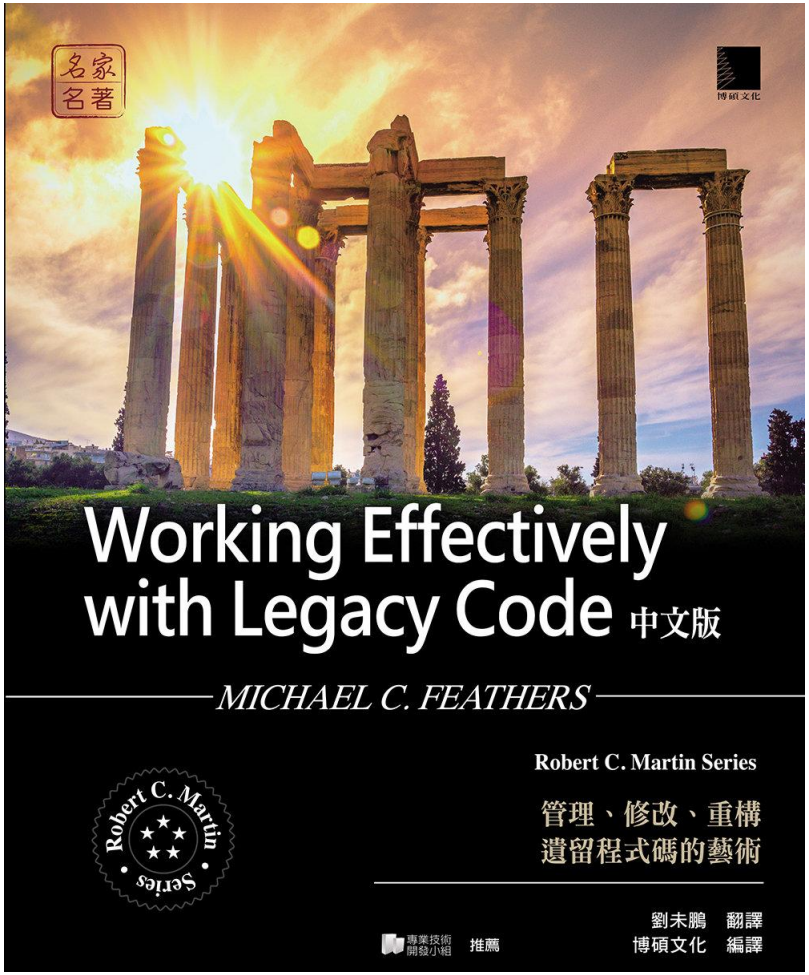
Unit Testing
單元測試

Shin-Jie Lee (李信杰)
Associate Professor
Department of CSIE
National Cheng Kung University

單元測試是什麼? (Michael Feature)

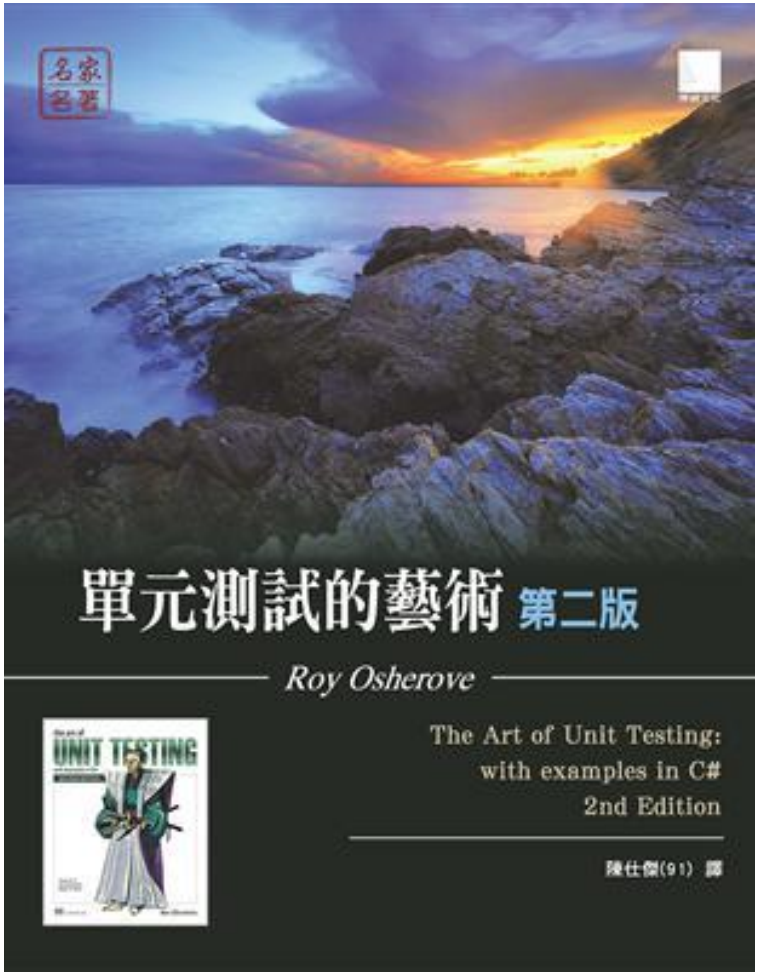
- 單元測試執行快，能幫助定位問題所在
  - 超過0.1秒算是一個慢的單元測試
- 以下這些測試被Michael Feathers認定不是單元測試
  - 與資料庫有互動
  - 進行了網路通訊
  - 接觸到檔案系統
  - 需要你對環境做特定的準備(如編輯設定檔案)才能夠執行
- 但不是說這些測試就是不好的，編寫它們仍有價值，但建議與單元測試分開，以利單元測試可快速被執行完



單元測試是什麼? (Roy Osherove)

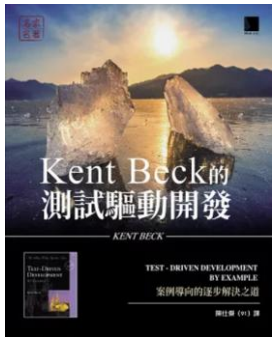
- 一個單元測試是一段自動化的程式碼，這段程式會呼叫被測試的工作單元，之後對這個單元的單一最終結果的某些假設或期望進行驗證

