JUnit A Cook's Tour

참고 :이 기사는 JUnit 3.8.x를 기반으로 합니다.

1. 소개

이전 기사 (Test Infected : Programmers Love Writing Tests, Java Report, July 1998, Volume 3, Number 7 참조)에서
우리는 간단한 프레임워크를 사용해 반복 가능한 테스트를 작성하는 방법을 설명했습니다.
이번 기사에서는 내부를 들여다보고 프레임워크 자체가 어떻게 구성되는지 보여줄 것입니다.

우리는 JUnit 프레임워크를 주의 깊게 연구하고 어떻게 구성했는지를 반영했습니다.
우리는 다양한 수준에서 수업을 찾았습니다.
이 기사에서 우리는 그것들을 한꺼번에 전달하려고 시도할 것입니다.
노력하겠지만, 적어도 검증된 가치를 가진 소프트웨어의 디자인과 구성을 보여주기 위해 시도할 것입니다.

우리는 프레임워크의 목표에 대한 토론으로 시작합니다.
목표는 프레임워크 자체를 표시하는 동안 많은 작은 세부 사항으로 다시 나타납니다.
다음으로 프레임 워크의 설계 및 구현에 관해 설명합니다.
디자인은 문해력 있는 프로그램으로 구현되는 패턴 (놀람, 놀라움)의 관점에서 설명될 것입니다.
프레임워크 개발에 대한 몇 가지 선택 사항으로 결론을 내릴 것입니다.

2. 목표

JUnit의 목표는 무엇입니까?

먼저, 우리는 개발의 가정으로 돌아가야 합니다.
프로그램 기능에 자동화된 테스트가 없는 경우 작동하지 않는다고 가정합니다.
이것은 개발자가 프로그램 기능이 작동한다고 확신하면
지금은 영원히 작동한다는 일반적인 가정보다 훨씬 안전해 보입니다.

이러한 관점에서 개발자는 코드를 작성하고 디버그할 때 테스트가 없으면 완료되지 않으며
프로그램이 작동함을 보여주는 테스트도 함께 작성해야 합니다.
하지만 모두가 너무 바쁘고, 할 일이 너무 많고, 테스트할 시간이 없습니다.
이미 작성할 코드가 너무 많습니다.
테스트 코드도 어떻게 작성해야 합니까?
대답해 주세요, PM 님!!

따라서 가장 중요한 목표는 개발자가 실제로 테스트를 작성할 것이라는 희망이 있는 프레임워크를 만드는 것입니다.
프레임워크는 익숙한 도구를 사용해야 하므로 새로 배울 내용이 거의 없습니다.
새로운 테스트를 작성하는데 절대적으로 필요한 것 이상의 더 많은 작업이 필요하지 않습니다.
중복된 노력을 제거해야 합니다.

이 모든 테스트가 수행해야 한다면 디버거에서 표현식을 작성하는 것만으로 완료됩니다.
그러나 이것은 테스트에 충분하지 않습니다.
당신의 프로그램이 지금 작동한다고 말해도 도움이 되지 않습니다.
통합 후 1분 후에 당신의 프로그램이 작동할 것이라는 보장도 없고
당신이 오래 자리를 비웠을 때 당신의 프로그램이 5년 후에도 계속 작동할 것이라는 보장도 없기 때문입니다.

따라서 테스트의 두 번째 목표는 시간이 지남에 따라 가치를 유지하는 테스트를 만드는 것입니다.
원래 작성자가 아닌 다른 사람이 테스트를 실행하고 결과를 해석할 수 있어야 합니다.
다양한 개발자들의 테스트를 결합하여 간섭에 대한 두려움 없이 함께 실행할 수 있어야 합니다.

마지막으로 기존 테스트를 활용하여 새 테스트를 생성할 수 있어야 합니다.
Setup 또는 Fixture를 만드는 것은 비용이 많이 들고 프레임워크는 Fixture를 재사용하여 다른 테스트를 실행할 수 있어야 합니다.
이게 전부입니다.

3. JUnit의 디자인

JUnit의 디자인은 ("Patterns Generate Architectures", Kent Beck 및 Ralph Johnson, ECOOP 94 참조)에서 처음 사용된 스타일로 제공됩니다.
아이디어는 시스템의 아키텍처를 가질 때까지 아무것도 시작하지 않고 패턴을 차례로 적용하여 시스템의 설계를 설명하는 것입니다.
우리는 해결해야 할 아키텍처 문제를 제시하고 이를 해결하는 패턴을 요약한 다음 패턴이 JUnit에 어떻게 적용되었는지 보여줄 것입니다.

3.1 시작하기 - TestCase

먼저 기본 개념인 TestCase를 나타내는 개체를 만들어야 합니다.
개발자는 종종 테스트 사례를 염두에 두고 있지만 다양한 방법으로 이를 실현합니다.

- 구문 출력
- 디버거 표현식
- 테스트 스크립트들

테스트를 쉽게 조작하려면 객체를 만들어야 합니다.
이것은 개발자의 마음속에 내재한 테스트를 거쳐 구체적으로 만들어
시간이 지남에 따라 가치를 유지하는 테스트를 만드는 우리의 목표를 뒷받침합니다.
동시에 개체 개발자는 객체를 사용하여 개발하는 데 익숙하므로
테스트를 객체로 만들기로 한 결정은 테스트 작성을 더 매력적으로 (또는 최소한 덜 강요하는) 만드는 우리의 목표를 뒷받침합니다.
Command 패턴(Gamma, E., et al. Design Patterns : Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995 참조)은 우리의 요구에 아주 잘 맞습니다.
의도에서 인용하면 "요청을 객체로 캡슐화하여 요청을 큐에 넣거나 기록을 남길 수 있습니다." 명령은 작업을 위한 객체를 생성하고 "execute" 메서드를 제공하도록 지시합니다.
다음은 TestCase의 클래스 정의에 대한 코드입니다.

```
public abstract class TestCase implements Test
{
    ...
}
```

이 클래스는 상속을 통해 재사용 될 것으로 예상하기 때문에 "public abstract"로 선언합니다.
지금은 Test 인터페이스를 구현한다는 사실을 무시하십시오.
현재 설계의 목적에 따라 TestCase를 하나의 class로 생각할 수 있습니다.
모든 TestCase는 이름으로 생성되므로 테스트가 실패하면 실패한 테스트를 식별할 수 있습니다.

