1. 了解測試障礙的可能原因

2. 識別建構子的問題

2.1. 測試性不良的設計

2.2. 以依賴注入提升測試性的設計

3. 識別初始化的問題

3.1. 測試性不良的設計

3.2. 以依賴注入提升測試性的設計

4. 識別private方法對單元測試的影響

4.1. 測試性不良的設計

4.2. 使用關注分離提升測試性的設計

4.3. 提升存取層級以提升測試性

5. 識別static方法對單元測試的影響

5.1. 測試性不良的設計

5.2. 以非static的方法重構

5.3. 啟用Mockito偽冒static的支援

6. 識別final方法對單元測試的影響

6.1. 測試性不良的設計

6.2. 以非final的方法重構

7. 識別final類別對單元測試的影響

7.1. 測試性不良的設計

7.2. 以建立final類別的介面進行重構

8. 識別使用new呼叫建構子造成的測試問題

8.1. 測試性不良的設計

8.2. 以依賴注入提升測試性的設計

9. 識別使用static變數和程式碼區塊造成的測試問題

1. 了解測試障礙的可能原因

我們了解了一些商業邏輯程式碼的撰寫方式將造成單元測試進行的困難，稱「測試障礙 (test impediments)」，如連線資料庫或印表機等，此時必須重構程式碼並將測試障礙移至另一個類別或方法；撰寫單元測試程式碼時則將移出的測試障礙替換為Mock Object。PowerMock的功能強大足以解決很多測試難題，但對象應該是legacy程式碼；若有機會開發新的商業邏輯程式碼應該還是提升「可測試性 (testability)」並做到「測試友善 (test friendly)」。

Java語言的類別結構中，有幾種設計可能是不利(unfavorable)於測試的：

* 在建構子裡建立測試障礙物件，或是建構子相依於測試障礙。
* 方法使用private宣告。
* 方法使用static宣告。
* 方法使用final宣告。
* 類別使用final宣告。
* 使用new建立物件。
* 類別欄位使用static宣告。
* 程式碼區塊使用static宣告。

以下章節將逐一說明解決方式。

為了表示測試障礙，以下範例裡將定義一個屬於RuntimeException的例外類別TestingImpedimentException。如果測試過程中拋出TestingImpedimentException錯誤，除了表示無法自動化測試外，也表示商業邏輯程式碼是不利於測試的，必須予以重構：

【範例】/testability/src/main/java/lab/testability/TestingImpedimentException.java

|  |  |
| --- | --- |
| 1 | public class TestingImpedimentException extends RuntimeException { |
| 2 | private static final long serialVersionUID = 1L; |
| 3 | public TestingImpedimentException(String msg) { |
| 4 | super(msg); |
| 5 | } |
| 6 | public TestingImpedimentException() { |
| 7 | } |
| 8 | } |

2. 識別建構子的問題

要進行單元測試，我們需要在測試工具中建立類別的物件實例，但常見問題是在建立物件時必須連帶建立其它依賴的物件實例，如建立資料庫連線以取得資料，或是取得屬性檔(\*.properties)的資料等。如果這類別有很多呼叫者，就更不能變更建構子以傳遞依賴關係，否則將導致編譯或執行時期錯誤。

以下先定義兩個測試障礙類別，分別是「DatabaseDependency」與「FileReadDependency」，代表連線資料庫和存取實體檔案；因此這兩個類別應該以Mock Object形式參與單元測試，不能以建構子產生真正物件。

我們在建構子裡安插拋出TestingImpedimentException的程式碼，在單元測試時若是建立了這兩個類別的真實物件就會馬上拋出錯誤訊息，所以只能建立Mock Object，分別如下：

【範例】/testability/src/main/java/lab/testability/DatabaseDependency.java

|  |  |
| --- | --- |
| 1 | public class DatabaseDependency { |
| 2 | public DatabaseDependency() { |
| 3 | throw new TestingImpedimentException("Calls database"); |
| 4 | } |
| 5 | } |

【範例】/testability/src/main/java/lab/testability/FileReadDependency.java

|  |  |
| --- | --- |
| 1 | public class FileReadDependency { |
| 2 | public FileReadDependency() { |
| 3 | throw new TestingImpedimentException("Reads file"); |
| 4 | } |
| 5 | } |

【2.1】測試性不良的設計

不良的商業邏輯類別BadConstructor示範如下，直接在建構子中使用「new」建立依賴關係：

【範例】/testability/src/main/…/constructor/bad/BadConstructor.java

|  |  |
| --- | --- |
| 1 | public class BadConstructor { |
| 2 | private DatabaseDependency dependency1; |
| 3 | private FileReadDependency dependency2; |
| 4 |  |
| 5 | public BadConstructor() { |
| 6 | this.dependency1 = **new DatabaseDependency()**; |
| 7 | this.dependency2 = **new FileReadDependency()**; |
| 8 | } |
| 9 |  |
| 10 | public Object someMethod(Object arg) { |
| 11 | return arg; |
| 12 | } |
| 13 | } |

撰寫單元測試程式碼：

【範例】/testability/src/test/…/constructor/bad/BadConstructorTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class BadConstructorTest { |
| 3 |  |
| 4 | BadConstructor instance; |
| 5 |  |
| 6 | @Before |
| 7 | public void setUp() { |
| 8 | instance = **new BadConstructor();** |
| 9 | } |
| 10 |  |
| 11 | @Test |
| 12 | public void show\_SUT\_is\_not\_null() throws Exception { |
| 13 | assertNotNull(instance); |
| 14 | assertEquals("Jim", instance.someMethod("Jim")); |
| 15 | } |
| 16 | } |

進行測試時，因為行8建構BadConstructor物件時因為必須先建構DatabaseDependency與FileReadDependency，因此拋出TestingImpedimentException並終止測試：

