()
{
        Random r = new Random();
        if (r.NextDouble() < 0.5)
        {
                return false;
        }
        return true;
}
```

The key is obviously to inject *controlled* randomness

```
public class RandomConsumer
{
        private Random random_;
        public RandomConsumer(Random r)
        {
                this.random_ = r;
        }
        public bool DoRandomStuff()
        {
                if (this.random_.NextDouble() < 0.5)
                        return false;
                return true;
        }
}
```

Tests

```
[TestMethod]
public void GetTrue()
{
        RandomConsumer rc = new RandomConsumer(new Random(0));
        Assert.IsTrue(rc.DoRandomStuff());
}
```

A seed value of 0 causes NextDouble to return approximately 0.7

```
[TestMethod]
public void GetTrue()
{
        RandomConsumer rc = new RandomConsumer(new Random(1));
        Assert.IsFalse(rc.DoRandomStuff());
}
```

A seed value of 1 causes NextDouble to return approximately 0.2

Testing against Guids

```
[TestMethod]
public void VerifyCreatedGuid()
{
        GuidClass gc = new GuidClass();

        Guid g = gc.DoStuff();

        Assert.AreNotEqual<Guid>(Guid.Empty, g);
}
```

For a newly created Guid, the only thing you are probably going to care about from a unit testing perspective is whether a new Guid was actually returned. However, you don’t need to be able to override the Guid creation process to unit test that.

Testing against the Current Time (Pt.1)

```
public bool IsTodayAWeekDay()
{
        DateTime now = DateTime.Now;
        if ((now.DayOfWeek == DayOfWeek.Saturday) || (now.DayOfWeek ==
                                                      DayOfWeek.Sunday))
        {
                return false;
        }
        return true;
}
```

Obviously, if you execute a test against this method on a weekday, the method will return true, but otherwise it will return false.

Testing against the Current Time (Pt.2)

```
[TestMethod]
public void CheckMonday()
{
        // 2007-05-07 is a Monday
        ServiceProvider<DateTime> dateTimeProvider = new ServiceProvider<DateTime>();

        dateTimeProvider.Preset(new DateTime(2007, 5, 7), “Now”);

        TimeConsumer tc = new TimeConsumer(dateTimeProvider);

        Assert.IsTrue(tc.IsTodayAWeekDay());
}
```

Testing against the Current Time (Pt.3)

```
public partial class TimeConsumer
{
        private ServiceProvider<DateTime> dateTimeProvider_;

        public TimeConsumer(ServiceProvider<DateTime> dateTimeProvider)
        {
                this.dateTimeProvider_ = dateTimeProvider;
        }

        public bool IsTodayAWeekDay()
        {
                DateTime now = this.dateTimeProvider_.Create(“Now”);
                if ((now.DayOfWeek == DayOfWeek.Saturday) || (now.DayOfWeek ==
                                                              DayOfWeek.Sunday))
                {
                        return false;
                }
                return true;
        }
}
```

Testing against the Passage of Time (Pt.1)

```
public void DoSomeWaiting()
{
        TimeSpan waitTime = TimeSpan.FromSeconds(30);

        DateTime initialTime = DateTime.Now;

        while (DateTime.Now < initialTime.Add(waitTime))
        {
                Thread.Sleep(TimeSpan.FromMilliseconds(10));
        }
}
```

Testing against the Passage of Time (Pt.2)

```
public void DoSomeWaiting()
{
        TimeSpan waitTime = TimeSpan.FromSeconds(30);

        DateTime initialTime = this.dateTimeProvider_.Create(“Now”);

        while (this.dateTimeProvider_.Create(“Now”) < initialTime.Add(waitTime))
        {
                Thread.Sleep(TimeSpan.FromMilliseconds(10));
        }
}
```

Testing against the Passage of Time (Pt.3)

```
[TestMethod]
public void PerformAcceleratedWait()
{
    ServiceProvider<DateTime> dateTimeProvider = new ServiceProvider<DateTime>();
    dateTimeProvider.Preset(new DateTime(2007, 5, 7), “Now”);
    TimeConsumer tc = new TimeConsumer(dateTimeProvider);
    Stopwatch watch = new Stopwatch();

    watch.Start();

    ThreadPool.QueueUserWorkItem(delegate(object state)
    {
        Thread.Sleep(TimeSpan.FromMilliseconds(10));
        dateTimeProvider.Preset(new DateTime(2007, 5, 7, 0, 0, 31), “Now”);
    });

    tc.DoSomeWaiting();

    watch.Stop();

    Assert.IsTrue(watch.Elapsed < TimeSpan.FromSeconds(30));
}
```

Mutation Testing

© https://ru.wikipedia.org/wiki/Мутационное_тестирование

Mutation Testing

Мутационное тестирование (*мутационный анализ* или *мутация программ*) — это метод тестирования программного обеспечения, который включает небольшие изменения кода программы.[1] Если набор тестов не в состоянии обнаружить такие изменения, то он рассматривается как недостаточный. Эти изменения называются *мутациями* и основываются на *мутирующих операторах*, которые или имитируют типичные ошибки программистов (например использование неправильной операции или имени переменной) или требуют создания полезных тестов.

Mutation Testing

Мутационное тестирование состоит в выборе мутирующих операторов и применения их одного за другим к каждому фрагменту исходного кода программы. Мутирующим оператором называется правило преобразования исходного кода. Результат одного применения мутационного оператора к программе называется *мутантом*. Если набор тестов способен обнаружить изменение (то есть один из тестов не проходит), то мутант называется *убитым*. Например, рассмотрим следующий фрагмент из программы