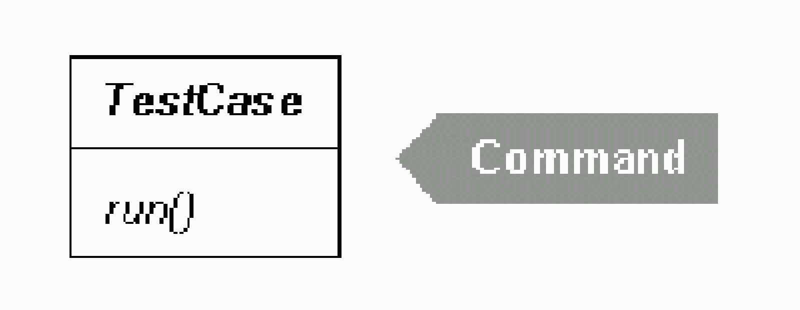
```
public abstract class TestCase implements Test
{
    private final String fName;

    public TestCase (String name) {
        fName = name;
    }

    public abstract void run ();
    ...
}
```

JUnit의 발전을 설명하기 위해 아키텍처의 스냅 샷을 보여주는 다이어그램을 사용합니다.
우리가 사용하는 표기법은 간단합니다.
연관된 패턴을 포함하는 음영 처리된 상자로 클래스에 주석을 답니다.
패턴에서 클래스의 역할이 분명하면 패턴 이름만 표시됩니다.
역할이 명확하지 않으면 음영 처리된 상자가 이 클래스에 해당하는 참가자의 이름으로 확대됩니다.
이 표기법은 다이어그램의 혼란을 최소화하기 위해
(Gamma, E., Applying Design Patterns in Java, in Java Gems, SIGS Reference Library, 1997 참조)에서 처음으로 표시되었습니다.

그림 1은 TestCase에 적용된 이 표기법을 보여줍니다.
단일 클래스를 다루고 있고 모호성이 없기 때문에 패턴 이름만 표시됩니다.



3.2 Blanks to fill in - run()

다음으로 해결해야 할 문제는 개발자에게 fixture 코드와 테스트 코드를 넣을 수 있는 편리한 "장소"를 제공하는 것입니다.
TestCase를 abstract로 선언하면 개발자가 subclassing을 통해 TestCase를 재사용할 것으로 예상됩니다.
그러나 우리가 할 수 있는 모든 것이 하나의 변수와 동작이 없는 super class를 제공하는 것이라면 첫 번째 목표를 충족시키기 위해 많은 일을 하지 않아서 테스트를 더 쉽게 작성할 수 있습니다.

다행히 모든 테스트에는 공통 구조가 있습니다.
test fixture를 설정하고 fixture에 대해 일부 코드를 실행하고 일부 결과를 확인한 다음 fixture를 정리합니다.
이는 각 테스트가 새로운 fixture로 실행되고 한 테스트의 결과가 다른 테스트의 결과에 영향을 미칠 수 없음을 의미합니다.
이것은 테스트의 가치를 극대화하려는 목표를 지원합니다.

템플릿 방법은 우리의 문제를 아주 잘 해결합니다.
"작업에서 알고리즘의 골격을 정의하고 일부 단계를 하위 클래스로 연기합니다.
 템플릿 방법을 사용하면 하위 클래스가 알고리즘의 구조를 변경하지 않고도 알고리즘의 특정 단계를 재정의할 수 있습니다." 이것은 정확히 맞습니다.
우리는 개발자가 fixture(setup and teardown) 코드를 작성하는 방법과 테스트 코드를 작성하는 방법을 별도로 고려할 수 있기를 바랍니다.
그러나 이 시퀀스의 실행은 fixture 코드가 작성되는 방법이나 테스트 코드가 작성되는 방법과 관계없이 모든 테스트에 대해 동일하게 유지됩니다.

다음은 템플릿 방법입니다.

```
public void run()
{
    setUp();
    runTest();
    tearDown();
}
```

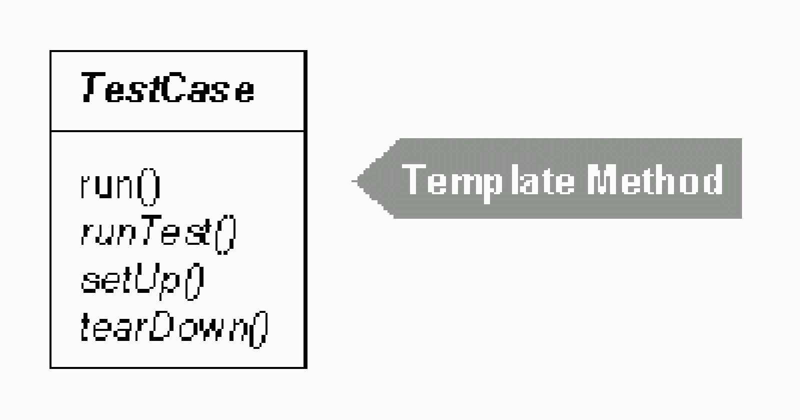
이러한 메서드의 기본 구현은 아무 작업도 수행하지 않습니다.

```
protected void runTest()
{
    ...
}
```

```
protected void setUp()
{
    ...
}
```

```
protected void tearDown()
{
    ...
}
```

setUp 및 tearDown은 재정의될 예정이지만 프레임워크에서 호출되므로 보호된 것으로 선언합니다.
투어의 두 번째 스냅샷은 그림 2에 나와 있습니다.