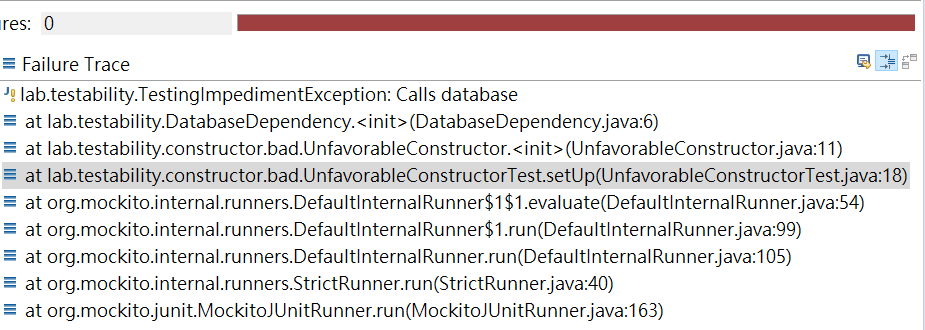


圖1

如果當成legacy程式碼就必須使用PowerMockito抑止建構子的執行，可以使用Whitebox.newInstance()方法避開建構子產生物件：

【範例】/testability/src/test/…/bad/BadConstructorTestPowerMock.java

|  |  |
| --- | --- |
| 1 | public class BadConstructorTestPowerMock { |
| 2 | @Test |
| 3 | public void show\_object\_is\_not\_null() throws Exception { |
| 4 | BadConstructor instance |
| 5 | = **Whitebox.newInstance**(BadConstructor.class); |
| 6 | assertNotNull(instance); |
| 7 | assertEquals("Jim", instance.someMethod("Jim")); |
| 8 | } |
| 9 | } |

【2.2】以依賴注入提升測試性的設計

若要撰寫具備可測試性的程式碼，我們應該透過建構子「注入」依賴關係，而不是在建構子中「直接建立」依賴關係，這也符合「依賴注入」的觀念。所以把商業邏輯程式碼重構如下：

【範例】/testability/src/main/…/constructor/good/GoodConstructor.java

|  |  |
| --- | --- |
| 1 | public class GoodConstructor { |
| 2 | private DatabaseDependency dep1; |
| 3 | private FileReadDependency dep2; |
| 4 |  |
| 5 | public GoodConstructor(DatabaseDependency d1, FileReadDependency d2) { |
| 6 | this.dep1 = d1; |
| 7 | this.dep2 = d2; |
| 8 | } |
| 9 | public Object someMethod(Object arg) { |
| 10 | return arg; |
| 11 | } |
| 12 | } |

也把單元測試程式碼修改如下。測試障礙類別DatabaseDependency與FileReadDependency不能直接建立物件，應該以Mock Object參與單元測試，並透過建構子「注入」依賴關係：

【範例】/testability/src/test/…/constructor/good/GoodConstructorTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class GoodConstructorTest { |
| 3 |  |
| 4 | **@Mock** |
| 5 | **DatabaseDependency dep1;** |
| 6 | **@Mock** |
| 7 | **FileReadDependency dep2;** |
| 8 |  |
| 9 | GoodConstructor instance; |
| 10 |  |
| 11 | @Before |
| 12 | public void setUp() { |
| 13 | instance = new GoodConstructor(**dep1**, **dep2**); |
| 14 | } |
| 15 |  |
| 16 | @Test |
| 17 | public void show\_object\_is\_not\_null() throws Exception { |
| 18 | assertNotNull(instance); |
| 19 | assertEquals("Jim", instance.someMethod("Jim")); |
| 20 | } |
| 21 | } |

也方便使用@InjectMocks簡化程式碼，如行9-10：

【範例】/testability/src/test/…/constructor/good/GoodConstructorTest2.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class GoodConstructorTest2 { |
| 3 |  |
| 4 | @Mock |
| 5 | DatabaseDependency dep1; |
| 6 | @Mock |
| 7 | FileReadDependency dep2; |
| 8 |  |
| 9 | **@InjectMocks** |
| 10 | **GoodConstructor instance;** |
| 11 |  |
| 12 | @Test |
| 13 | public void show\_object\_is\_not\_null() throws Exception { |
| 14 | assertNotNull(instance); |
| 15 | assertEquals("Jim", instance.someMethod("Jim")); |
| 16 | } |
| 17 | } |

3. 識別初始化的問題

【3.1】測試性不良的設計

宣告類別層級的變數並同時完成初始化將造成無法偽冒該變數值，如以下範例行2：

【範例】/testability/src/main/…/instantiate/bad/BadVariableInitialization.java

|  |  |
| --- | --- |
| 1 | public class BadVariableInitialization { |
| 2 | DatabaseDependency dependency1 = **new DatabaseDependency();** |
| 3 |  |
| 4 | public Object someMethod(Object arg) { |
| 5 | return arg; |
| 6 | } |
| 7 | } |

撰寫單元測試：

【範例】/testability/src/test/…/instantiate/bad/BadVariableInitializationTest.java

|  |  |
| --- | --- |
| 1 | public class BadVariableInitializationTest { |
| 2 | @Test |
| 3 | public void show\_object\_is\_not\_null() throws Exception { |
| 4 | BadVariableInitialization instance= **new BadVariableInitialization();** |
| 5 | assertNotNull(instance); |
| 6 | assertEquals("Jim", instance.someMethod("Jim")); |
| 7 | } |
| 8 | } |

執行時因為行4建立SUT必須先建立DatabaseDependency欄位並完成初始化，因此拋出TestingImpedimentException並終止測試：

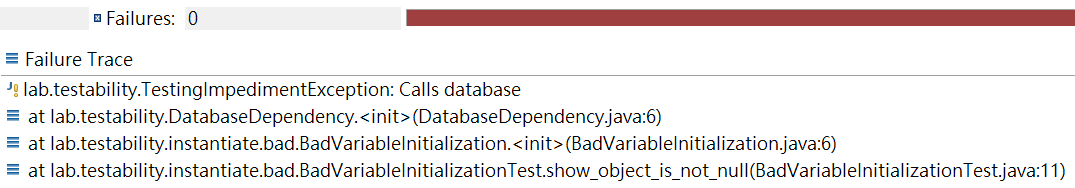


圖2

使用PowerMockito可以將建立BadVariableInitialization所需要的DatabaseDependency欄位在進行單元測試時以Mock Object取代，如以下單元測試。只要在行15建立BadVariableInitialization前先聲明一旦以new呼叫DatabaseDependency類別的建構子，則回傳預定義的DatabaseDependency的Mock Object，如行10-13：