- 一個單元測試範圍，可以小到一個方法(Method)，大到實現某個功能的多個類別與函數

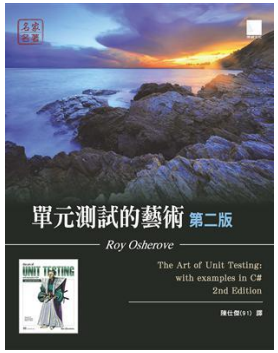


何時撰寫測試案例

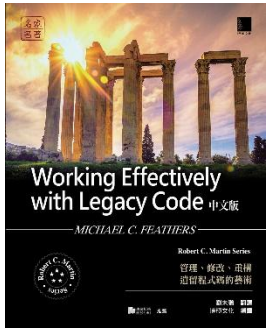
- 程式開發前
  - For TDD (Test-Driven Development)



- 程式開發後
  - For Regression Testing

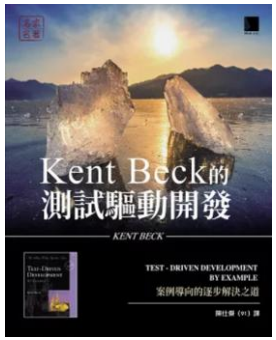


- 程式變成Legacy Code時
  - For Refactoring

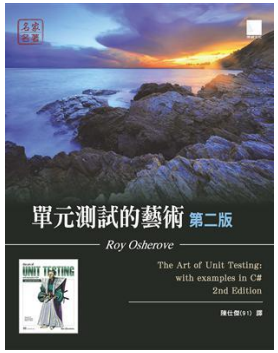


此教材聚焦

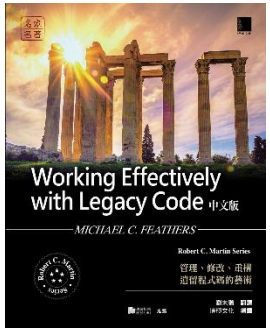
- 程式開發前
  - For TDD (Test-Driven Development)



- 程式開發後
  - For Regression Testing



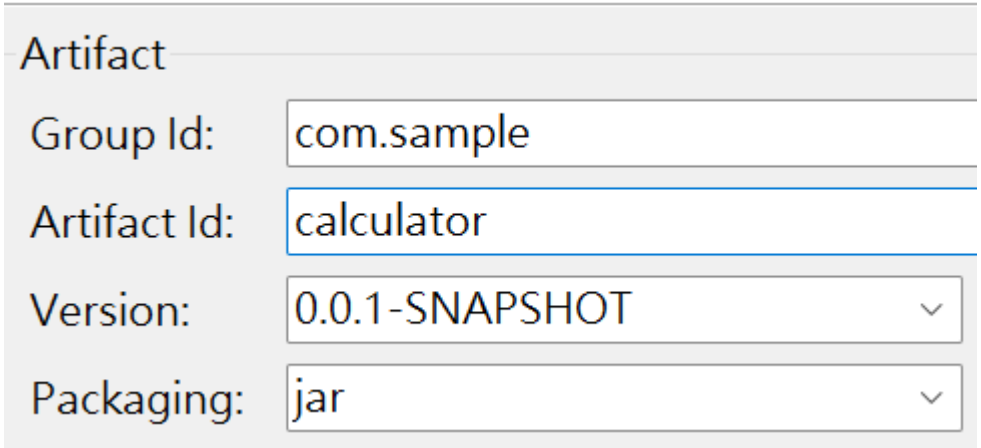
- 程式變成Legacy Code時
  - For Refactoring



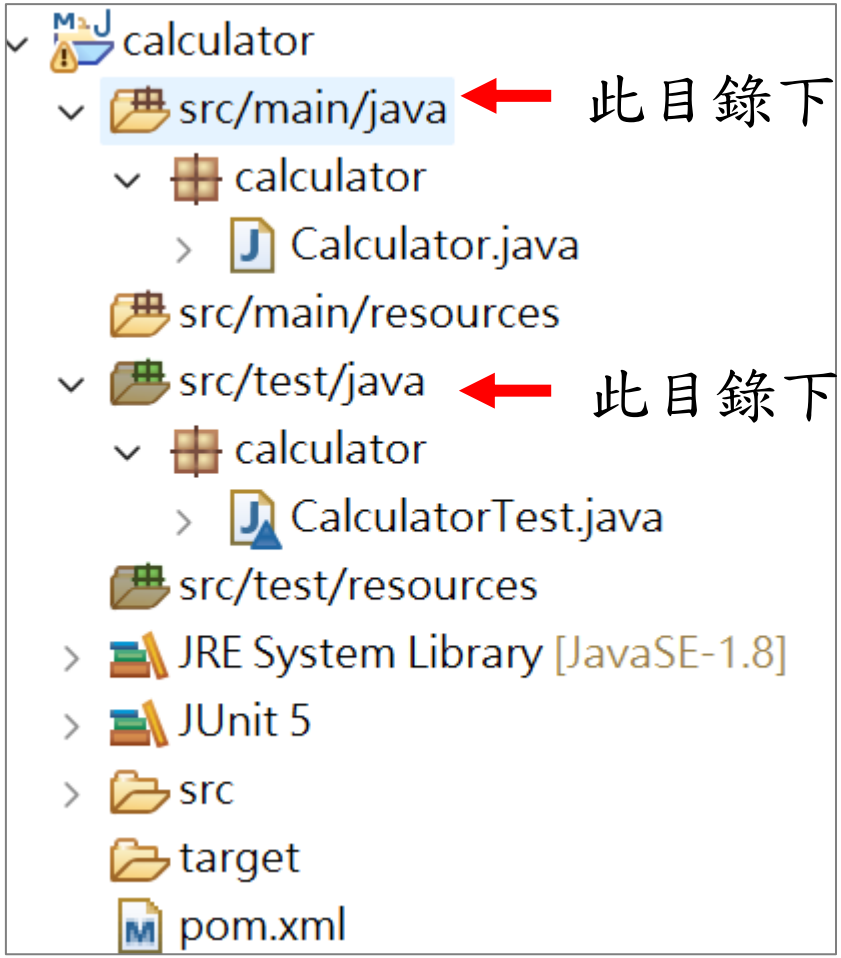
使用Eclipse+Maven+Junit
撰寫單元測試

在Eclipse中建立一個Maven Project

- File ➔ New ➔ Maven Project ➔ 勾選Create a simple project ➔ Next
- 輸入
  - Group Id (組織/公司名稱)
  - 輸入Artifact Id (Project名稱)