3.3 Reporting results - TestResult

TestCase가 forest에서 실행되는 경우 누군가 결과에 관심이 있습니까? 물론입니다.
테스트를 실행하여 실행되는지 확인합니다.
테스트가 실행된 후 작동한 작업과 실행되지 않은 작업에 대한 요약을 기록해야 합니다.

테스트가 성공하거나 실패할 확률이 같거나 테스트를 한 번만 실행했다면 TestCase 객체에 flag를 설정하고
테스트가 완료되면 flag를 살펴볼 수 있습니다.
그러나 테스트는 매우 비대칭적이며 일반적으로 작동합니다.
따라서 실패와 성공에 대한 매우 압축된 요약만 기록하기를 원합니다.

Smalltalk 모범 사례 패턴 (Beck, K. Smalltalk 모범 사례 패턴, Prentice Hall, 1996 참조)에는 적용 가능한 패턴이 있습니다.
Collecting Parameter라고 합니다.
여러 메서드에 대한 결과를 수집해야 하는 경우 메서드에 매개 변수를 추가하고 결과를 수집할 객체를 전달해야 합니다.
테스트 실행 결과를 수집하기 위해 새로운 객체인 TestResult를 만듭니다.

```
public class TestResult extends Object
{
    protected int fRunTests;
    public TestResult ()
    {
       fRunTests = 0;
    }
}
```

이 간단한 TestResult 버전은 실행된 테스트 수만 계산합니다.
이를 사용하려면 TestCase.run() 메서드에 매개 변수를 추가하고 TestResult에 테스트가 실행 중임을 알려야 합니다.

```
public void run (TestResult 결과) {
    result.startTest (this);
  설정();
    runTest ();
    tearDown ();
}
```

그리고 TestResult는 실행된 테스트의 수를 추적해야 합니다.

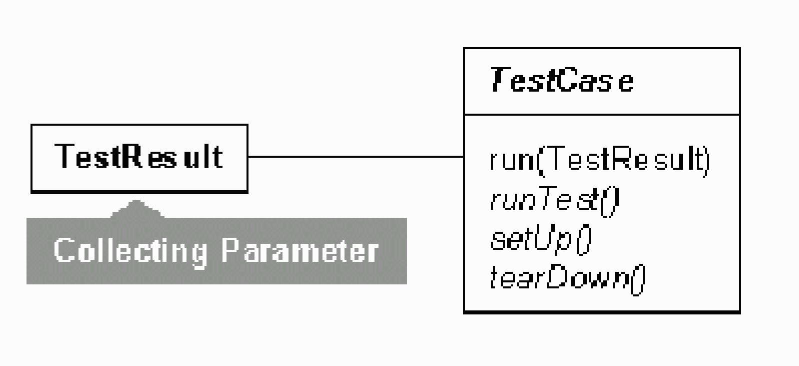
```
public synchronized void startTest (Test test)
{
    fRunTests ++;
}
```

테스트가 다른 스레드에서 실행될 때 단일 TestResult가 결과를 안전하게 수집할 수 있도록 TestResult 메서드 startTest를 synchronized로 선언합니다.
마지막으로 TestCase의 간단한 외부 인터페이스를 유지하려고 하므로 자체 TestResult를 생성하는 매개 변수가 없는 run() 버전을 생성합니다.

```
public TestResult run () {
    TestResult 결과 = createResult ();
  실행 (결과);
  반환 결과;
}
```

```
protected TestResult createResult () {
    return new TestResult ();
}
```

그림 3은 다음 설계 스냅샷을 보여줍니다.



테스트가 항상 올바르게 실행되었다면 작성하지 않아도 됩니다.

```
테스트는 실패할 때 흥미롭습니다.
특히 실패할 것으로 예상하지 않은 경우 더욱더 그렇습니다.
또한 테스트는 잘못된 결과를 계산하는 등 우리가 예상하는 방식으로 실패할 수 있으며,
예를 들어 배열 경계를 벗어난 쓰기를 통해 더 멋진 방식으로 실패할 수 있습니다.
테스트가 아무리 실패하더라도 다음 테스트를 실행하려고 합니다.

JUnit는 실패와 오류를 구분합니다.
실패 가능성이 예상되고 assertions로 확인됩니다.
오류는 ArrayIndexOutOfBoundsException과 같은 예상치 못한 문제입니다.
실패는 AssertionFailedError 오류로 표시됩니다.
예상치 못한 오류와 실패를 구별하기 위해 추가 catch 절 (1) 에서 실패를 포착합니다.
두 번째 절 (2) 은 다른 모든 예외를 포착하고 테스트 실행이 계속되도록 합니다.
```

```java
public void run(TestResult result)
{
    result.startTest(this);
    setUp();
    try{
        runTest();
    }
    catch(AssertionFailedError e){// 1
        result.addFailure(this, e);
    }
    catch(Throwable e){// 2
        result.addError(this, e);
    }
    finally{
        tearDown();
    }
}
```

```
AssertionFailedError는 TestCase에서 제공하는 assert 메서드에 의해 트리거 됩니다.
JUnit는 다양한 목적을 위해 일련의 assert 메서드를 제공합니다.
다음은 가장 간단한 것입니다.
```

```java
protected void assertTrue (boolean condition)
{
    if(!condition)
```

```
        throw new AssertionFailedError();
}
```