```
if (a && b) {
    c = 1;
} else {
    c = 0;
}
```

Оператор мутации условий заменит `&&` на `||`, и создаст следующий мутант:

```
if (a || b) {
    c = 1;
} else {
    c = 0;
}
```

Mutation Testing

Для того, чтобы тест мог убить этого мутанта, необходимо чтобы были выполнены следующие условия:

- Тест должен достигнуть (Reach) *мутированного оператора*.
- Входные данные теста должны привести к разным состояниям программы-мутанта (Infect) и исходной программы. Например, тест с `a = 1` и `b = 0` приведет к этому.
- Значение переменной `c` должно повлиять (Propagate) на вывод программы и быть проверено тестом.

Данные условия вместе называются *RIP моделью*.

Слабое мутационное тестирование (или *слабое мутационное покрытие*) требует выполнение только первых двух условий. *Сильное мутационное тестирование* требует выполнение всех трех условий и гарантирует что набор тестов в действительности может обнаружить изменение. Слабое мутационное тестирование тесно связано с методами покрытия кода. Проверка теста на соответстве условиям слабой мутации, требует намного меньше вычислений, чем проверка выполнения условий сильной мутации.

Mutation Testing

Мутационные операторы

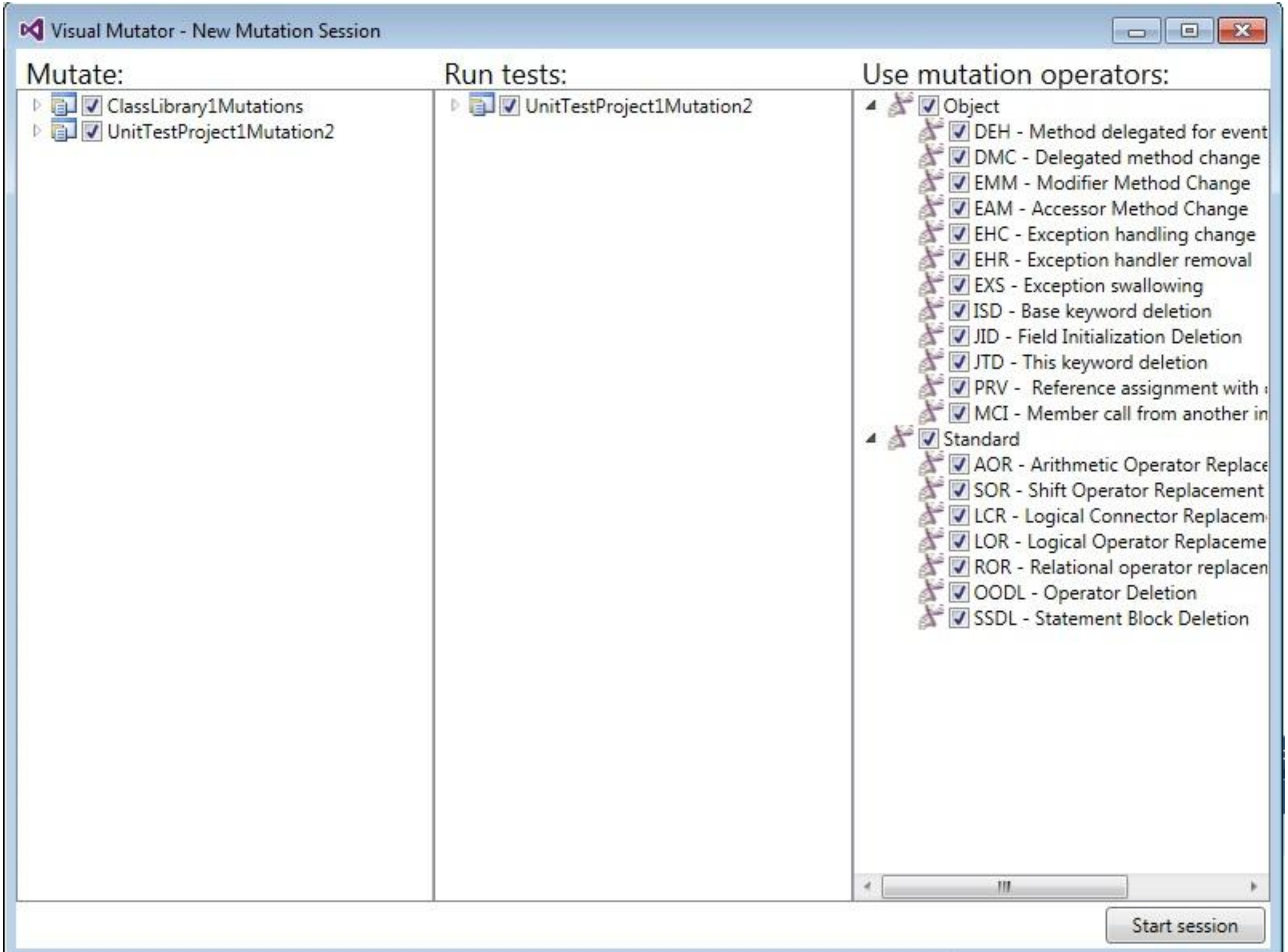
Многие виды мутационных операторов были исследованы. Например, для императивных языков следующие операторы могут быть использованы:

- Удалить оператор программы.
- Заменить каждое логическое выражение на логическую константу «истина» или «ложь».
- Заменить каждую арифметическую операцию на другую. Например, `+` на `*`, `-` или `/`.
- Заменить каждую логическую операцию на другую. Например, `>` на `>=`, `==` или `<=`.
- Заменить каждую переменную на другую (из той же области видимости). Переменные должны иметь одинаковые типы.

Кроме того существуют операторы для объектно-ориентированных языков,[4] операторы для параллельного программирования,[5] операторы для структур данных, таких как контейнеры[6] и др.