Maven Project目錄結構



此目錄下將存放主要程式碼

此目錄下將存放單元測試程式碼

單元測試類型

1. 驗證回傳值
2. 驗證系統狀態

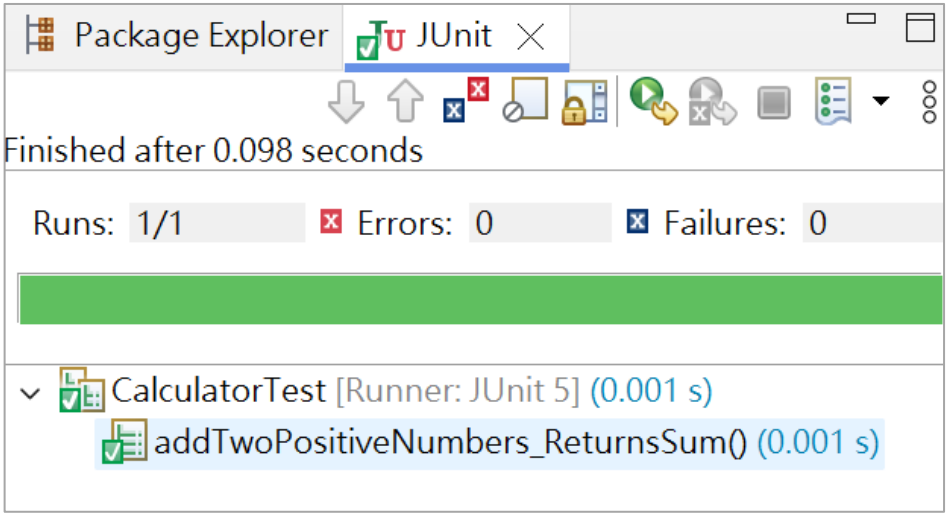
驗證回傳值

受測Method有回傳值

```
public class Calculator {
  public int addTwoNumber(int i, int j) {
    return i + j;
  }
}
```

驗證回傳值

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {
  @Test
  void addTwoPositiveNumbers_ReturnsSum() {
    Calculator calculator = new Calculator();
    int result = calculator.addTwoNumber(3, 5);
    assertEquals(8, result);
  }
}
```



Tip (單元測試的命名)

- 測試名稱可以很長
- 可包含
  - 被測試方法名稱
  - 測試情境
  - 預期行為
- 例如
  - addTwoPositiveNumbers_RetrunsSum()

Lab

- 利用AI工具生成更多的測試案例

Copilot4Eclipse Chat

Unit Testing the Calculator Class

/tests

Copilot4Eclipse

```
package calculator;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @Test
    void addTwoPositiveNumbers_ReturnsSum() {
        Calculator calculator = new Calculator();
        int result = calculator.addTwoNumber(3, 5);
        assertEquals(8, result);
    }

    @Test
    void addPositiveAndNegativeNumber_ReturnsSum() {
        Calculator calculator = new Calculator();
        int result = calculator.addTwoNumber(3, -5);
        assertEquals(-2, result);
    }
```

Ask Copilot a question or type '/' for commands

驗證系統狀態

受測Method沒有回傳值

- 例如，cumulate method沒有回傳值，可嘗試驗證系統狀態sum是否正確

```
public class Calculator {
  private int sum=0;

  public void cumulate(int number) {
    sum+=number;
  }
  public int getSum() {
    return sum;
  }

  public int addTwoNumber(int i, int j) {
    return i + j;
  }
}
```

驗證系統狀態

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

  @Test
  void cumulateMultiplePositiveNumbers_IncreasesSum() {
    Calculator calculator = new Calculator();
    calculator.cumulate(5);
    calculator.cumulate(10);
    assertEquals(15, calculator.getSum());
  }

  @Test
  void addTwoPositiveNumbers_ReturnsSum() {
    Calculator calculator = new Calculator();
    int result = calculator.addTwoNumber(3, 5);
    assertEquals(8, result);
  }
}
```

Lab

- 利用AI工具生成更多的測試案例

Refactor untestable code to testable code

透過Stub解決依賴問題

Untestable Code

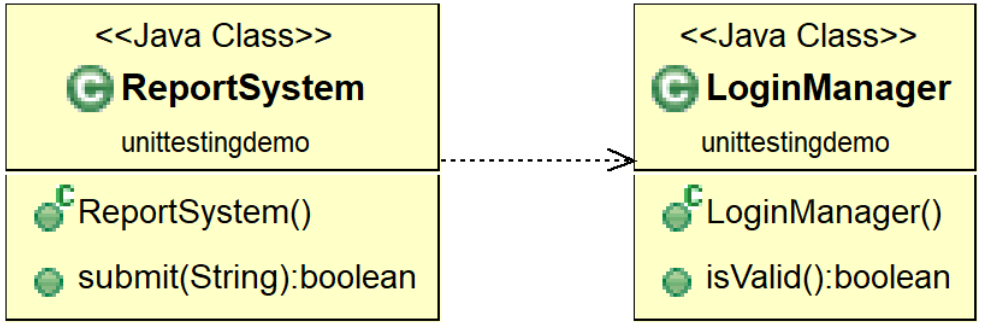
- 當被測試的物件依賴於另一個無法控制(或尚未實作)的物件時，要造成無法進行單元測試
  - 例如依賴於一個Web Service、系統時間、執行緒、資料庫、本地檔案等