AssertionFailedError는 클라이언트(TestCase 내부의 테스트 메서드)가 포착하기 위한 것이 아니라
템플릿 메서드 TestCase.run() 내부에 있습니다. 따라서 오류에서 AssertionFailedError를 파생시킵니다.

```
public cliass AssertionFailedError extends Error
{
    public AssertionFailedError () {}
}
```

TestResult에서 오류를 수집하는 방법은 다음과 같습니다.

```
public synchronized void addError (Test test, Throwable t)
{
    fErrors.addElement (new TestFailure (test, t));
}
```

```
public synchronized void addFailure (Test test, Throwable t)
{
    fFailures.addElement (new TestFailure (test, t));
}
```

TestFailure는 나중에 보고할 수 있도록 실패한 테스트와 예외를 함께 바인딩하는 작은 프레임워크 내부 helper class입니다.

```
public class TestFailure extends Object
{
    protected Test fFailedTest;
    protected Throwable fThrownException;
}
```

collecting parameter의 표준 형식은 collecting parameter를 각 메서드에 전달해야 합니다.
이 조언을 따랐다면 각 테스트 방법에는 TestResult에 대한 매개 변수가 필요합니다.
이로 인해 이러한 메서드 서명이 "오염" 됩니다.
예외를 사용하여 실패를 알리는 자비로운 부작용으로 이러한 서명 오염을 피할 수 있습니다.
테스트 케이스 메서드 또는 여기에서 호출된 도우미 메서드는 TestResult에 대해 알 필요 없이 예외를 throw 할 수 있습니다.
여기에 MoneyTest 제품군의 샘플 테스트 방법이 있습니다.
테스트 방법이 TestResult에 대해 알 필요가 없는 것을 보여줍니다.

```
public void testMoneyEquals()
{
    assertTrue(!f12CHF.equals(null));
    assertEquals(f12CHF, f12CHF);
    assertEquals(f12CHF, new Money(12, "CHF"));
    assertTrue(!f12CHF.equals(f14CHF));
}
```

JUnit는 TestResult의 다양한 구현과 함께 제공됩니다.
기본 구현은 실패 및 오류 수를 계산하고 결과를 수집합니다.
TextTestResult는 결과를 수집하여 텍스트 형식으로 표시합니다.
마지막으로 UITestResult는 그래픽 테스트 상태를 업데이트하기 위해 JUnit Test Runner의 그래픽 버전에서 사용됩니다.
TestResult는 프레임 워크의 확장 지점입니다.
클라이언트는 자신의 사용자 정의 TestResult 클래스를 정의할 수 있습니다.
예를 들어 HTMLTestResult는 결과를 HTML문서로 보고합니다.

3.4 No stupid subclasses - TestCase again

테스트를 나타내기 위해 Command를 적용했습니다.
Command는 이를 호출하기 위해 execute()(TestCase에서 run()라고 함) 와 같은 단일 메서드에 의존합니다.
이 간단한 인터페이스를 통해 동일한 인터페이스를 통해 명령의 다른 구현을 호출 할 수 있습니다.

일반적으로 테스트를 실행하려면 인터페이스가 필요합니다.
그러나 모든 테스트 케이스는 동일한 클래스에서 다른 메서드로 구현됩니다.
이것은 클래스의 불필요한 확산을 방지합니다.

주어진 테스트 케이스 클래스는 각각 단일 테스트 케이스를 정의하는 다양한 메서드를 구현할 수 있습니다.
각 테스트 케이스에는 testMoneyEquals 또는 testMoneyAdd와 같은 설명 이름이 있습니다.
테스트 케이스는 간단한 Command 인터페이스를 따르지 않습니다.
동일한 Command 클래스의 다른 인스턴스는 다른 메서드를 사용하여 호출해야 합니다.
따라서 우리의 다음 문제는 테스트 호출자의 관점에서 모든 테스트 케이스를 동일하게 만드는 것입니다.

사용 가능한 디자인 패턴으로 해결된 문제를 검토하면 Adapter 패턴이 떠 오릅니다.
Adapter에는 "클래스의 인터페이스를 클라이언트가 기대하는 다른 인터페이스로 변환" 의도가 있습니다.
이것은 좋은 일치처럼 들립니다.
Adapter는 이를 수행하는 다양한 방법을 알려줍니다.
그중 하나는 인터페이스를 조정하기 위해 sub classing을 사용하는 클래스 Adapter입니다.

예를 들어 testMoneyEquals를 runTest에 적용하려면 MoneyTest의 하위 클래스를 구현하고 runTest를 재정의하여 testMoneyEquals를 호출합니다.

```java
public class TestMoneyEquals extends MoneyTest
{
    public TestMoneyEquals() {super("testMoneyEquals"); }
    protected void runTest() {testMoneyEquals(); }
}
```

sub class를 사용하려면 각 테스트 케이스에 대해 sub class를 구현해야 합니다.
이것은 테스터에게 추가적인 부담을 줍니다.

이것은 프레임워크가 테스트 케이스를 가능한 한 간단하게 추가해야 한다는 JUnit 목표에 위배됩니다.
또한 각 테스트 방법에 대한 하위 클래스를 만들면 클래스가 부풀어집니다.
하나의 메서드만 있는 많은 클래스는 비용이 들지 않으며 의미 있는 이름을 찾기가 어렵습니다.