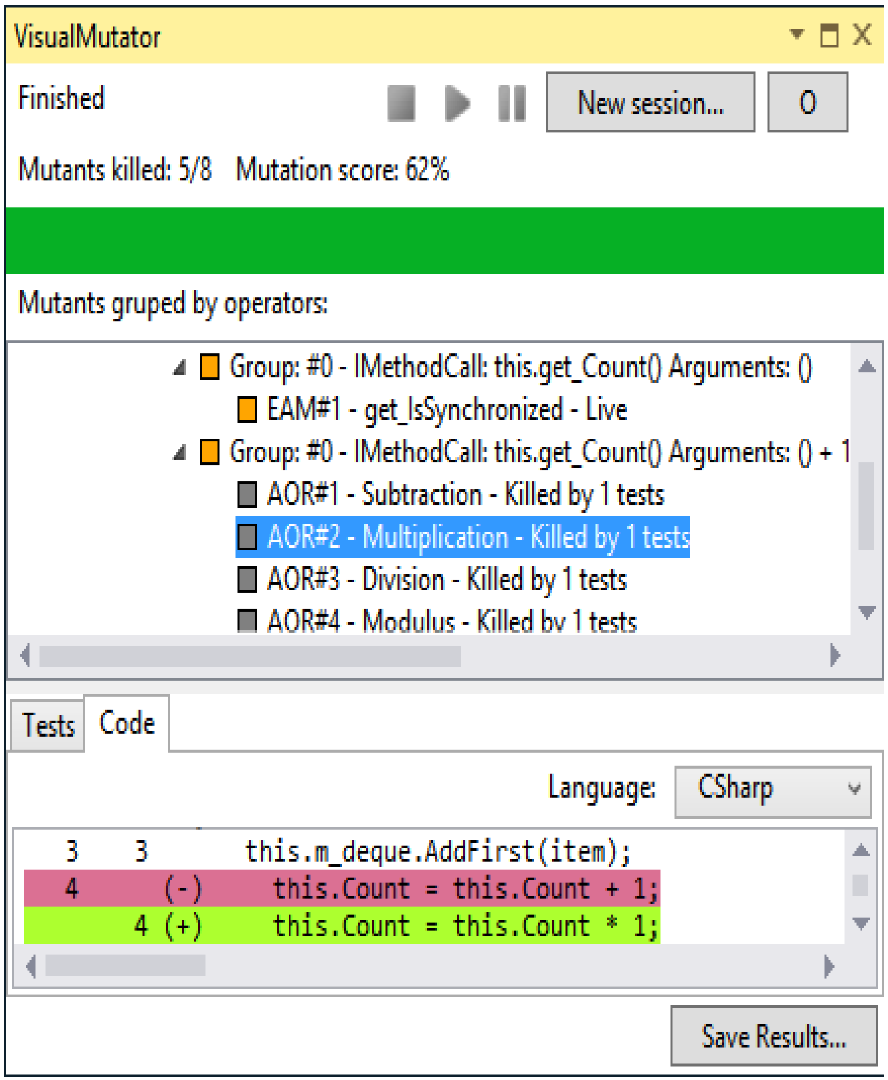
Tools

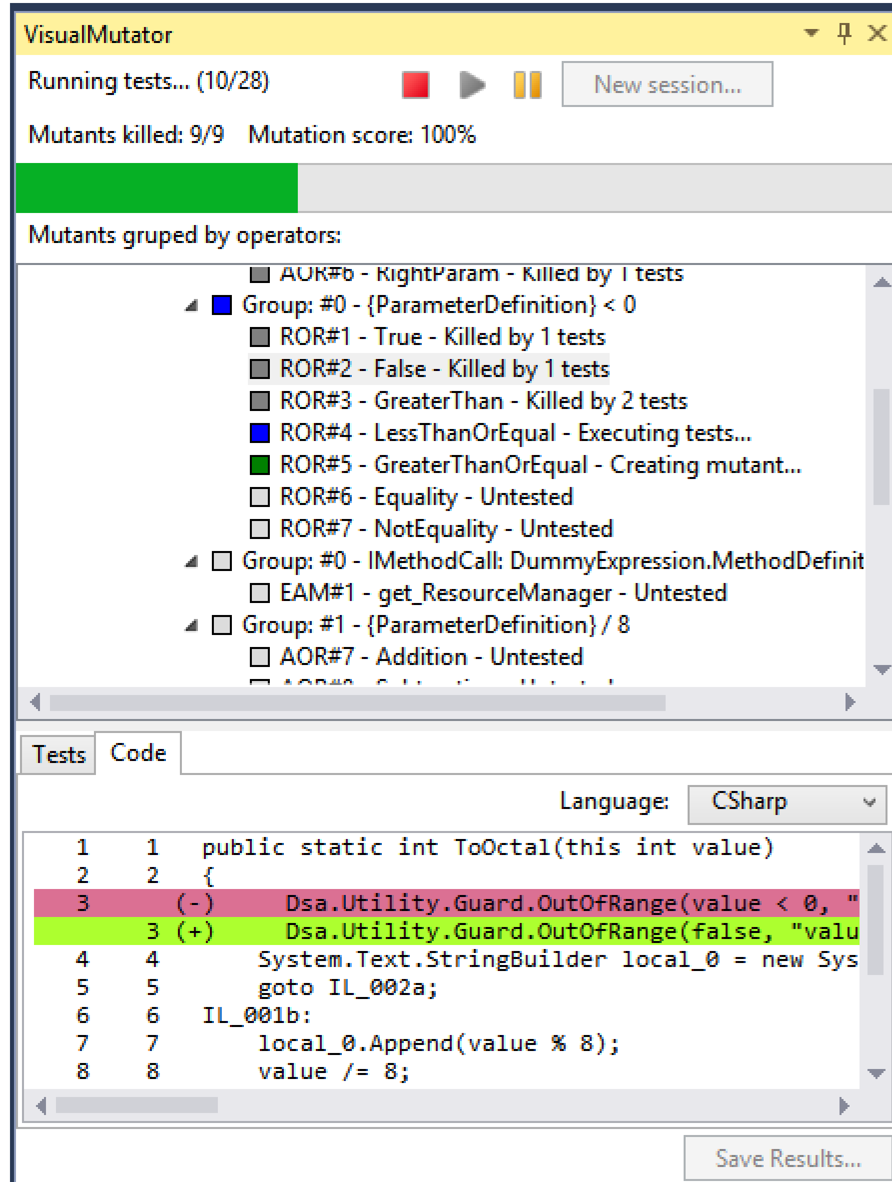
© http://visualmutator.github.io/web/



Tools

© http://visualmutator.github.io/web/





Test Patterns
And
Test Smells

Goals of Test Automation

Project Goals

- Tests as Specification
- Tests as Documentation
- Tests as Safety Net
- Defect Localization
- Easy to Write/Maintain
- Improve Quality
- Reduce Risk
- Bug Repellent

Test Writing Goals

- Fully Automated
- Self-Checking
- Repeatable Test
- Robust Test
- Simple Test
- Expressive Tests
- Separation of Concerns
- Do No Harm

Principles of Test Automation

- Write the Tests First
- Use the Front Door First
- Isolate the SUT
- Verify One Condition per Test
- Don't Modify the SUT
- Test Concerns Separately
- Minimize Test Overlap
- Keep Tests Independent
- Communicate Intent
- Minimize Untestable Code
- Keep Test Logic out of Production
- Ensure Commensurate Effort and Responsibility

Code Smells

- *Conditional Test Logic*