【範例】/testability/src/test/…/bad/BadVariableInitializationTestPowerMock.java

|  |  |
| --- | --- |
| 1 | @RunWith(PowerMockRunner.class) |
| 2 | @PrepareForTest(BadVariableInitialization.class) |
| 3 | public class BadVariableInitializationTestPowerMock { |
| 4 |  |
| 5 | **@Mock** |
| 6 | **DatabaseDependency dependency;** |
| 7 |  |
| 8 | @Test |
| 9 | public void show\_object\_is\_not\_null() throws Exception { |
| 10 | **PowerMockito** |
| 11 | **.whenNew(DatabaseDependency.class)** |
| 12 | **.withNoArguments()** |
| 13 | **.thenReturn(dependency);** |
| 14 |  |
| 15 | BadVariableInitialization instance = new BadVariableInitialization(); |
| 16 | assertNotNull(instance); |
| 17 | assertEquals("Jim", instance.someMethod("Jim")); |
| 18 | } |
| 19 | } |

【3.2】以依賴注入提升測試性的設計

解決方式是將類別重構如下，欄位的初始化改以「依賴注入」的方式進行：

【範例】/testability/src/main/…/instantiate/good/GoodVariableInitialization.java

|  |  |
| --- | --- |
| 1 | public class GoodVariableInitialization { |
| 2 | DatabaseDependency dependency1; |
| 3 |  |
| 4 | public GoodVariableInitialization(DatabaseDependency d) { |
| 5 | this.dependency1 = d; |
| 6 | } |
| 7 |  |
| 8 | public Object someMethod(Object arg) { |
| 9 | return arg; |
| 10 | } |
| 11 | } |

改版後的單元測試如下：

【範例】/testability/src/test/…/instantiate/good/GoodVariableInitializationTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class GoodVariableInitializationTest { |
| 3 |  |
| 4 | @Mock |
| 5 | DatabaseDependency dependency; |
| 6 |  |
| 7 | @InjectMocks |
| 8 | GoodVariableInitialization instance; |
| 9 |  |
| 10 | @Test |
| 11 | public void show\_object\_is\_not\_null() throws Exception { |
| 12 | assertNotNull(instance); |
| 13 | assertEquals("Jim", instance.someMethod("Jim")); |
| 14 | } |
| 15 | } |

若類別欄位的型態不涉及「測試障礙」還是可以在宣告時一併初始化完成，如List<String> field = new ArrayList<>()，讓類別負責自己的內部欄位。

4. 識別private方法對單元測試的影響

【4.1】測試性不良的設計

private方法對於隱藏物件內部狀態和封裝很有用，但是也有可能包含測試障礙的邏輯：

【範例】/testability/src/main/…/privates/bad/PrivateMethodDemo.java

|  |  |
| --- | --- |
| 1 | public class PrivateMethodDemo { |
| 2 | public Object validate(Object arg) { |
| 3 | if (arg == null) { |
| 4 | showError("Null input"); |
| 5 | } |
| 6 | return arg; |
| 7 | } |
| 8 | **private** void showError(String msg) { |
| 9 | new GraphicalInterface().showMessage(msg); |
| 10 | } |
| 11 | } |

類別PrivateMethodDemo具備public的validate()方法，若傳入的參數為null等同驗證訊息失敗，將呼叫另一個private的方法showError()，再轉呼叫另一類別GraphicalInterface的showMessage()方法。該方法模擬使用者介面的訊息彈出，因此需要與使用者互動，屬於測試障礙，將拋出TestingImpedimentException：

【範例】/testability/src/main/java/lab/testability/GraphicalInterface.java

|  |  |
| --- | --- |
| 1 | public class GraphicalInterface { |
| 2 | public void showMessage(String msg) { |
| 3 | throw new TestingImpedimentException("GUI need manual operation"); |
| 4 | } |
| 5 | } |

撰寫單元測試程式碼如下，執行showMessage()時將出錯：

【範例】/testability/src/test/…/privates/bad/PrivateMethodTest.java

|  |  |
| --- | --- |
| 1 | public class PrivateMethodTest { |
| 2 | PrivateMethodDemo privateMethod; |
| 3 | @Before |
| 4 | public void setUp() { |
| 5 | privateMethod = new PrivateMethodDemo(); |
| 6 | } |
| 7 | @Test |
| 8 | public void validate() throws Exception { |
| 9 | privateMethod.validate(null); |
| 10 | } |
| 11 | } |

【4.2】使用關注分離提升測試性的設計

建立一個新的類別，將測試障礙移到該類別，然後將新類別依賴注入到原類別，如此private方法就可以呼叫依賴注入的結果。重構後如下：

【範例】/testability/src/main/…/privates/good/PrivateMethodInjection.java

|  |  |
| --- | --- |
| 1 | public class PrivateMethodInjection { |
| 2 | private GraphicalInterface graphicalInterface; |
| 3 | public PrivateMethodInjection(GraphicalInterface ui) { |
| 4 | this.graphicalInterface = ui; |
| 5 | } |
| 6 |  |
| 7 | public Object validate(Object arg) { |
| 8 | if (arg == null) { |
| 9 | showError("Null input"); |
| 10 | } |
| 11 | return arg; |
| 12 | } |
| 13 | private void showError(String msg) { |
| 14 | graphicalInterface.showMessage(msg); |
| 15 | } |
| 16 | } |

改寫單元測試：

【範例】/testability/src/test/…/privates/good/PrivateMethodInjectionTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class PrivateMethodInjectionTest { |
| 3 | @Mock |
| 4 | GraphicalInterface graphicalInterface; |
| 5 | @InjectMocks |
| 6 | PrivateMethodInjection privateMethod; |
| 7 | @Test |
| 8 | public void validate() throws Exception { |
| 9 | privateMethod.validate(null); |
| 10 | } |
| 11 | } |

【4.3】提升存取層級以提升測試性

將有測試障礙的private方法提升存取層級到default或protected，如行8：

【範例】/testability/src/main/…/privates/good/PrivateMethod2Default.java

|  |  |
| --- | --- |
| 1 | public class PrivateMethod2Default { |
| 2 | public Object validate(Object arg) { |
| 3 | if (arg == null) { |
| 4 | showError("Null input"); |
| 5 | } |
| 6 | return arg; |
| 7 | } |
| 8 | void showError(String msg) { |
| 9 | new GraphicalInterface().showMessage(msg); |
| 10 | } |
| 11 | } |

改寫單元測試：

【範例】/testability/src/test/…/privates/good/PrivateMethod2DefaultTest.java

|  |  |
| --- | --- |
| 1 | public class PrivateMethod2DefaultTest { |
| 2 | PrivateMethod2Default privateMethod; |
| 3 | @Before |
| 4 | public void setUp() { |
| 5 | **privateMethod = new PrivateMethod2Default() {** |
| 6 | **void showError(String msg) {** |
| 7 | **// do nothing** |
| 8 | **}** |
| 9 | **};** |
| 10 | } |
| 11 | @Test |
| 12 | public void validate() throws Exception { |
| 13 | privateMethod.validate(null); |
| 14 | } |
| 15 | } |