Java는 클래스 이름 지정 문제에 대한 흥미로운 Java 관련 솔루션을 제공하는 익명 내부 클래스를 제공합니다.
익명의 내부 클래스를 사용하면 클래스 이름을 만들지 않고도 어댑터를 만들 수 있습니다.

```
    TestCase test = new MoneyTest ( "testMoneyEquals")
    {
        protected void runTest () {testMoneyEquals (); }
    };
```

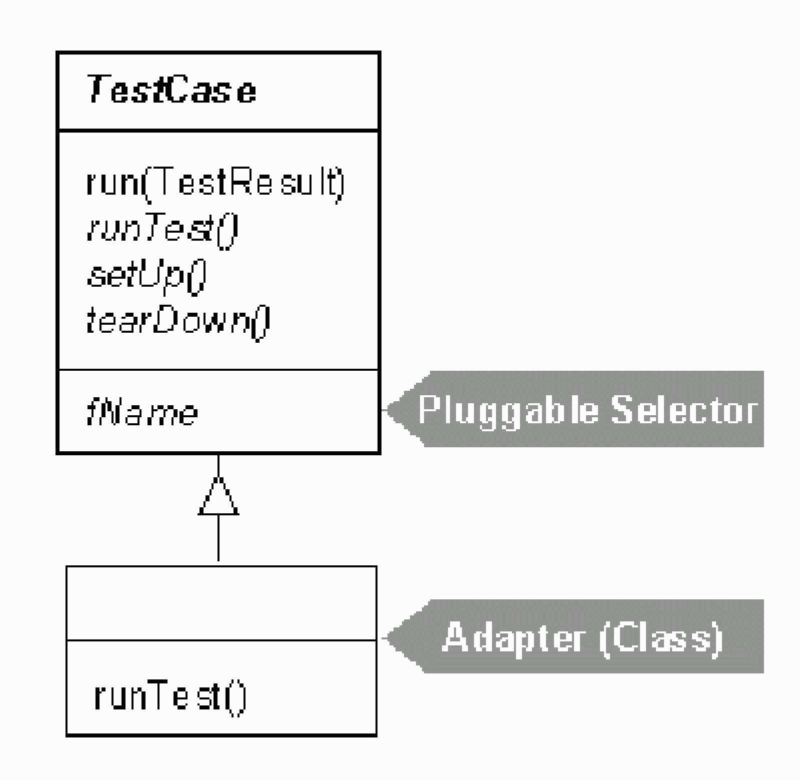
이것은 전체 sub classing보다 훨씬 편리합니다.
개발자의 부담을 감수하면서 컴파일 타임 유형 검사를 유지합니다.
smalltalk 모범 사례 패턴은 pluggable 동작이라는 공통 제목 아래에서 다르게 동작하는 여러 인스턴스의 문제에 대한 또 다른 솔루션을 설명합니다.
아이디어는 sub classing 없이 다른 로직을 수행하도록 매개 변수화할 수 있는 단일 클래스를 사용하는 것입니다.

가장 간단한 형태의 pluggable 동작은 pluggable Selector 입니다.
pluggable Selector는 인스턴스 변수에 smalltalk 메서드 Selector를 저장합니다.
이 아이디어는 smalltalk에만 국한되지 않습니다. Java에도 적용됩니다.
Java에는 메서드 Selector 개념이 없습니다.
그러나 Java reflection API를 사용하면 메서드 이름을 나타내는 문자열에서 메서드를 호출 할 수 있습니다.
이 기능을 사용하여 Java에서 pluggable Selector를 구현할 수 있습니다.
여담이지만, 우리는 일반적으로 일반 애플리케이션 코드에서 reflection을 사용하지 않습니다.
우리의 경우 우리는 인프라 프레임워크를 다루고 있으므로 reflection 모자를 쓰는 것이 좋습니다.

JUnit는 클라이언트에게 pluggable Selector를 사용하거나 위에 표시된 대로 익명 adapter class를 구현할 수 있는 옵션을 제공합니다.
이를 위해 runTest 메서드의 기본 구현으로 pluggable Selector를 제공합니다.
이 경우 테스트 케이스의 이름은 테스트 방법의 이름과 일치해야합니다.
reflection을 사용하여 아래와 같이 메서드를 호출합니다.
먼저 Method 객체를 찾습니다.
메서드 객체가 있으면 이를 호출하고 인수를 전달할 수 있습니다.
테스트 메서드는 인수를 받지 않기 때문에 빈 인수 배열을 전달할 수 있습니다.

```
protected void runTest() throws Throwable
{
    Method runMethod = null;
    try
    {
        runMethod = getClass().getMethod(fName, new Class [0]);
    }
    catch (NoSuchMethodException e)
    {
        assertTrue ("Method \" " + fName + "\"찾을 수 없음", false);
    }
    try
    {
        runMethod.invoke (this, new Class [0]);
    }
    // InvocationTargetException 및 IllegalAccessException 포착
}
```

```
JDK 1.1 reflection API를 사용하면 public 메서드를 찾을 수만 있습니다.
이러한 이유로 테스트 메서드를 public으로 선언해야합니다.
그렇지 않으면 NoSuchMethodException이 발생합니다.
다음은 Adapter 및 pluggable selector가 추가된 다음 디자인 스냅샷입니다.
```