- *Hard-to-test Code*
- *Test Code Duplication*
- *Test Logic In Production*
- *Obscure Test*
  - *Eager Test*
  - *General Fixture*
  - *Indirect Testing*
  - *Mystery Guest*
  - *And more!*

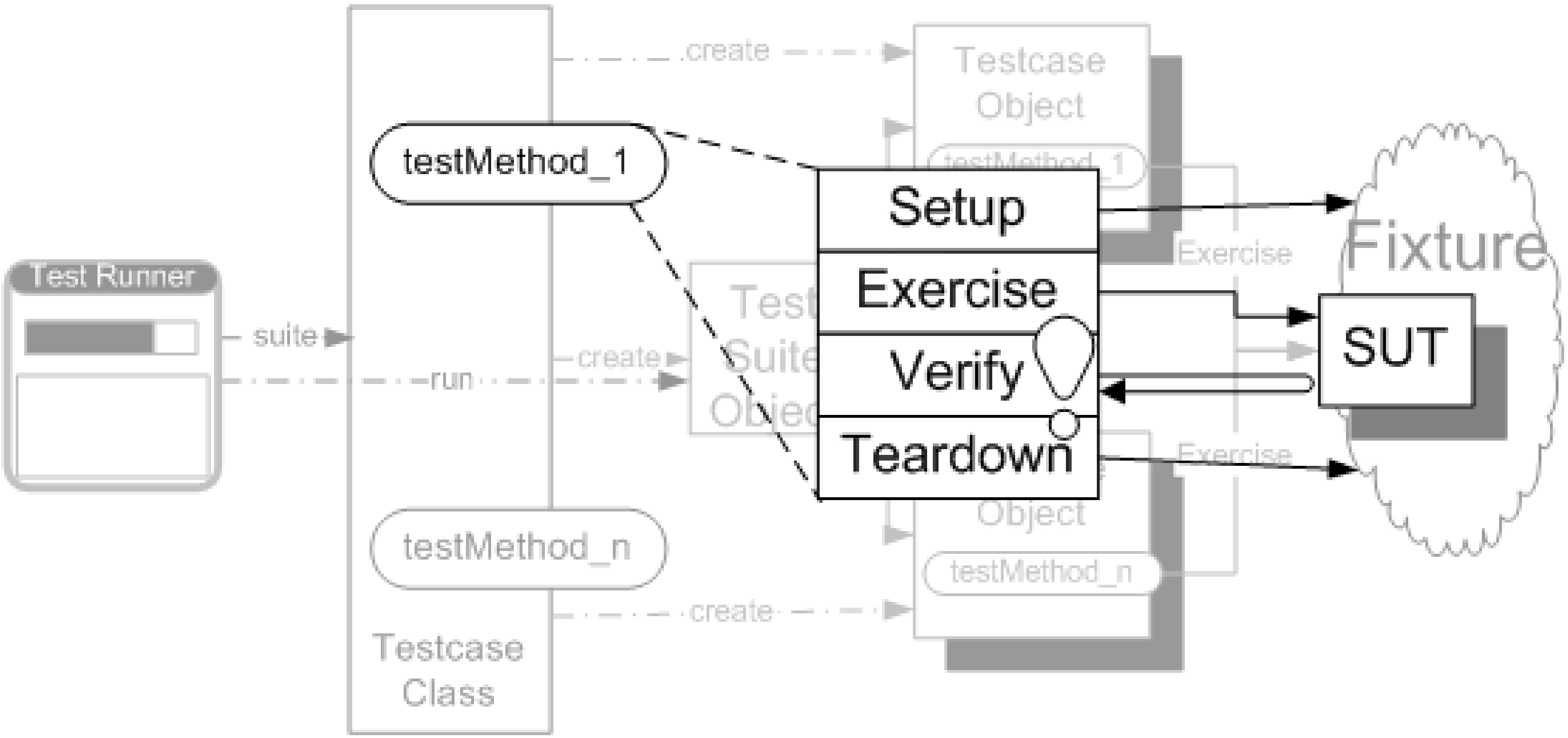
Behavior Smells

- *Erratic Test*
  - *Unrepeatable Test*
  - *Interacting Tests*
  - *Test Run War*
  - *Resource Optimism*
  - *And more!*
- *Slow Tests*
- *Assertion Roulette*
  - *Eager Test*
- *Frequent Debugging*
- *Manual Intervention*
- *Fragile Test*
  - *Fragile Fixture*

Project Smells

- *Buggy Tests*
- *Production Bugs*
- *Developers Not Writing Tests*
- *High Test Maintenance Cost*

Four-Phase Test



© Gerard Meszaros

Example (inline)

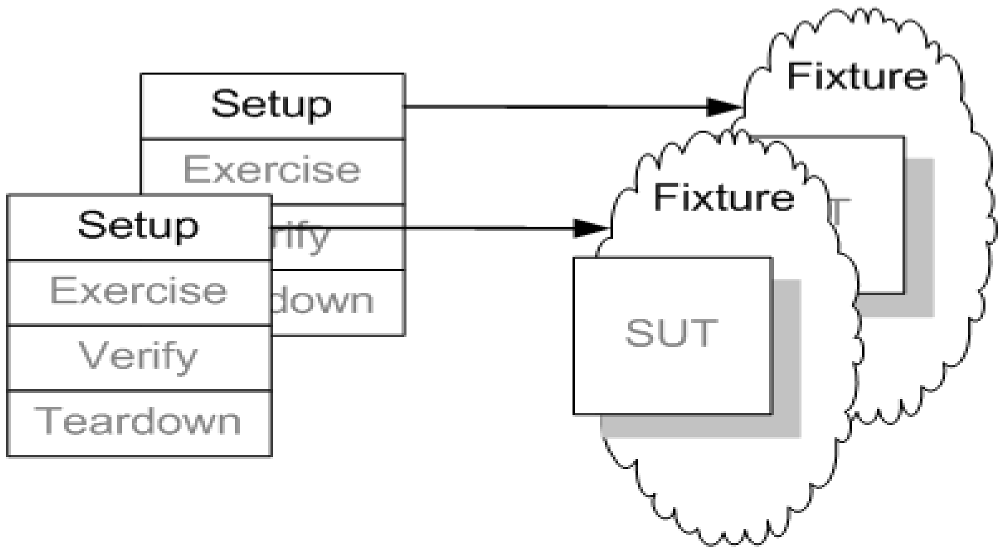
```
public void testGetFlightsByOriginAirport_NoFlights_inline() throws Exception {
     // Fixture setup
     NonTxFlightMngtFacade facade = new NonTxFlightMngtFacade();
     BigDecimal airportId = facade.createTestAirport("1OF");