- 此時可利用Stub概念來重構解耦 (Refactor untestable code to testable code)

例如面對Legacy Code - ReportSystem依賴LoginManager

```java
public class ReportSystem {

  public boolean submit(String report) {
    LoginManager loginmgr = new
                          LoginManager();
    if (loginmgr.isValid()) {
      // submit the report
      return true;
    }
    return false;
  }
}
```

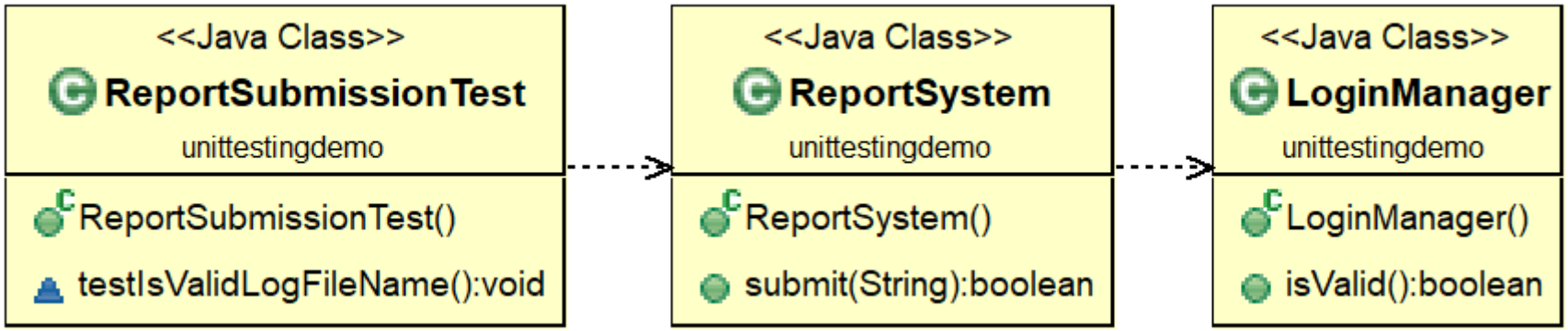
```java
public class LoginManager {

  public boolean isValid() {
    //validate the login here
    return false;
  }
}
```



問題 - Submit Report的單元測試案例因沒有先登入而失敗

- 記住，在這裡要執行的是單元測試，不是端對端測試，若每次執行單元測試都要登入將會降低執行效率

```
import org.junit.jupiter.api.Test;
public class ReportSubmissionTest {
  @Test
  void testIsValidLogFileName() {
    ReportSystem reportsm = new
                    ReportSystem();
    boolean result =
      reportsm.submit("my report");
    assert (result);
  }
}
```



解決方法 - 重構以增加可測性 (Testable)

Steps:

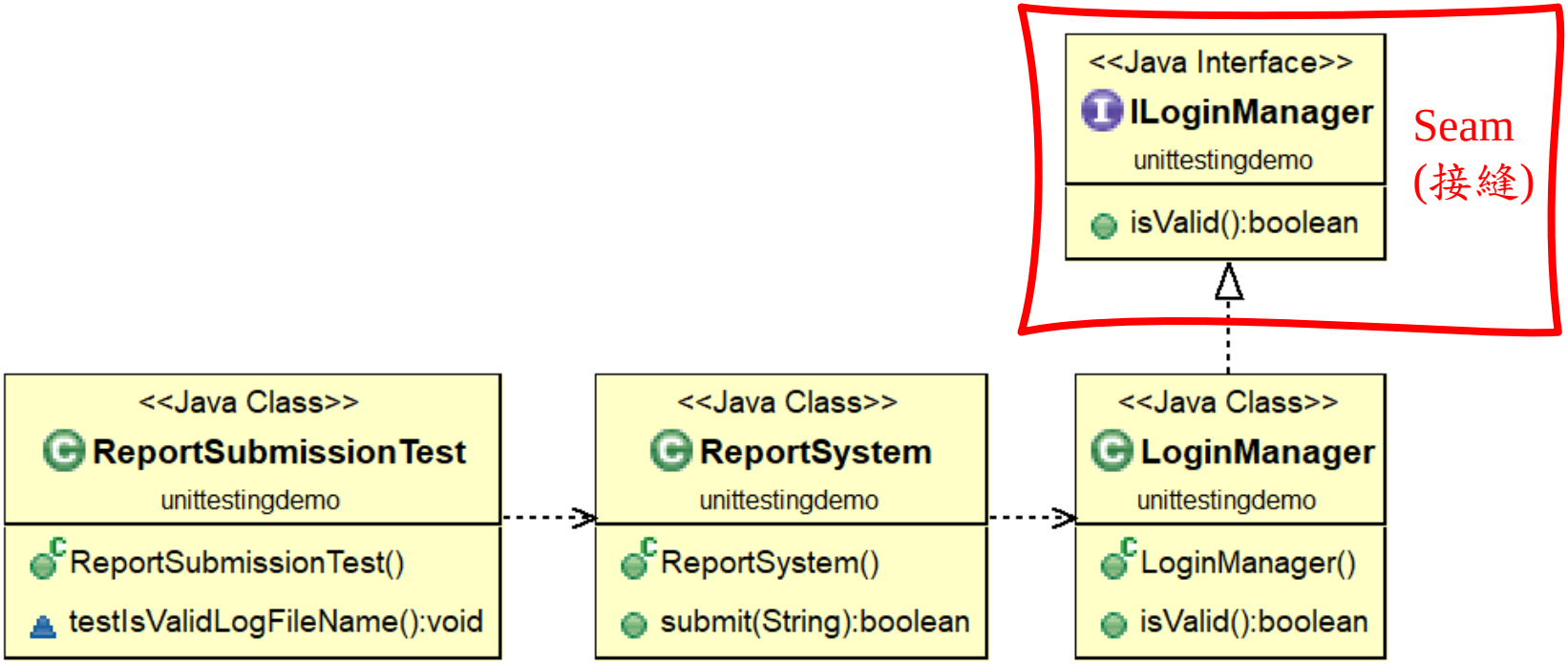
1. Extract Interface as Seam
2. Create Stub Class
3. Program to an interface, not an implementation
4. Dependency Injection