3.5 Don’t care about one or many - TestSuite

```
시스템 상태에 대한 확신을 얻으려면 많은 테스트를 실행해야 합니다.
지금까지 JUnit는 단일 테스트 케이스를 실행하고 테스트 결과에 결과를 보고 할 수 있습니다.
다음 과제는 다양한 테스트를 실행할 수 있도록 확장하는 것입니다.
이 문제는 테스트 호출자가 하나 또는 여러 테스트 케이스를 실행하는지에 대해 신경 쓸 필요가 없을 때 쉽게 해결할 수 있습니다.
이러한 상황에서 가장 많이 사용되는 패턴은 Composite입니다.
"객체를 트리 구조로 구성하여 부분-전체 계층을 나타냅니다.
 Composite를 사용하면 클라이언트가 개별 객체와 객체 구성을 균일하게 처리 할 수      있습니다."
부분 전체 계층 구조에 대한 요점은 여기서 중요합니다.

suites of suites of suites of tests를 지원하려고합니다.

Composite는 다음 참가자를 소개합니다.
   Component : 테스트와 상호 작용하는 데 사용할 인터페이스를 선언합니다.
   Composite : 이 인터페이스를 구현하고 테스트 모음을 유지합니다.
   Leaf : Component 인터페이스를 준수하는 composition의 테스트 케이스를 나타냅니다.
이 패턴은 단일 및 복합 객체에 대한 공통 인터페이스를 정의하는 추상 클래스를 도입하도록 알려줍니다.
클래스의 주요 목적은 인터페이스를 정의하는 것입니다.
Java에서 Composite를 적용할 때 우리는 추상 클래스가 아닌 인터페이스를 정의하는 것을 선호합니다.
인터페이스를 사용하면 테스트를 위해 JUnit를 특정 기본 클래스로 커밋하지 않아도 됩니다.
필요한 것은 테스트가 이 인터페이스를 준수하는 것입니다.
```

따라서 패턴 설명을 조정하고 테스트 인터페이스를 도입합니다.

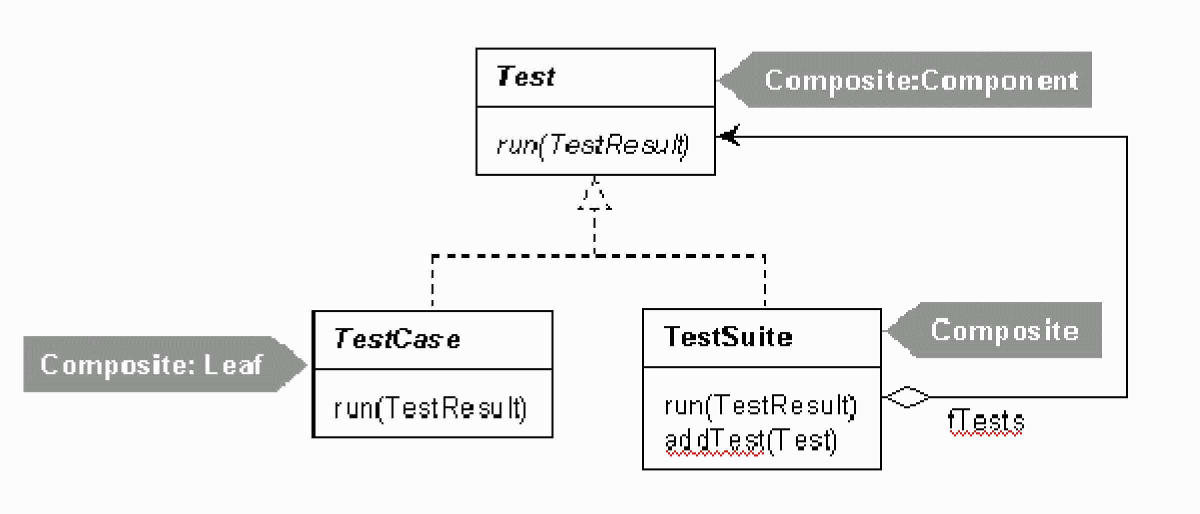
```
public interface Test
{
    public abstract void run(TestResult result);
}
```

TestCase는 Composite의 Leaf에 해당하며 위에서 본 것처럼 이 인터페이스를 구현합니다.
다음으로 Composite 참가자를 소개합니다.
클래스 이름을 TestSuite로 지정합니다.
TestSuite는 Vector에 자식 테스트를 유지합니다.

```
public class TestSuite implements Test {
    private Vector fTests = new Vector ();
}
```

run() 메서드는 자식에게 위임합니다.

```
public void run(TestResult result)
{
    for (Enumeration e = fTests.elements (); e.hasMoreElements ();) {
        Test test = (Test)e.nextElement ();
        test.run (result);
    }
}
```



마지막으로 클라이언트는 테스트를 suite에 추가할 수 있어야 하며 addTest 메서드를 사용하여 수행할 수 있습니다.

```
public void addTest (Test test)
 {
    fTests.addElement (test);
}
```

위의 모든 코드가 테스트 인터페이스에만 의존하는 방식에 유의하십시오.
TestCase와 TestSuite는 모두 Test 인터페이스를 따르기 때문에 test suites 모음을 재귀적

으로 구성 할 수 있습니다.
모든 개발자는 자신 의 TestSuite를 만들 수 있습니다.
이러한 제품군으로 구성된 TestSuite를 생성하여 모두 실행할 수 있습니다.
다음은 TestSuite를 만드는 예입니다.

```java
public static Test suite()
{
    TestSuite suite= new TestSuite ();
    suite.addTest(new MoneyTest ( "testMoneyEquals"));
    suite.addTest(new MoneyTest ( "testSimpleAdd"));
}
```