     try {
        // Exercise System
        List flightsAtDestination1 = facade.getFlightsByOriginAirport(airportId);

        // Verify Outcome
        assertEquals( 0, flightsAtDestination1.size() );
     }
     finally {

        // Fixture teardown
        facade.removeAirport( airportId );
     }
  }
```

Example (setUp and tearDown)

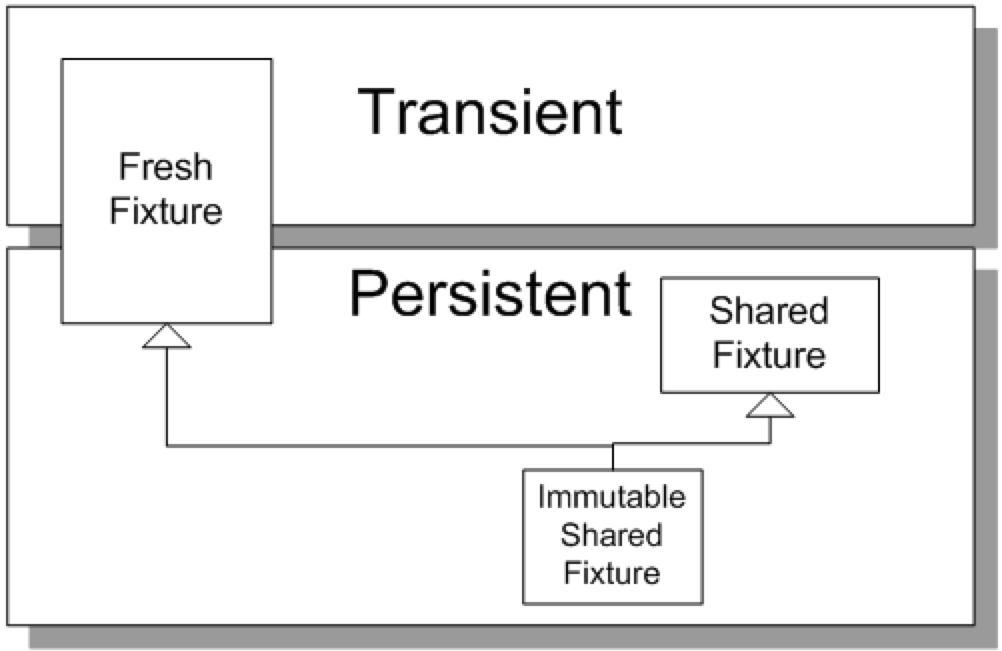
```
NonTxFlightMngtFacade facade = new NonTxFlightMngtFacade();
private BigDecimal airportId;

protected void setUp() throws Exception {
   // Fixture setup
   super.setUp();
   airportId = facade.createTestAirport("1OF");
}

public void testGetFlightsByOriginAirport_NoFlights_implicit() throws Exception {
   // Exercise SUT
   List flightsAtDestination1 = facade.getFlightsByOriginAirport(airportId);
   // Verify Outcome
   assertEquals( 0, flightsAtDestination1.size() );
}

protected void tearDown() throws Exception {
   // Fixture teardown
   facade.removeAirport(airportId);
   super.tearDown();
}
```

Fresh Fixture

Each test constructs its own brand-new test fixture for its own private use.



Fresh Fixture



Test Smells

Code smells:

- Obscure Test
- Conditional Test Logic
- Hard-to-Test code
- Test Code Duplication
- Test Logic in Production

Behaviour smells:

- Assertion Roulette
- Erratic Test
- Fragile Test
- Frequent Debugging
- Manual Intervention
- Slow Tests

Project smells:

- Buggy Tests
- Developers Not Writing Tests
- High Test Maintenance Cost
- Production Bugs

Test Smells: Obscure Test

Also known as: Long Test, Complex Test, Verbose Test

It is difficult to understand the test at a glance.

Causes:

Wrong Information:

- Eager Test
- Mystery Guest

Verbose Tests:

- General Fixture
- Irrelevant Information
- Hard-Coded Test Data
- Indirect Testing

Eager Test

The test is verifying too much functionality in a single Test Method.

```
public void testFlightMileage_asKm2() throws Exception {
     // setup fixture
     // exercise contructor
     Flight newFlight = new Flight(validFlightNumber);
     // verify constructed object
     assertEquals(validFlightNumber, newFlight.number);
     assertEquals("", newFlight.airlineCode);
     assertNull(newFlight.airline);
     // setup mileage
     newFlight.setMileage(1122);
     // exercise mileage translater
     int actualKilometres = newFlight.getMileageAsKm();
     // verify results
     int expectedKilometres = 1810;
     assertEquals( expectedKilometres, actualKilometres);
     // now try it with a canceled flight:
     newFlight.cancel();
     try {
        newFlight.getMileageAsKm();
        fail("Expected exception");
     } catch (InvalidRequestException e) {
        assertEquals( "Cannot get cancelled flight mileage", e.getMessage());
     }
  }
```

Mystery Guest

The test reader is not able to see the cause and effect between fixture and verification logic because part of it is done outside the Test Method.

