دانشگاه اصفهان

دانشکده فنی- مهندسی

گروه کامپیوتر

تحقیقی درباره‌ی Junit

تهیه‌کننده: **محسن مؤمنی**

نرم‌افزار تحقیق : **CASRE 3.0**

استاد گرامی: **دکتر ناصر نعمت‌بخش**

مقدمه

قطعاً تاکنون کدهای خود را تست کرده‌اید. دو نحوه برای انجام عمل تست وجوددارد:

۱) می‌توانید فرآیند تست را به صورت *ad hoc* انجام دهید. یعنی هر تستی را که در آن لحظه مناسب می‌بینید، روی کد امتحان کنید و نتیجه را ببینید.

۲) می‌توانید یک test suite بسازید (مجموعه‌ای از تست‌ها که می‌توانید در هر زمانی اجرا شود. و همیشه به صورت آماده موجود است.)

مضرات ایجاد test suite:

۱) programming زیادی را می‌طلبد و کار را پیچیده و مشکل می‌کند.

- درست است، این مشکل وجود دارد، اما با وجود یک framework مناسب این کار بسیار آسان‌تر می‌شود.

۲) ما زمان کافی برای انجام همه‌ی این عملیات اضافی را نداریم.

- این غلط است، آزمایش‌های فراوان نشان می‌دهدکه test suite همواره فرآیند debugging را بیش از زمانی که صرف ساخت test suiteها می‌شود، کاهش می‌دهد.

فواید ایجاد test suite:

- تعداد کل bugها را در کد تحویلی کاهش می‌دهد.

- کد را بسیار قابل نگهداری‌تر می‌کند و refactorable‌تر.
- این یک موفقیت بسیار بزرگ برای برنامه است که پیش از تحویل، استفاده‌ی واقعی داشته‌باشد.

در راهبرد Extreme Programming:

- تست‌ها پیش از نوشته‌شدنِ خود کد نوشته‌می‌شوند.
- اگر کد هیچ test case خودکاری نداشته باشد، فرض بر این است که کار نمی‌کند.
- Test frameworkd‌ی استفاده می‌شود، که پس از هر تغییر کوچک در کد، به صورت خودکار تست را انجام می‌دهد.
- که این کار غالباً ممکن است هر پنج تا ده دقیقه صورت گیرد.
- اگر bug‌ی پس از توسعه یافت شود، تستی برای جلوگیری از بازگشت آن bug ایجاد می‌شود.

پی‌آمدها:

- bug‌های کم‌تر
- کد قابل نگهداری‌تر
- تجمیع (integration) مداوم –در طول توسعه، برنامه همیشه کار می‌کند—شاید همه‌ی نیازها را برآورده نسازد، اما اگر کاری را انجام دهد، آن را درست انجام می‌دهد.

آهنگ (ریتم) انجام تست

- ریتمی برای ایجاد اویه‌ی software unit test‌ها وجود دارد.

ابتدا شما تستی را برای تعریف یک جنبه‌ی کوچک از مسئله‌ای که در دست است، ایجاد می‌کنید.

سپس ساده‌ترین کدی را که توانایی عبور از این آزمون دارد، را (به عنوان کد برنامه) ایجاد می‌کنید.

سپس second test را ایجاد می‌کنید.

حال شما به کد آنچه را باعث می‌شود، از این تست نیز با موفقیت عبور کند، اضافه می‌کنید، نه چیزی بیش‌تر!

این کار را ادامه می‌دهید تا چیزی برای تست باقی نماند.

http://www.extremeprogramming.org/rules/testfirst.html

Junit چیست؟

- Junit، framework‌ی برای نوشتن تست‌هاست.

- به‌وسیله‌ی Erich Gamma (مشهور در Design Pattern) و Kent Beck (ایجادکننده‌ی متدولوژی XP) نوشته شده است.
- این نرم‌افزار از قابلیت‌های بازتاب جاوا (Java's reflection capabilities) استفاده می‌کند. (برنامه‌های جاوا می‌توانند کد خود را آزمون کنند.)
- Junit به برنامه‌نویس کمک می‌کند تا:
  - ✓ تست‌ها و test suite‌ها را تعریف و اجرا کند.
  - ✓ نیازمندی‌ها را رسمی (formalize) کند و معماری را روشن سازد.