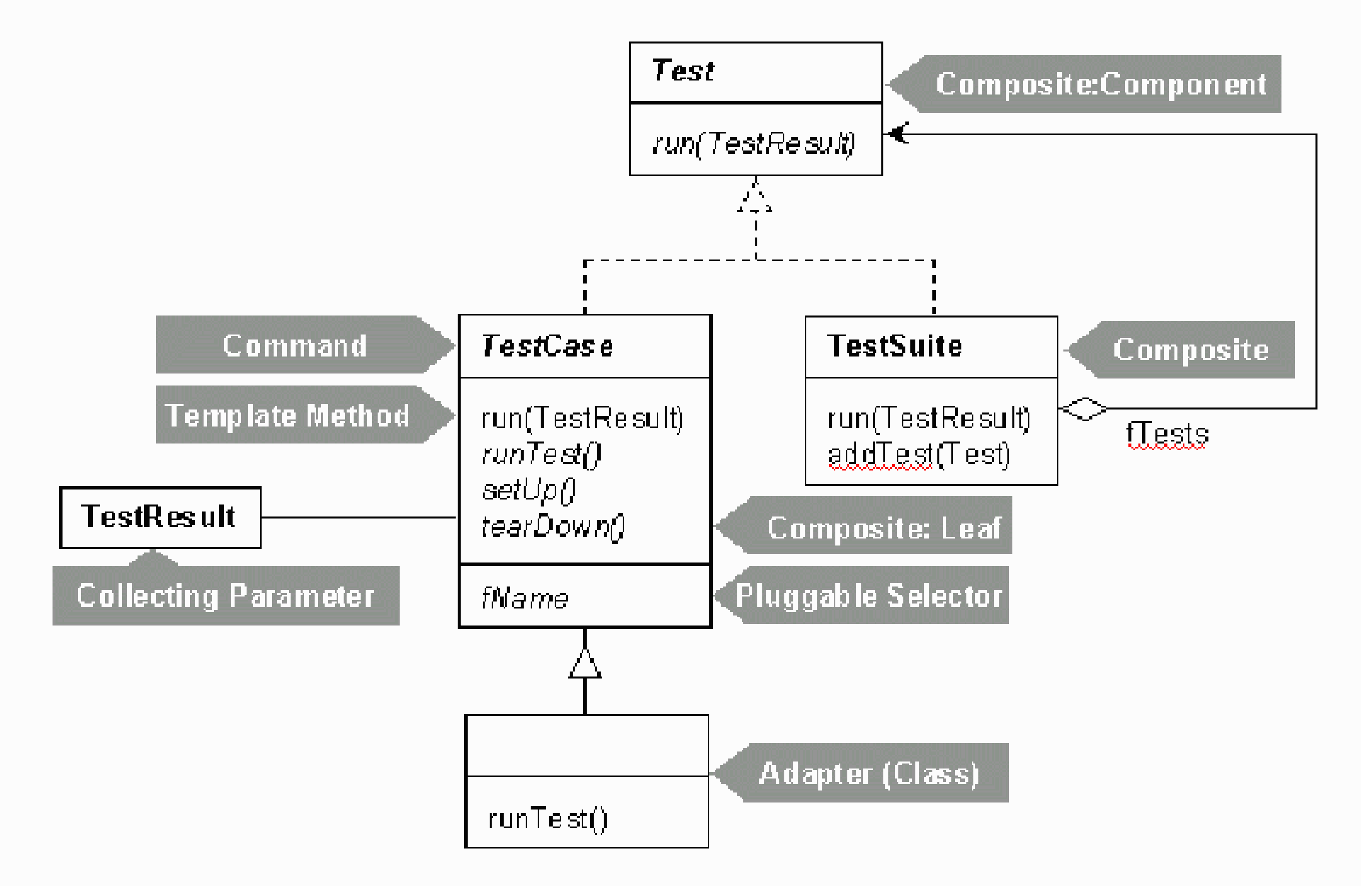
이것은 잘 작동하지만 모든 테스트를 스위트에 수동으로 추가해야 합니다.
JUnit의 얼리 어답터들은 이것이 어리석다고 말했습니다.
새로운 테스트 케이스를 작성할 때마다 이를 static suite() 메서드에 추가해야합니다.
그렇지 않으면 실행되지 않습니다.
테스트 케이스 클래스를 인수로 취하는 편리한 생성자를 TestSuite에 추가했습니다.
그 목적은 테스트 메서드를 추출하고 이를 포함하는 제품군을 만드는 것입니다.
테스트 메서드는 접두사 "test"로 시작하고 인수를 사용하지 않는 간단한 규칙을 따라야 합니다.
편의 생성자는 이 규칙을 사용하여 테스트 방법을 찾기 위해 reflection을 사용하여 테스트 객체를 구성합니다.
이 생성자를 사용하면 위의 코드가 다음과 같이 단순화됩니다.

```java
public static Test suite ()
{
    return new TestSuite (MoneyTest.class);
}
```

원래 방법은 테스트 케이스의 하위 집합을 실행하려는 경우에도 유용합니다.

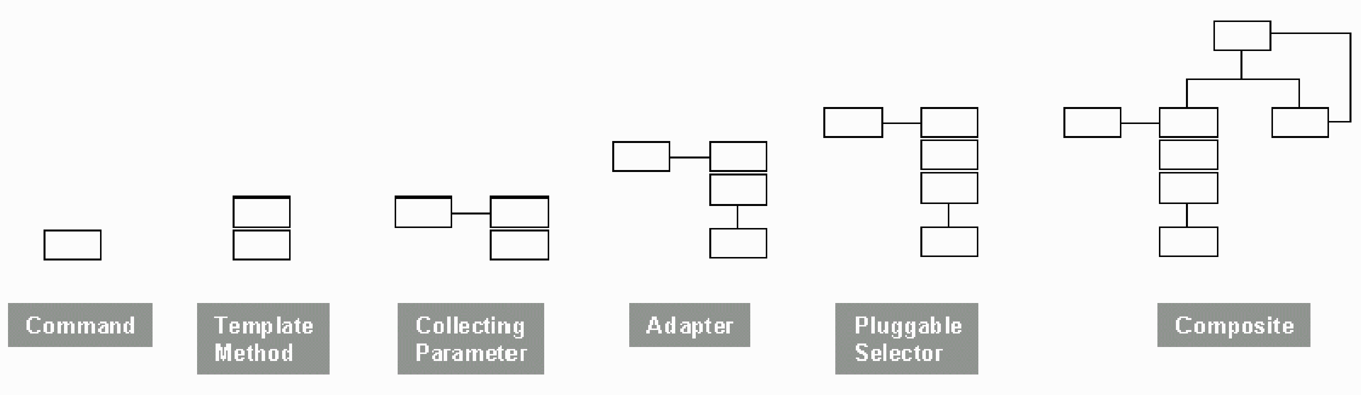
3.6 요약

JUnit A Cook's Tour는 끝이 났습니다.
다음 그림은 패턴으로 설명된 JUnit의 디자인을 한 눈에 보여줍니다.