```
 public void testGetFlightsByFromAirport_OneOutboundFlight_mg() throws Exception
{
     loadAirportsAndFlightsFromFile("test-flights.csv");

     // Exercise System
     List flightsAtOrigin = facade.getFlightsByOriginAirportCode( "YYC");

     // Verify Outcome
     assertEquals( 1, flightsAtOrigin.size());
     FlightDto firstFlight = (FlightDto) flightsAtOrigin.get(0);
     assertEquals( "Calgary", firstFlight.getOriginCity());
  }
```

Mystery Guest

A test depends on mysterious external resources making it difficult to understand the behavior that it is verifying. Mystery Guests may take many forms:

• A filename of an existing external file is passed to a method of the SUT; the contents of the file should determine the behavior of the SUT.

• The contents of a database record identified by a literal key are read into an object that is then used by the test or passed to the SUT.

• The contents of a file is read and used in calls to Assertion Methods to verify the expected outcome.

• A Setup Decorator is used to create a Shared Fixture and objects in the fixture are referenced via variables within the result verification logic.

• A General Fixture is set up using Implicit Setup (page X) and the Test Methods access them via instance variables or class variables.

General Fixture

The test is building or referencing a larger fixture than is needed to verify the functionality in question

```
public void testGetFlightsByFromAirport_OneOutboundFlight() throws Exception {
     setupStandardAirportsAndFlights();
     FlightDto outboundFlight = findOneOutboundFlight();
     // Exercise System
     List flightsAtOrigin = facade.getFlightsByOriginAirport(
                    outboundFlight.getOriginAirportId());
     // Verify Outcome
     assertOnly1FlightInDtoList( "Flights at origin", outboundFlight,
                                 flightsAtOrigin);
  }

  public void testGetFlightsByFromAirport_TwoOutboundFlights() throws Exception {
     setupStandardAirportsAndFlights();
     FlightDto[] outboundFlights = findTwoOutboundFlightsFromOneAirport();
     // Exercise System
     List flightsAtOrigin = facade.getFlightsByOriginAirport(
                    outboundFlights[0].getOriginAirportId());
     // Verify Outcome
     assertExactly2FlightsInDtoList( "Flights at origin", outboundFlights,
                                     flightsAtOrigin);
  }
```

Irrelevant Information

The test is exposing a lot of irrelevant details about the fixture that distract the test reader from what really affects the behavior of the SUT.

```
public void testAddItemQuantity_severalQuantity_v10(){
     //   Setup Fixture
     Address billingAddress = createAddress( "1222 1st St SW", "Calgary", "Alberta", "T2N 2V2",
                                                                                  "Canada");
     Address shippingAddress = createAddress( "1333 1st St SW", "Calgary", "Alberta", "T2N 2V2",
                                                                                  "Canada");
     Customer customer = createCustomer( 99, "John", "Doe", new BigDecimal("30"),
                                                            billingAddress, shippingAddress);
     Product product = createProduct( 88,"SomeWidget",new BigDecimal("19.99"));
     Invoice invoice = createInvoice(customer);
     // Exercise SUT
     invoice.addItemQuantity(product, 5);
     // Verify Outcome
     LineItem expected = new LineItem(invoice, product,5, new BigDecimal("30"), new BigDecimal("69.96"));
     assertContainsExactlyOneLineItem(invoice, expected);
  }
```

Hard-Coded Test Data

Data values in the fixture, assertions or arguments of the SUT are hard-coded in the Test Method obscuring cause-effect relationships between inputs and expected outputs.

```
public void testAddItemQuantity_severalQuantity_v12() {
     //  Setup Fixture
     Customer cust = createACustomer(new BigDecimal("30"));
     Product prod = createAProduct(new BigDecimal("19.99"));
     Invoice invoice = createInvoice(cust);
     // Exercise SUT
     invoice.addItemQuantity(prod, 5);
     // Verify Outcome
     LineItem expected = new LineItem(invoice, prod, 5,
           new BigDecimal("30"), new BigDecimal("69.96"));
     assertContainsExactlyOneLineItem(invoice, expected);
  }
```

Hard-Coded Test Data

The best way to get rid of Obscure Test is to **replace** literal constants with something else. Fixture values that determine which scenario is being executed (e.g. type codes) are probably the only ones that are reasonable to leave as literals but even these can be converted to named constants.

Fixture values **that do not matter to the test** (those which do not affect the expected outcome) should be **defaulted** within Creation Methods. In this way we say to the test reader "the values you don't see don't affect the expected outcome". We can replace fixture values that appear in both the fixture setup and outcome verification parts of the test with suitably initialized named constants as long as we are using a Fresh Fixture approach to fixture setup.

Values in the result verification logic **that are based on values used in the fixture or as arguments of the SUT** should be **replaced with Derived Values** to make that calculations obvious to the test reader.

If we are using any variant of Shared Fixture, we should try to use Distinct Generated Values to ensure that each time a test is run it uses a different value. This is especially important for fields that are used as unique keys in databases. A common way of encapsulating this logic is through the use of Anonymous Creation Methods.

Indirect Testing

The Test Method is interacting with the SUT indirectly via another object thereby making the interactions more complex