因為showError()方法的存取層級更改為default，建立PrivateMethod2Default物件實例時可以採用匿名(anonymous)類別的作法覆寫(override)原本呼叫測試障礙的邏輯。如此正式的商業邏輯程式碼依然呼叫GraphicalInterface的showMessage()，但單元測試時就避開無法測試的邏輯，讓其它單元測試可以繼續自動化執行。這種針對測試障礙建立一個覆寫的版本的作法稱為建立Fake Object，若是原類別含有多個測試障礙的方法，可以在測試類別建立內部(inner)類別繼承原類別，然後覆寫所有含有測試障礙的方法。

由本案例也可得知，撰寫正式的商業邏輯程式碼時，應該盡量不要將測試障礙隱藏在private方法中。

5. 識別static方法對單元測試的影響

【5.1】測試性不良的設計

static方法常用於工具/設施類(utility)的類別，如Math類別。如先前內容所述Mockito在較新的版本可以選用對static方法的支援，但預設沒有，因此我們還是在使用static方法時拋出測試障礙的例外：

【範例】/testability/src/main/…/staticmethods/bad/StaticMethodDemo.java

|  |  |
| --- | --- |
| 1 | public class StaticMethodDemo { |
| 2 | public **static** void aStaticMethod() { |
| 3 | throw new **TestingImpedimentException**("Calls static method"); |
| 4 | } |
| 5 | } |

單元測試如下。我們嘗試建立StaticMethodDemo的Mock Object，並偽冒惟一的static方法aStaticMethod()並預期「doNothing」；若偽冒失敗就會呼叫原本的程式邏輯，將拋出TestingImpedimentException：

【範例】/testability/src/test/…/staticmethods/bad/StaticMethodTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class StaticMethodTest { |
| 3 | @Mock |
| 4 | StaticMethodDemo staticMethodDemo; |
| 5 | @Test |
| 6 | public void show\_mock\_static\_method() throws Exception { |
| 7 | // Given |
| 8 | Mockito.doNothing().when(staticMethodDemo).**aStaticMethod**(); |
| 9 | // When |
| 10 | staticMethodDemo.**aStaticMethod**(); |
| 11 | } |
| 12 | } |

單元測試結果如預期將拋出例外，確定偽冒static方法失敗。

【5.2】以非static的方法重構

解決問題的方法之一是建立另一個可以被偽冒的非static方法，並讓它去轉呼叫static方法，如以下範例。因為該方法惟一的功能就是將呼叫轉給aStaticMethod()，故名delegate()，如行6：

【範例】/testability/src/main/…/staticmethods/good/StaticMethodDelegate.java

|  |  |
| --- | --- |
| 1 | public class StaticMethodDelegate { |
| 2 | public static void aStaticMethod() { |
| 3 | throw new TestingImpedimentException("Calls static method"); |
| 4 | } |
| 5 | **void delegate() {** |
| 6 | **aStaticMethod();** |
| 7 | **}** |
| 8 | } |

因為無法偽冒static方法，因此改偽冒delegate()方法，並預期「doNothing」，如果再拋出TestingImpedimentException就是重構失敗：

【範例】/testability/src/test/…/staticmethods/good/StaticMethodDelegateTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class StaticMethodDelegateTest { |
| 3 | @Mock |
| 4 | StaticMethodDelegate staticMethodDelegate; |
| 5 | @Test |
| 6 | public void show\_mock\_static\_method() { |
| 7 | // Given |
| 8 | **doNothing()**.when(staticMethodDelegate).delegate(); |
| 9 | try { |
| 10 | // When |
| 11 | staticMethodDelegate.delegate(); |
| 12 | } catch (TestingImpedimentException e) { |
| 13 | // Then |
| 14 | fail(); |
| 15 | } |
| 16 | } |
| 17 | } |

本例如預期通過測試！

【5.3】啟用Mockito偽冒static的支援

Mockito在較新的版本支援偽冒static方法，但預設關閉，要啟動支援必須依照以下步驟：

1. 建立路徑「src/test/resources/mockito-extensions/」。
2. 在資料夾「mockito-extensions」內建立檔案名稱為「org.mockito.plugins.MockMaker」的空白檔案，無附檔名。
3. 在檔案內登打「mock-maker-inline」，如本專案的：

【範例】src/test/resources/mockito-extensions/org.mockito.plugins.MockMaker

|  |  |
| --- | --- |
| 1 | mock-maker-inline |

確定啟用後，以下範例將示範void方法、有return回傳值的方法、有參數輸入且有return回傳值的方法，且都以static宣告：

【範例】/testability/src/main/…/staticmethods/StaticMethodDemo2.java

|  |  |
| --- | --- |
| 1 | public class StaticMethodDemo2 { |
| 2 | public **static** void aVoid() { |
| 3 | throw new RuntimeException(); |
| 4 | } |
| 5 | public **static** String aReturn() { |
| 6 | return "Hello"; |
| 7 | } |
| 8 | public **static** String aParamReturn(String s) { |
| 9 | return "Hello " + s; |
| 10 | } |
| 11 | } |

【小標】單元測試案例1

未偽冒aVoid()時將拋出RuntimeException：

【範例】/testability/src/test/…/staticmethods/StaticMethodDemo2Test.java

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void show\_not\_mock\_static\_void\_method() { |
| 3 | try { |
| 4 | StaticMethodDemo2.**aVoid**(); |
| 5 | fail(); |
| 6 | } catch (Exception e) { |
| 7 | assertTrue(e instanceof RuntimeException); |
| 8 | } |
| 9 | } |

【小標】單元測試案例2

使用Mockito.mockStatic()方法可以建立指定類別的Mock Object。建立Mock Object後，和過去介紹的規則相同，預設void方法都是doNothing()。要注意的是，Mockito.mockStatic()回傳的物件有實作AutoCloseable介面，因此宣告在try-with-resource區塊中：