프레임워크의 추상화 중심인 `TestCase`가 4가지의 디자인 패턴과 어떻게 관련되어 있는지 확인하십시오
성숙한 객체 디자인의 그림은 이와 동일한 "패턴 밀도"를 보여줍니다.
디자인의 스타는 지원하는 플레이어와 풍부한 관계를 맺고 있습니다.

다음은 `JUnit`의 모든 패턴을 보는 또 다른 방법입니다.
이 스토리보드에서는 각 패턴의 효과에 대한 추상적인 표현을 차례로 볼 수 있습니다.
따라서 `Command` 패턴은 `TestCase` 클래스를 만들고 `Templage Method` 패턴은 `run` 메서드를 만드는 식입니다.
(스토리보드의 표기법은 모든 텍스트가 삭제된 그림 6의 표기법입니다.)



스토리보드에서 주목해야 할 점은 `Composite`를 적용할 때 그림의 복잡성이 어떻게 점프하는지입니다.
이것은 `Composite`가 강력한 패턴이지만 "그림을 복잡하게 만든다"는 우리의 직감에 대한 그림 적 확증입니다.
따라서 주의해서 사용해야 합니다.

4. 결론

결론을 내리기 위해 몇 가지 일반적인 관찰을 해보겠습니다.

Patterns

우리는 프레임워크를 개발할 때와 다른 사람들에게 설명하려고 할 때
패턴 측면에서 디자인을 논의하는 것이 매우 중요하다는 것을 알았습니다.
이제 패턴이 있는 프레임워크를 설명하는 것이 효과적인지 판단할 수 있는 완벽한 위치에 있습니다.
위의 설명이 마음에 들면 자신의 시스템에 동일한 스타일의 프레젠테이션을 시도해 보십시오.

Pattern density

JUnit의 핵심 추상화인 TestCase 주변에는 높은 패턴 "밀도"가 있습니다.
패턴 밀도가 높은 디자인은 사용하기 쉽지만 변경하기가 더 어렵습니다.
우리는 키 추상화 주변의 높은 패턴 밀도가 성숙한 프레임 워크에서 일반적이라는 것을 발견했습니다.
미성숙한 프레임워크는 그 반대입니다.
패턴 밀도가 낮아야 합니다.
실제로 해결하고 있는 문제를 발견하면 솔루션을 "압축"하기 시작하여 더 조밀하고 조밀한 패턴 필드를 활용하여 활용할 수 있습니다.

Eat your own dog food (내부에서 사용)

기본 단위 테스트 기능을 구현하자마자 직접 적용했습니다.
TestTest는 프레임워크가 오류, 성공 및 실패에 대한 올바른 결과를 보고하는지 확인합니다.
우리는 프레임워크의 디자인을 계속 발전시켰을 때 이것이 매우 중요하다는 것을 알게 되었습니다.
우리는 JUnit의 가장 어려운 애플리케이션이 자체 동작을 테스트하는 것임을 발견했습니다.

Intersection, not union

프레임워크 개발에는 가능한 모든 기능을 포함하려는 유혹이 있습니다.
결국 프레임워크를 가능한 한 가치 있게 만들고 싶습니다.
그러나 이에 대응하는 힘이 있습니다. 개발자는 프레임워크를 사용하기로 해야 합니다.
프레임워크에 있는 기능이 적을수록 배우기가 더 쉬울수록 개발자가 이를 사용할 가능성이 커집니다.
JUnit는 이 스타일로 작성되었습니다. 테스트 실행에 절대적으로 필요한 기능 (테스트 모음 실행, 서로 테스트 실행 격리, 테스트 자동 실행)만 구현합니다.
물론 우리는 일부 기능을 추가하는 것을 거부할 수 없었지만, 그것들을 자체 확장 패키지
(test.extensions)에 넣도록 주의했습니다.
이 패키지의 주목할만한 멤버는 테스트 전후에 추가 코드를 실행할 수 있는 TestDecorator입니다.

프레임워크 작성자는 코드를 읽습니다.

우리는 JUnit 코드를 작성하는 것보다 훨씬 더 많은 시간을 소비했고,
새로운 기능을 추가하는 데 소비한 것만큼 중복 기능을 제거하는 데 거의 시간을 소비했습니다.
우리는 우리가 상상할 수 있는 한 다양한 방식으로 새로운 클래스를 추가하고 책임을 이동하면서 디자인을 적극적으로 실험했습니다.
우리는 JUnit, 테스트, 객체 디자인, 프레임워크 개발 및 추가 기사에 대한 기회에 대한 지속적인 인사이트 흐름을 통해 우리의 기사에 대해 보상을 받았습니다
(그리고 여전히 보상을 받고 있습니다).

최신 버전의 JUnit는 http://www.junit.org에서 다운로드할 수 있습니다.

5. 감사의 말

```
John Vlissides, Ralph Johnson, Nick Edgar에게 감사드립니다.
```