Extract Interface as Seam

- 抽出LoginManager的介面當作Seam(接縫)

```
public interface ILoginManager {
  public abstract boolean isValid();
}
```

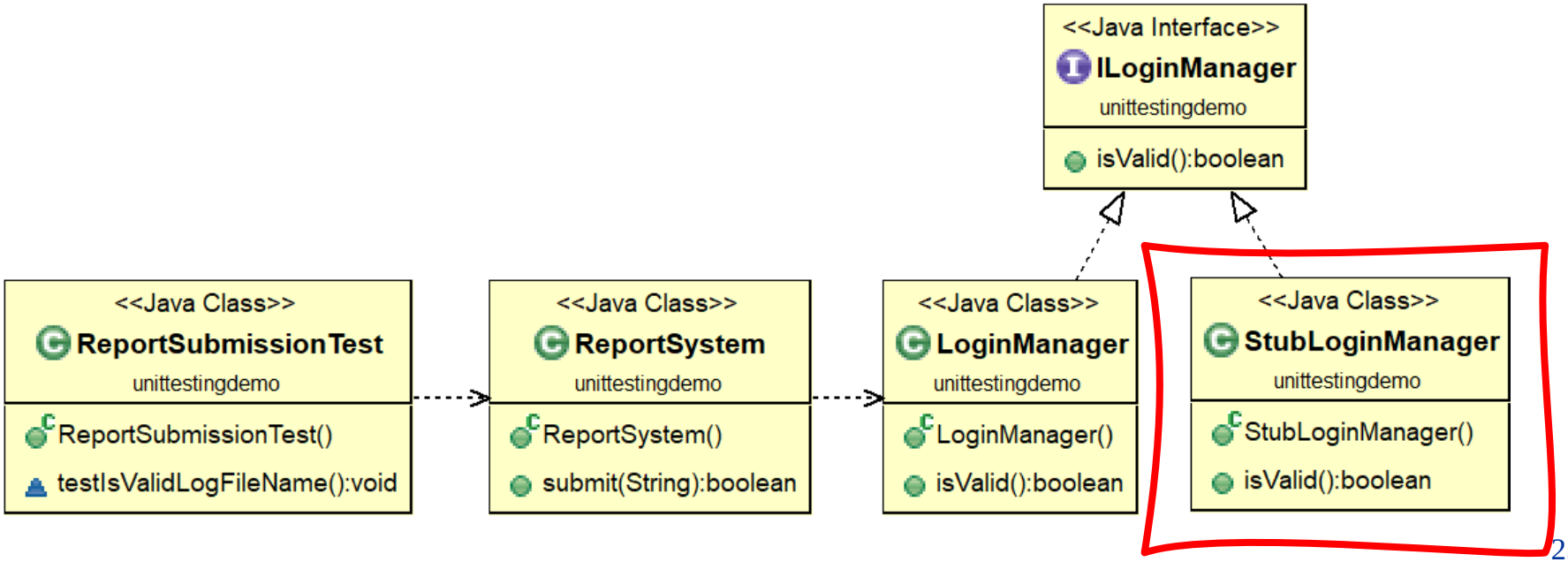
```
public class LoginManager implements ILoginManager{
  //…
}
```



Create Stub Class (假物件)

- 建立實作ILoginManager介面的Stub Class，模擬永遠為Login狀態

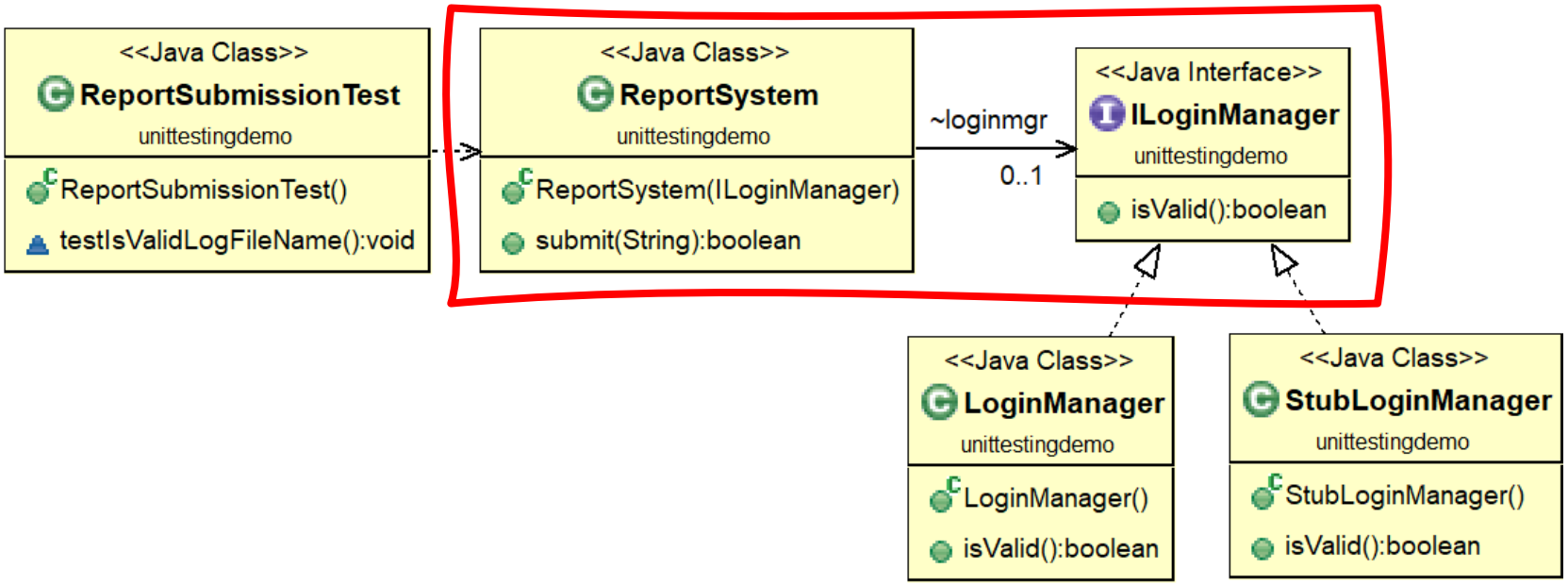
```
public class StubLoginManager implements ILoginManager{
  @Override
  public boolean isValid() {
    return true; // always return true
  }
}
```