【範例】/testability/src/main/…/staticmethods/StaticMethodDemo2.java

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void show\_mock\_static\_void\_method\_by\_default() { |
| 3 | // Given |
| 4 | try (MockedStatic<StaticMethodDemo2> mock |
| 5 | = Mockito.mockStatic(StaticMethodDemo2.class)) { |
| 6 | // When |
| 7 | StaticMethodDemo2.**aVoid**(); |
| 8 | // Then |
| 9 | mock.verify(() -> StaticMethodDemo2.aVoid(), times(1)); |
| 10 | } |
| 11 | } |

【小標】單元測試案例3

也可以使用「MockedStatic.when(Verification).then(Answer<?>)」結構的介面Answer<Void>定義static的void方法的偽冒內容，如行6-11。本例依然定義偽冒的內容為什麼都不做，但實際上也可以利用answer()方法定義要執行的程式邏輯：

【範例】/testability/src/main/…/staticmethods/StaticMethodDemo2.java

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void show\_mock\_static\_void\_method\_by\_answer() { |
| 3 | // Given |
| 4 | try (MockedStatic<StaticMethodDemo2> mock |
| 5 | = Mockito.mockStatic(StaticMethodDemo2.class)) { |
| 6 | mock.when(StaticMethodDemo2::aVoid).then(**new Answer<Void>() {** |
| 7 | **@Override** |
| 8 | **public Void answer(InvocationOnMock invocation) throws Throwable {** |
| 9 | **return null;** |
| 10 | **}** |
| 11 | **}** ); |
| 12 | // When |
| 13 | StaticMethodDemo2.aVoid(); |
| 14 | // Then |
| 15 | mock.verify(() -> StaticMethodDemo2.aVoid(), times(1)); |
| 16 | } |
| 17 | } |

【小標】單元測試案例4

承前述範例，改用lambda語法將更為簡潔，如行6：

【範例】/testability/src/main/…/staticmethods/StaticMethodDemo2.java

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void show\_mock\_static\_void\_method\_by\_answer\_lambda() { |
| 3 | // Given |
| 4 | try (MockedStatic<StaticMethodDemo2> mock |
| 5 | = Mockito.mockStatic(StaticMethodDemo2.class)) { |
| 6 | mock.when(StaticMethodDemo2::aVoid).then(**invocation -> null**); |
| 7 | // When |
| 8 | StaticMethodDemo2.aVoid(); |
| 9 | // Then |
| 10 | mock.verify(() -> StaticMethodDemo2.aVoid(), times(1)); |
| 11 | } |
| 12 | } |

【小標】單元測試案例5

類似的語法也可以用在有return回傳值的static方法。使用「MockedStatic.when(Verification).thenReturn(T value)」結構可以在thenReturn()方法中定義偽冒的回傳值，如行7，而且只在try-with-resource區塊中有效：

【範例】/testability/src/main/…/staticmethods/StaticMethodDemo2.java

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void show\_mock\_static\_return\_method() { |
| 3 | assertEquals("Hello", StaticMethodDemo2.aReturn()); |
| 4 | // Given |
| 5 | try (MockedStatic<StaticMethodDemo2> mock |
| 6 | = Mockito.mockStatic(StaticMethodDemo2.class)) { |
| 7 | mock.when(StaticMethodDemo2::aReturn).thenReturn("method is mocked!"); |
| 8 | // When & Then |
| 9 | assertEquals("method is mocked!", StaticMethodDemo2.aReturn()); |
| 10 | } |
| 11 |  |
| 12 | assertEquals("Hello", StaticMethodDemo2.aReturn()); |
| 13 | } |

【小標】單元測試案例6

對於有輸入參數且return回傳值得static方法一樣使用「MockedStatic.when(Verification).thenReturn(T value)」結構，只是介面Verification的實作直接指定傳入的參數值，如行7-8：

【範例】/testability/src/main/…/staticmethods/StaticMethodDemo2.java

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void show\_mock\_static\_return\_method\_with\_params() { |
| 3 | assertEquals("Hello Jim", StaticMethodDemo2.aParamReturn("Jim")); |
| 4 | // Given |
| 5 | try (MockedStatic<StaticMethodDemo2> mock |
| 6 | = Mockito.mockStatic(StaticMethodDemo2.class)) { |
| 7 | mock.when(() -> StaticMethodDemo2.aParamReturn("Jim")) |
| 8 | .thenReturn("mocked!"); |
| 9 | // When & Then |
| 10 | assertEquals("mocked!", StaticMethodDemo2.aParamReturn("Jim")); |
| 11 | } |
| 12 |  |
| 13 | assertEquals("Hello Jim", StaticMethodDemo2.aParamReturn("Jim")); |
| 14 | } |

6. 識別final方法對單元測試的影響

【6.1】測試性不良的設計

當方法宣告為final時在子類別將無法覆寫它，因此也無法建立偽冒方法，這導致單元測試執行時失敗。以下FinalMethodDemo設計了一個含有final方法的類別：

【範例】/testability/src/main/…/finals/bad/method/FinalMethodDemo.java

|  |  |
| --- | --- |
| 1 | public class FinalMethodDemo { |
| 2 | public **final** void aFinalMethod() { |
| 3 | System.out.println("do something"); |
| 4 | } |
| 5 | } |

範例UseFinalMethod則是關連注入了FinalMethodDemo，並在doSomething()方法中呼叫了aFinalMethod()：

【範例】/testability/src/main/…/finals/bad/method/UseFinalMethod.java

|  |  |
| --- | --- |
| 1 | public class UseFinalMethod { |
| 2 | private FinalMethodDemo finalMethod; |
| 3 | public UseFinalMethod(FinalMethodDemo finalMethod) { |
| 4 | this.finalMethod = finalMethod; |
| 5 | } |
| 6 | public void doSomething() { |
| 7 | finalMethod.aFinalMethod(); |
| 8 | } |
| 9 | } |

在單元測試中我們建立FinalMethodDemo的Mock Object並注入到UseFinalMethod中：

【範例】/testability/src/test/…/finals/bad/method/UseFinalMethodTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class UseFinalMethodTest { |
| 3 | @Mock |
| 4 | FinalMethodDemo finalMethod; |
| 5 | @InjectMocks |
| 6 | UseFinalMethod useFinalMethod; |
| 7 | @Test |
| 8 | public void show\_final\_method\_test() throws Exception { |
| 9 | doNothing().when(finalMethod).aFinalMethod(); |
| 10 | useFinalMethod.doSomething(); |
| 11 | } |
| 12 | } |