- ✓ کد را آسان‌تر بنویسد و آن را debug کند.
- ✓ کد را تجمیع کند و همیشه برای release نسخه‌ای کاری از نرم‌افزار آماده باشد.

- Junit هنوز در SDKیِ Sun وارد نشده، اما بسیاری از IDEها (از جمله Jbuilder، Eclipse، NetBeans و...) آن را در خود گنجانده‌اند.
- BlueJ، Jbuilder، Eclipse ابزارهای لازم برای حمایت از Junit را دارا هستند.

برخی تعاریف

- یک test fixture مجموعه‌ای از داده‌هاست که برای اجرای یک تست لازم است.
- یک unit test تست یک کلاسِ تکی (single class) است.
- یک test case تست‌هایی است که در سنجش واکنش‌های یک متد به قسمت خاصی از دامنه‌ی ورودی آن استفاده می‌شود.
- یک test suite مجموعه‌ای از test caseهاست.
- یک test runner نرم‌افزاری است که تست‌ها را اجرا می‌کند و نتایج را گزارش می دهد.
- تست تجمیع، تستی است که نشان می‌دهد کلاس‌ها چقدر هماهنگ با یکدیگر کار می‌کنند.
- Junit پشتیبانی‌های محدودی برای انجام تست‌های تجمیع دارد.

برای فهم به‌تر چگونگی کارکرد Junit ما در اینجا کلاسی به نام Counter را ایجاد و تست می‌کنیم.

توصیف (specification) کلاسی که باید ساخته شود به صورت زیر است:

۱) سازنده شیء را می‌سازد و آن را به مقدار صفر مقداردهی می‌کند.

۲) متد increment یک شماره به شمارنده اضافه می‌کند و مقدار جدید شمارنده را برمی‌گرداند.

۳) متد decrement یک شماره از شمارنده کم می‌کند و مقدار جدید را برمی‌گرداند.

پیش از آن که کد کلاس را آغاز کنیم، متدهای تست آن را می‌نویسیم. برحسب ابزاری که برای Junit استفاده می‌کنیم، ممکن است مجبور باشیم پیش از ساخت متدهای تست، کلاس را ایجاد کنیم.

```
public class CounterTest extends junit.framework.TestCase
{
Counter counter1;

public CounterTest() { }   // default constructor

protected void setUp() {   // creates a (simple) test fixture
counter1 = new Counter();
  }

protected void tearDown() { } // no resources to release

public void testIncrement() {
  assertTrue(counter1.increment() == 1);
```

```
  assertTrue(counter1.increment() == 2);
}

public void testDecrement() {
assertTrue(counter1.decrement() == -1);
}
}
```

```
public class Counter
{
        int count = 0;

        public int increment()
        {
          return ++count;
        }

        public int decrement()
        {
           return --count;
        }

        public int getCount()
        {
          return count;
        }
}
```

دید Extreme Programming چنین است: اگر چیزی تست نشده، فرض بر این است که کار نمی کند. مجبور نیستید برای همه‌ی کلاس‌ها کلاس تست به این گُندگی بنویسید. اغلب برنامه‌نویسان XP برای متدهای ساده‌ای مثل getCount() دیگر تست نمی‌نویسند. ما در این مثال تنها از assertTrue استفاده کردیم، اما assertهای دیگری نیز وجود دارند.

assertEquals(***expected***, ***actual***)
assertEquals(String ***message***, ***expected***, ***actual***)

در این متدها ارگومان اول و دوم هر دو باید شیء باشن یا ان که از یک نوع اولیه‌ی یکسان مشتق شده باشند. برای اشیاء اگر متد public boolean equals(Object o) تعریف شده باشد، تست از آن بهره می‌گیرد و گرنه از عملگر مساوی استفاده می‌شود.

assertSame(Object ***expected***, Object ***actual***)
assertSame(String ***message***, Object ***expected***, Object ***actual***)

که نشان می‌دهد، دو ارجاعِ شیء، به شیء واحدی ارجاع دارند یا خیر.

assertNotSame(Object ***expected***, Object ***actual***)
assertNotSame(String ***message***, Object ***expected***, Object ***actual***)

assertNull(Object ***object***)
assertNull(String ***message***, Object ***object***)

assertNotNull(Object ***object***)
assertNotNull(String ***message***, Object ***object***)

fail()
fail(String ***message***)

باعث می‌شود تست fail شود و یک AssertionFailedError ارسال گردد. این متد زمانی که با تست‌هایی پیچیده‌ای مواجه هستیم که دیگر متدها بکار نمی‌آیند می‌تواند مفید واقع شود.