Program to an interface, not an implementation

```
public class ReportSystem {
  ILoginManager loginmgr;
  public ReportSystem(ILoginManager loginmgr) {
    this.loginmgr = loginmgr;
  }
  public boolean submit(String report) {
    LoginManager loginmgr = new LoginManager();
    if (loginmgr.isValid()) {
// …
```

讓ReportSystem依賴Interface
而非Concrete Class

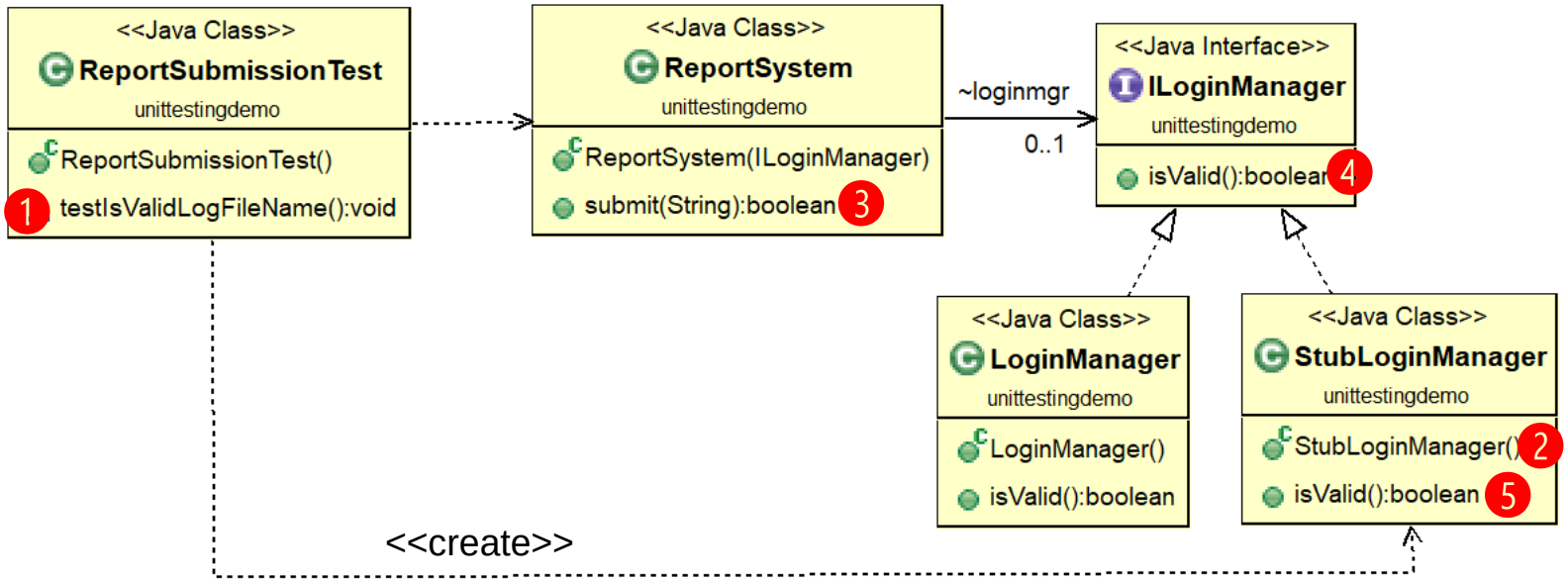


Dependency Injection

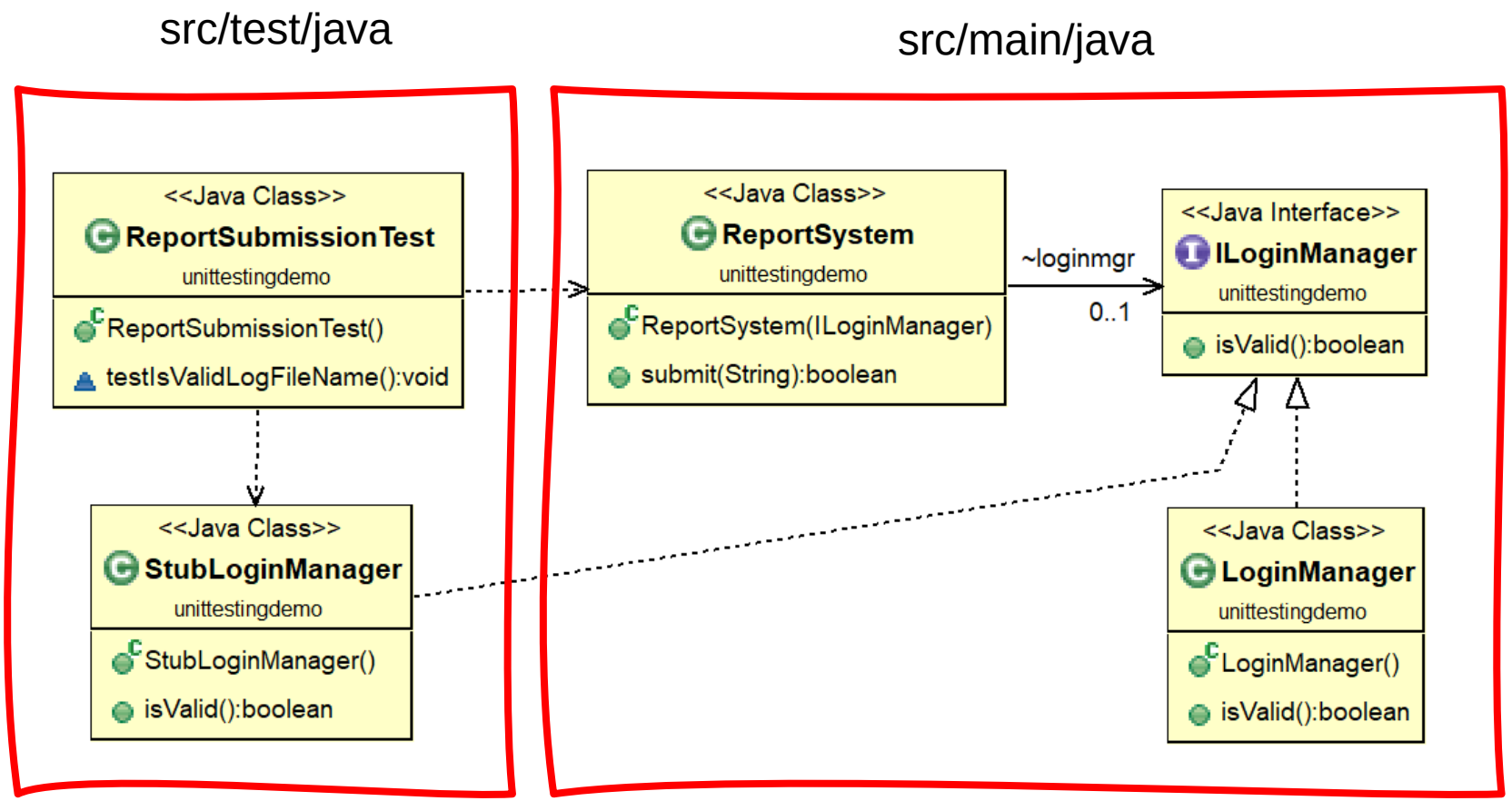
```