執行單元測試時，拋出了org.mockito.exceptions.misusing.UnfinishedStubbingException！

Mockito提示了幾項可能原因，本例出錯是因為我們嘗試偽冒final方法：

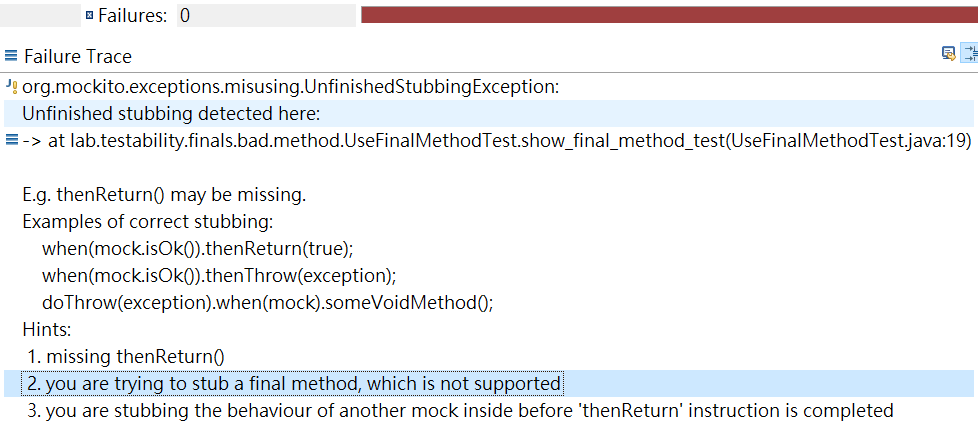


圖3

因此建議不要在final方法中包含測試障礙，Mockito預設不支援偽冒final方法。若是啟用org.mockito.plugins.MockMaker=mock-maker-inline則相同單元測試可通過！

【6.2】以非final的方法重構

重構的作法之一是將final方法的內容提取至新建類別的「非final」的方法，如範例NotFinalMethodDemo的**delegate**()：

【範例】/testability/src/main/…/finals/good/method/NotFinalMethod.java

|  |  |
| --- | --- |
| 1 | public class NotFinalMethod { |
| 2 | public void **delegate**() { |
| 3 | System.out.println("do something"); |
| 4 | } |
| 5 | } |

然後重構FinalMethodDemo的final方法改呼叫NotFinalMethod的**delegate**()，新類別是FinalMethodDemoRefactored，並關連注入NotFinalMethod：

【範例】/testability/src/main/…/good/method/FinalMethodDemoRefactored.java

|  |  |
| --- | --- |
| 1 | public class FinalMethodDemoRefactored { |
| 2 | private NotFinalMethod notFinalMethod; |
| 3 | public FinalMethodDemoRefactored (NotFinalMethod notFinalMethod) { |
| 4 | this.notFinalMethod = notFinalMethod; |
| 5 | } |
| 6 | public **final** void aFinalMethod() { |
| 7 | notFinalMethod.**delegate**(); |
| 8 | } |
| 9 | } |

重構UseFinalMethod為UseFinalMethodRefactored，改呼叫FinalMethodDemoRefactored：

【範例】/testability/src/main/…/good/method/UseFinalMethodRefactored.java

|  |  |
| --- | --- |
| 1 | public class UseFinalMethodRefactored { |
| 2 | private **FinalMethodDemoRefactored** finalMethod; |
| 3 | public UseFinalMethodRefactored(**FinalMethodDemoRefactored** finalMethod) { |
| 4 | this.finalMethod = finalMethod; |
| 5 | } |
| 6 | public void doSomething() { |
| 7 | finalMethod.aFinalMethod(); |
| 8 | } |
| 9 | } |

如此可以對「非final」方法進行偽冒，如行17的doNothing()：

【範例】/testability/src/test/…/good/method/UseFinalMethodRefactoredTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class UseFinalMethodRefactoredTest { |
| 3 |  |
| 4 | @Mock |
| 5 | NotFinalMethod notFinalMethod; |
| 6 |  |
| 7 | FinalMethodDemoRefactored finalMethod; |
| 8 | UseFinalMethodRefactored useFinalMethod; |
| 9 |  |
| 10 | @Before |
| 11 | public void setUp() { |
| 12 | finalMethod = new FinalMethodDemoRefactored(notFinalMethod); |
| 13 | useFinalMethod = new UseFinalMethodRefactored(finalMethod); |
| 14 | } |
| 15 | @Test |
| 16 | public void show\_final\_method\_test() throws Exception { |
| 17 | doNothing().when(notFinalMethod).delegate(); |
| 18 | useFinalMethod.doSomething(); |
| 19 | } |
| 20 | } |

如果接觸不到原始程式碼或其他原因導致無法重構，就只能使用PowerMockito或是啟用Mockito偽冒final方法的支援。

7. 識別final類別對單元測試的影響

【7.1】測試性不良的設計

final類別無法被其它類別所繼承並覆寫，因此在單元測試裡會遇到一些問題。我們先建立一個final類別FinalClassDemo如下：

【範例】/testability/src/main/…/finals/bad/klass/FinalClassDemo.java

|  |  |
| --- | --- |
| 1 | public final class FinalClassDemo { |
| 2 | public void methodInFinalClass() { |
| 3 | // do something |
| 4 | } |
| 5 | } |

再建立範例類別UseFinalClass並關連注入前述的final類別：

【範例】/testability/src/main/java/lab/testability/finals/bad/klass/UseFinalClass.java

|  |  |
| --- | --- |
| 1 | public class UseFinalClass { |
| 2 | private **FinalClassDemo** finalClass; |
| 3 | public UseFinalClass(**FinalClassDemo** finalClass) { |
| 4 | this.finalClass = finalClass; |
| 5 | } |
| 6 | public void doSomething() { |
| 7 | finalClass.methodInFinalClass(); |
| 8 | } |
| 9 | } |

建立單元測試程式碼時建立final類別的Mock Object：

【範例】/testability/src/test/…/finals/bad/klass/UseFinalClassTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class UseFinalClassTest { |
| 3 | **@Mock** |
| 4 | **FinalClassDemo finalClass;** |
| 5 | @InjectMocks |
| 6 | UseFinalClass useFinalClass; |
| 7 | @Test |
| 8 | public void show\_final\_class\_test() throws Exception { |
| 9 | doNothing().when(finalClass).methodInFinalClass(); |
| 10 | useFinalClass.doSomething(); |
| 11 | } |
| 12 | } |