دو فرم از جمله‌ی assert وجود دارد:

assert ***boolean_condition***;
assert ***boolean_condition***: ***error_message***;

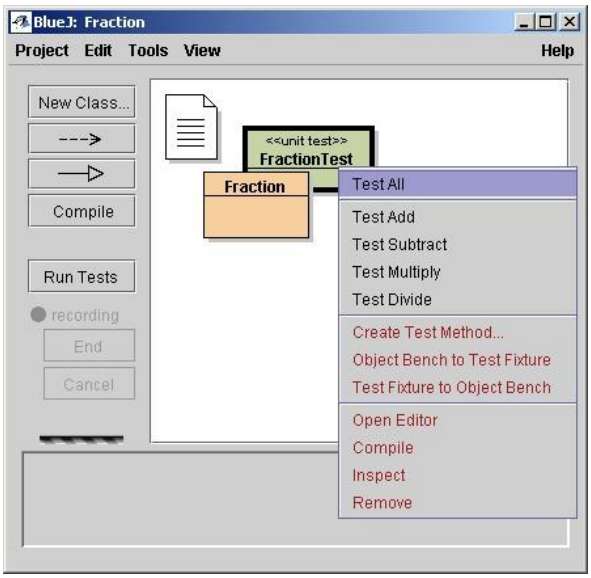
هر دو فرم چنانچه ***boolean_condition*** درست نباشد، یک AssertionFailedError برمی گردانند. فرم دوم برای تست‌هایی لازم است که از جمله‌ی assert آن‌ها:

Use it to document a condition that you "know" to be true.
Use assert false; in code that you "know" cannot be reached (such as a default case in a switch statement).
Do **not** use assert to check whether parameters have legal values, or other places where throwing an Exception is more appropriate.