import org.junit.jupiter.api.Test;
public class ReportSubmissionTest {
  @Test
  void testIsValidLogFileName() {
    ReportSystem reportsm = new ReportSystem(new StubLoginManager());
    boolean result = reportsm.submit("my report");
    assert (result);
  }
}
```

測試開始先注入一個Stub Object
(稱之為Constructor Injection)

submit執行時，ReportSystem就會動態
依賴此Stub Object，模擬已登入狀態

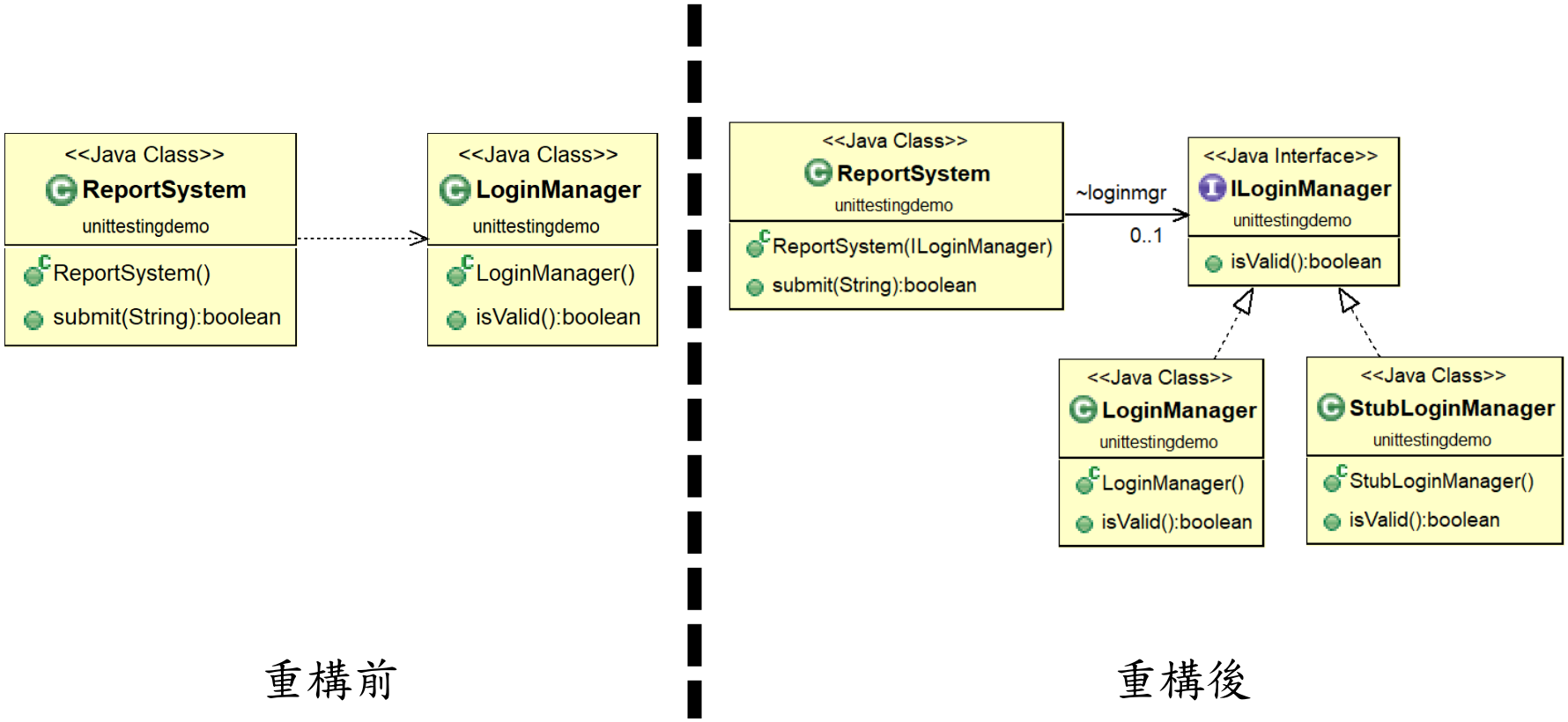


建議將Stub放置於測試案例目錄



Refactoring to Testable Code

- 重構後的程式碼結構「可測試性」較高(More testable)