單元測試時拋出MockitoException錯誤訊息：

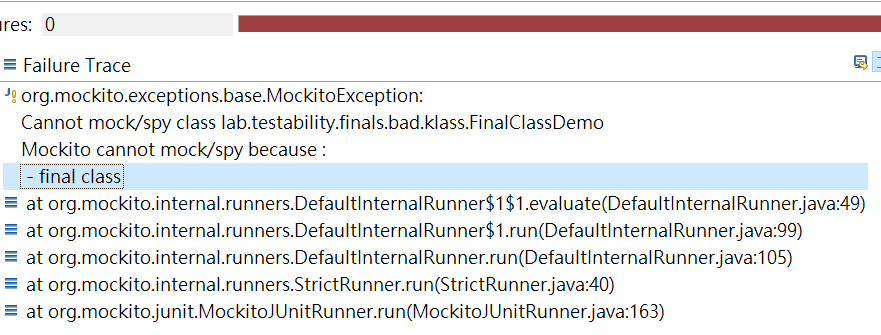


圖4

錯誤訊息顯示「Mockito cannot mock/spy because : - final class」，因為Mockito預設不支援建立final類別的Mock Object！

【7.2】以建立final類別的介面進行重構

final類別對於框架或架構設計很重要，一個安全性考量是可以避免執行中的程式碼被覆寫因而置換。本例重構的解決方案是先建立final類別的interface，如介面IFinalClass：

【範例】/testability/src/main/java/lab/testability/finals/good/klass/IFinalClass.java

|  |  |
| --- | --- |
| 1 | public interface IFinalClass { |
| 2 | public void methodInFinalClass(); |
| 3 | } |

重構原本的final類別使實作介面IFinalClass：

【範例】/testability/src/main/…/finals/good/klass/FinalClassRefactored.java

|  |  |
| --- | --- |
| 1 | public **final** class FinalClassRefactored implements **IFinalClass** { |
| 2 | @Override |
| 3 | public void methodInFinalClass() { |
| 4 | // do something |
| 5 | } |
| 6 | } |

重構UseFinalClass，關連注入時改注入滿足介面IFinalClass的實作而非指定final類別：

【範例】/testability/src/main/…/finals/good/klass/UseFinalClassRefactored.java

|  |  |
| --- | --- |
| 1 | public class UseFinalClassRefactored { |
| 2 | private **IFinalClass** finalClass; |
| 3 | public UseFinalClassRefactored(**IFinalClass** finalClass) { |
| 4 | this.finalClass = finalClass; |
| 5 | } |
| 6 | public void doSomething() { |
| 7 | finalClass.methodInFinalClass(); |
| 8 | } |
| 9 | } |

建立單元測試程式碼時Mock Object型態以interface宣告，而非原本final類別：

【範例】/testability/src/test/…/finals/good/klass/UseFinalClassRefactoredTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class UseFinalClassRefactoredTest { |
| 3 | @Mock |
| 4 | **IFinalClass** finalClass; |
| 5 | @InjectMocks |
| 6 | UseFinalClassRefactored useFinalClass; |
| 7 | @Test |
| 8 | public void show\_final\_class\_test() throws Exception { |
| 9 | doNothing().when(finalClass).methodInFinalClass(); |
| 10 | useFinalClass.doSomething(); |
| 11 | } |
| 12 | } |

如此可以通過測試！

8. 識別使用new呼叫建構子造成的測試問題

【8.1】測試性不良的設計

使用new呼叫建構子是程式裡很常見的片段，但也常常造成單元測試的困擾。以下範例建立一個很普通的類別，在myMethod()方法故意拋出TestingImpedimentException例外：

【範例】/testability/src/main/java/lab/testability/newexpression/NewExpression.java

|  |  |
| --- | --- |
| 1 | public class NewExpression { |
| 2 | public void myMethod() { |
| 3 | throw new TestingImpedimentException ("should not be called!"); |
| 4 | } |
| 5 | } |

在具有相依性的類別UseNewExpression中，我們使用new直接建立NewExpression的物件：

【範例】/testability/src/main/…/newexpression/bad/UseNewExpression.java

|  |  |
| --- | --- |
| 1 | public class UseNewExpression { |
| 2 | public void doSomething() { |
| 3 | NewExpression stuff = **new NewExpression()**; |
| 4 | stuff.myMethod(); |
| 5 | } |
| 6 | } |

單元測試時，因為UseNewExpression直接建立NewExpression，兩者關連性糾結，無法只測試UseNewExpression類別：

【範例】/testability/src/test/…/newexpression/bad/UseNewExpressionTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class UseNewExpressionTest { |
| 3 | UseNewExpression useNewExpression; |
| 4 | @Before |
| 5 | public void setUp() { |
| 6 | useNewExpression = **new UseNewExpression();** |
| 7 | } |
| 8 | @Test |
| 9 | public void show\_new\_expression\_test() throws Exception { |
| 10 | useNewExpression.doSomething(); |
| 11 | } |
| 12 | } |

呼叫UseNewExpression的doSomething()方法時，會直接呼叫真實NewExpression物件的myMethod()方法，因此拋出TestingImpedimentException例外，終止測試：

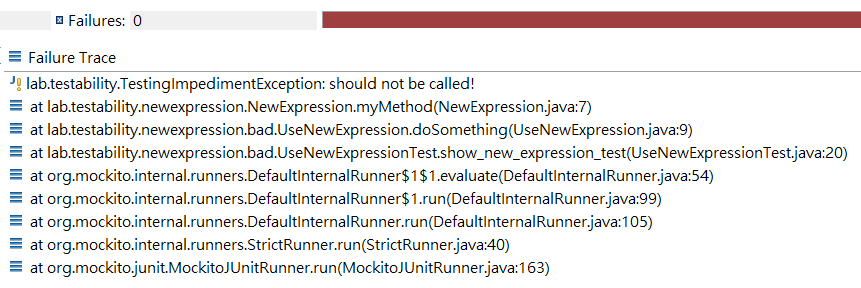


圖5

8.2. 以依賴注入提升測試性的設計

解決方式和過去的幾個重構案例一致，都是關連注入相依物件，因此將UseNewExpression重構為UseNewExpressionRefactored：

【範例】/testability/src/main/…/good/UseNewExpressionRefactored.java

|  |  |
| --- | --- |
| 1 | public class UseNewExpressionRefactored { |
| 2 | private **NewExpression** stuff; |
| 3 | public UseNewExpressionRefactored(**NewExpression** stuff) { |
| 4 | this.stuff = stuff; |
| 5 | } |
| 6 | public void doSomething() { |
| 7 | stuff.myMethod(); |
| 8 | } |
| 9 | } |