```
private final int LEGAL_CONN_MINS_SAME = 30;

public void testAnalyze_sameAirline_LessThanConnectionLimit() throws Exception {
      // setup
      FlightConnection illegalConn = createSameAirlineConn( LEGAL_CONN_MINS_SAME - 1);
      // exercise
      FlightConnectionAnalyzerImpl sut = new FlightConnectionAnalyzerImpl();
      String actualHtml = sut.getFlightConnectionAsHtmlFragment( illegalConn.getInboundFlightNumber(),
                        illegalConn.getOutboundFlightNumber());
      // verification
      StringBuffer expected = new StringBuffer();
      expected.append("<span class=”boldRedText”>");
      expected.append("Connection time between flight ");
      expected.append(illegalConn.getInboundFlightNumber());
      expected.append(" and flight ");
      expected.append(illegalConn.getOutboundFlightNumber());
      expected.append(" is ");
      expected.append(illegalConn.getActualConnectionTime());
      expected.append(" minutes.</span>");
      assertEquals("html", expected.toString(), actualHtml);
}
```

Test Smells: Conditional Test Logic

Also known as: Indented Test Code

A test contains code that may or may not be executed

Causes:

Flexible Test
Conditional Verification Logic
Production Logic In Test
Complex Teardown
Multiple Test Conditions

Flexible Test

Test code verifies different functionality depending on when or where it is run

```
public void testDisplayCurrentTime_whenever() {
     // fixture setup
     TimeDisplay sut = new TimeDisplay();
     // exercise sut
     String result = sut.getCurrentTimeAsHtmlFragment();
     // verify outcome
     Calendar time = new DefaultTimeProvider().getTime();
     StringBuffer expectedTime = new StringBuffer();
     expectedTime.append("<span class=\"tinyBoldText\">");
     if ((time.get(Calendar.HOUR_OF_DAY) == 0)
        && (time.get(Calendar.MINUTE) <= 1)) {
        expectedTime.append( "Midnight");
     } else if ((time.get(Calendar.HOUR_OF_DAY) == 12)
                && (time.get(Calendar.MINUTE) == 0)) { // noon
        expectedTime.append("Noon");
     } else  {
        SimpleDateFormat fr = new SimpleDateFormat("h:mm a");
        expectedTime.append(fr.format(time.getTime()));
     }
     expectedTime.append("</span>");
     assertEquals( expectedTime, result);
  }
```

Conditional Verification Logic

The use of Conditional Test Logic to verify the expected outcome. This is usually caused by wanting to prevent the execution of assertions if the SUT fails to return the right objects or the use of loops to verify the contents of collections returned by the SUT.

```
//   verify Vancouver is in the list:
actual = null;
i = flightsFromCalgary.iterator();
while (i.hasNext()) {
   FlightDto flightDto = (FlightDto) i.next();
   if (flightDto.getFlightNumber().equals( expectedCalgaryToVan.getFlightNumber()))
   {
      actual = flightDto;
      assertEquals("Flight from Calgary to Vancouver", expectedCalgaryToVan,
                   flightDto);
      break;
   }
}
```

Production Logic In Test

Some forms of Conditional Test Logic are found in the result verification section of tests

```
public void testCombinationsOfInputValues() {
    // Setup Fixture:
    Calculator sut = new Calculator();
    int expected;  // TBD inside loops
    for (int i = 0; i < 10; i++) {
       for (int j = 0; j < 10; j++) {
          // Exercise SUT:
          int actual = sut.calculate( i, j );

          // Verify result:
          if (i==3 & j==4)  // special case
             expected = 8;
          else
             expected = i+j;
             assertEquals(message(i,j), expected, actual);
       }
    }
 }
```

Complex Teardown

Complex fixture teardown code is more likely to leave test environment corrupted by not cleaning up correctly. It is hard to verify that it has been written correctly and can easily result in "data leaks" that may later cause this or other tests to fail for no apparent reason

```
...

try {
         inboundAirport = createTestAirport("1IF");
         expFlightDto = createTestFlight(outboundAirport, inboundAirport);
         // Exercise System
         List flightsAtDestination1 = facade.getFlightsByOriginAirport(inboundAirport);
         // Verify Outcome
         assertEquals(0,flightsAtDestination1.size());
      } finally {
         try {
            facade.removeFlight(expFlightDto.getFlightNumber());
         } finally {
            try {
               facade.removeAirport(inboundAirport);
            } finally  {
               facade.removeAirport(outboundAirport);
            }
         }
      }
```

Multiple Test Conditions

A test is trying to apply the same test logic to many sets of input values each with their own corresponding expected result. In this example, the test is iterating over a collection of test values and applying the test logic to each set

```
public void testMultipleValueSets() {
     // Setup Fixture:
     Calculator sut = new Calculator();
     TestValues[] testValues = { new TestValues(1,2,3),
                    new TestValues(2,3,5),
                    new TestValues(3,4,8), // special case!
                    new TestValues(4,5,9) };
     for (int i = 0; i < testValues.length; i++) {
        TestValues values = testValues[i];
        // Exercise SUT:
        int actual = sut.calculate( values.a, values.b);
        // Verify result:
        assertEquals(message(i), values.expectedSum, actual);
     }
  }
```