Tip (From Stub to Mock)$_1$

- 若將Stub擴增為可記錄受測物件與它的互動歷史(Interaction)

```
public class MockLoginManager implements ILoginManager{

  private boolean called = false;

  @Override
  public boolean isValid() {
    called = true;
    return true; // always return true
  }
  public boolean wasCalled() {
    return called;
  }
}
```

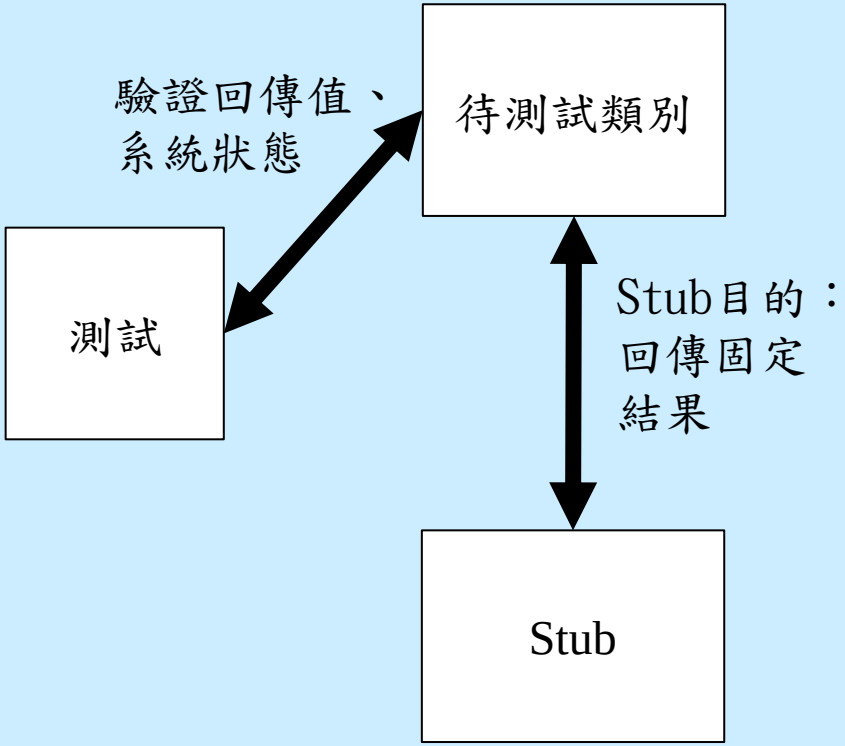
Tip (From Stub to Mock)$_2$

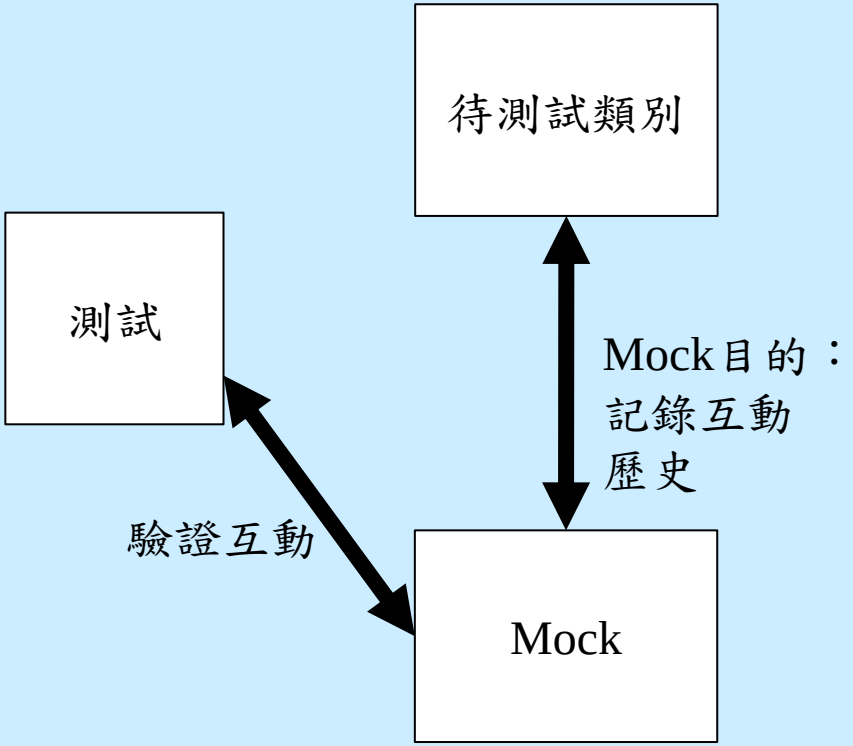
- 接著在測試案例中加入驗證這些互動歷史是否符合預期，則此Stub的概念即成為Mock概念

```
import org.junit.jupiter.api.Test;

public class ReportSubmissionTest {

  @Test
  void testSubmitWithMockLoginManager() {
    MockLoginManager mocklogin = new MockLoginManager();
    ReportSystem reportsm = new ReportSystem(mocklogin);
    boolean result = reportsm.submit("my report");
    assert (result);
    assert (mocklogin.wasCalled());
  }
}
```

Tip (Stub vs. Mock)

- 雖然Stub與Mock目的不同，但手動重構實作手法基本上一樣
- 有時Stub與Mock皆稱為假物件





Tip (使用Mock驗證互動)

- 因此，單元測試類型除了
  - 驗證回傳值
  - 驗證系統狀態

- 運用Mock即可增加一種測試類型
  - **驗證互動**

Lab

- 除了使用上述Dependency Injection來製作出Stub與Mock外，可否使用Subclassing機制來實作呢?
- 什麼情況下較不適用Subclassing?