如此，撰寫單元測試就可以建立NewExpression的Mock Object，並關連注入至UseNewExpressionRefactored的建構子中：

【範例】/testability/src/test/…/good/UseNewExpressionRefactoredTest.java

|  |  |
| --- | --- |
| 1 | @RunWith(MockitoJUnitRunner.class) |
| 2 | public class UseNewExpressionRefactoredTest { |
| 3 | **@Mock** |
| 4 | **NewExpression** newExpression; |
| 5 | **@InjectMocks** |
| 6 | **UseNewExpressionRefactored** useNewExpression; |
| 7 | @Test |
| 8 | public void show\_new\_expression\_test() throws Exception { |
| 9 | useNewExpression.doSomething(); |
| 10 | } |
| 11 | } |

如此分離NewExpression並完成UseNewExpressionRefactored的單元測試！

物件導向程式設計有一句名言「Program to an interface, not an implementation.」程式開發時應該盡量以抽象型態或父類別來建立相依性，直接以new產生子類別或實作的物件是比較不建議的作法，比較好的作法是以「關連注入」來決定執行時期相依的子類別或實作，框架如Spring就是以這樣的基本設計貫串整個架構！

以我們過去的諸多測試障礙的重構方法可知，「關連注入」同時也是讓我們比較好撰寫單元測試的好習慣。

9. 識別使用static變數和程式碼區塊造成的測試問題

【9.1】測試性不良的設計

static變數的初始化和程式碼區塊都是在類別載入時執行；因為無法覆寫，因此也無法建立偽冒方法，我們將使用以下範例說明。

先建立StaticBlockDependency類別：

【範例】/testability/src/main/…/staticblock/StaticBlockDependency.java

|  |  |
| --- | --- |
| 1 | public class StaticBlockDependency { |
| 2 | private Date loadTime; |
| 3 | public **StaticBlockDependency**() { |
| 4 | **throw new TestingImpedimentException("Can't be loaded!!");** |
| 5 | } |
| 6 | public Date getLoadTime() { |
| 7 | return loadTime; |
| 8 | } |
| 9 | public void setLoadTime(Date loadTime) { |
| 10 | this.loadTime = loadTime; |
| 11 | } |
| 12 | } |

該類別被StaticBlockOwner的static變數所參照，並在static程式碼區塊中初始化：

【範例】/testability/src/main/…/staticblock/bad/StaticBlockOwner.java

|  |  |
| --- | --- |
| 1 | public class StaticBlockOwner { |
| 2 | private static **StaticBlockDependency** dependency; |
| 3 | **static {** |
| 4 | dependency = **new StaticBlockDependency();** |
| 5 | dependency.setLoadTime(new Date()); |
| 6 | **}** |
| 7 | public boolean isLoadingTimeBefore(Date base) { |
| 8 | return dependency.getLoadTime().before(base); |
| 9 | } |
| 10 | } |

我們將以StaticBlockOwnerTest類別對StaticBlockOwner進行單元測試，如以下範例。因為StaticBlockOwner相依於StaticBlockDependency，因此測試時不能產生StaticBlockDependency的真實物件，只能建立Mock Object。所以我們設計StaticBlockDependency時故意在建構子裡拋出例外物件，只要建構子被呼叫執行，就表示測試失敗！

【範例】/testability/src/test/…/staticblock/bad/StaticBlockOwnerTest.java

|  |  |
| --- | --- |
| 1 | public class StaticBlockOwnerTest { |
| 2 | StaticBlockOwner staticBlockDemo; |
| 3 | @Before |
| 4 | public void setUp() { |
| 5 | staticBlockDemo = **new StaticBlockOwner();** |
| 6 | } |
| 7 | @Test |
| 8 | public void testLoadingTime() throws ParseException { |
| 9 | Date base = new SimpleDateFormat("yyyy-MM-dd").parse("2020-05-29"); |
| 10 | assertTrue(staticBlockDemo.isLoadingTimeBefore(base)); |
| 11 | } |
| 12 | } |

這個測試如預期會失敗，主要是因為我們無法在建立StaticBlockOwner物件時避免建立StaticBlockDependency真實物件，更無法以Mock Object取代：

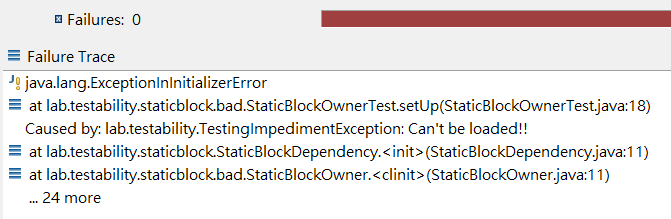


圖6

在static程式碼區塊中建立物件的關連性會成測試程式碼無法撰寫，還是應該以依賴注入的方式建立物件關連。若只能使用static程式碼區塊，就必須使用PowerMock抑止static初始化區塊後再置入StaticBlockDependency的Mock Object，如下：

【範例】/testability/src/test/…/bad/StaticBlockOwnerTestByPowerMock.java

|  |  |
| --- | --- |
| 1 | @RunWith(PowerMockRunner.class) |
| 2 | @SuppressStaticInitializationFor("lab.testability.staticblock.bad.StaticBlockOwner") |
| 3 | public class StaticBlockOwnerTestByPowerMock { |
| 4 | @Test |
| 5 | public void Should\_return\_true\_When\_given\_earlier\_than\_loading\_time() throws ParseException { |
| 6 | // Given |
| 7 | Date loadTime = new SimpleDateFormat("yyyy-MM-dd").parse("2019-05-29"); |
| 8 | StaticBlockDependency dependency = mock(StaticBlockDependency.class); |
| 9 | when(dependency.getLoadTime()).thenReturn(loadTime); |
| 10 |  |
| 11 | Whitebox.setInternalState(StaticBlockOwner.class, dependency); |
| 12 |  |
| 13 | // When |
| 14 | StaticBlockOwner instance = new StaticBlockOwner(); |
| 15 | // Then |
| 16 | Date base = new SimpleDateFormat("yyyy-MM-dd").parse("2020-05-29"); |
| 17 | assertTrue(instance.isLoadingTimeBefore(base)); |
| 18 | } |
| 19 | } |