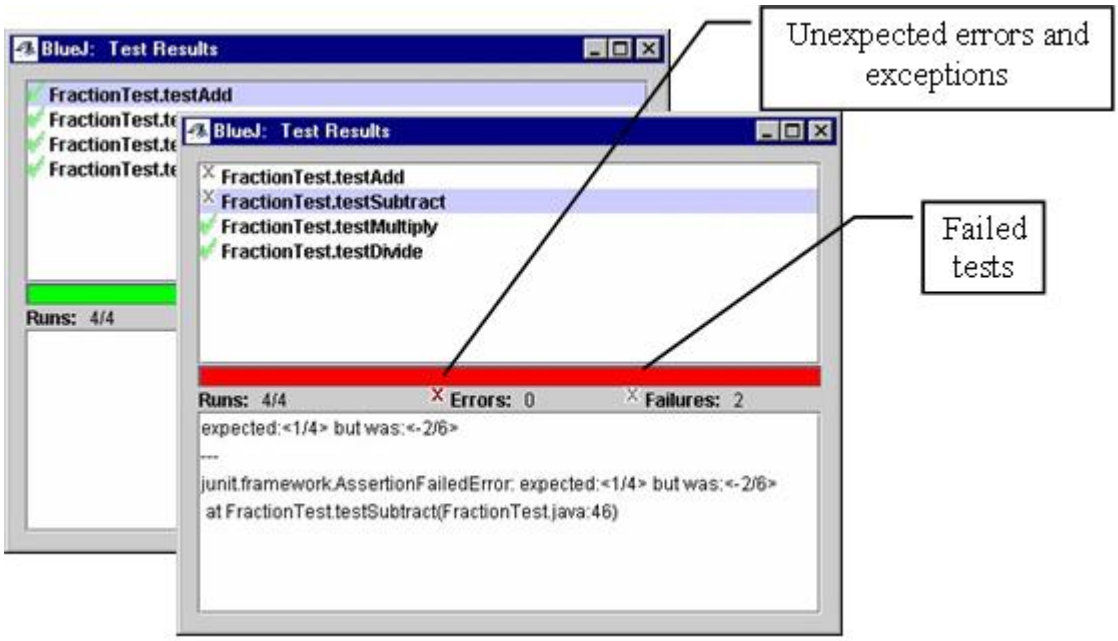
BlueJ

BlueJ 1.3.0 پشتیبانی Junit را با دستوراتی چون Create Test Class و Create Test Method فراهم می‌کند. می‌توانید اشیایی را در میز تست (test bench) بسازید و به fixture ببرید، یا آن را دوباره برگردانید. همچنین BlueJ یک مُد "recording" دارد که در آن شما می‌توانید اشیاء را بسازیدو تغییر دهید و BlueJ اعمال شما را روی test code اعمال می‌کند.

BlueJ همچنین اجرای یک تست تکی، یک suite از تست‌ها و یا همه‌ی تست‌ها را فراهم می‌سازد.



برای استفاده از Junit در BlueJ، به قسمت Prefrences بروید و نمایش ابزارهای Unit Testing را چک کنید. Junit در BlueJ منوهای خاص خود را دارد.



اگر یک تست تکی را اجرا کنید و موفقیت‌آمیز باشد، تنها یک پیام مبنی بر موفقیت در قسمت نوار وضعیت خواهید دید.

JBuilder

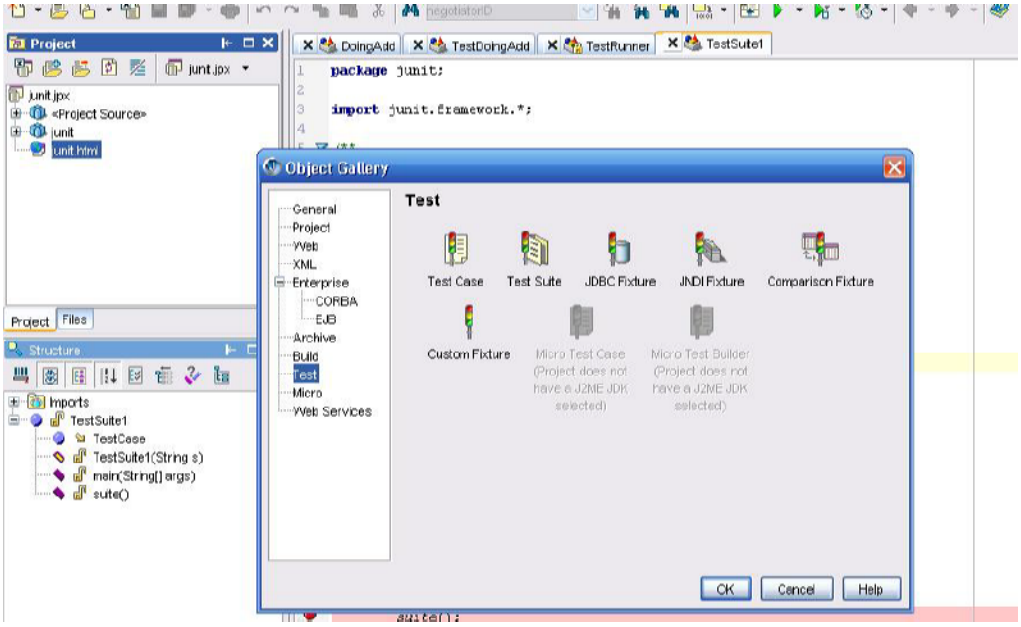
در Jbuilder کلاس‌هایی که برای انجام عمل تست وجود دارند و مفید هستند در یک مجموعه گردآوری شده‌اند، هنگامی‌که گزینه‌ی new را از منوی فایل به منظور ساختن کلاس جدید کلیک می‌کنید، یکی از انتخاب‌ها در ساختن کلاس جدید رفتن به گالری test و انتخاب یکی از ۸ گزینه‌ی ممکنی است که در این مجموعه در اختیار شما قرار گرفته است. با انتخاب هر یک از آن‌ها ویزاردی پیش روی شما قرار می‌گیرد، که به شما در تکمیل قسمت‌هایی از آن ابزار تست برای استفاده کمک می‌کند.

این هشت ابزار این‌ها هستند:

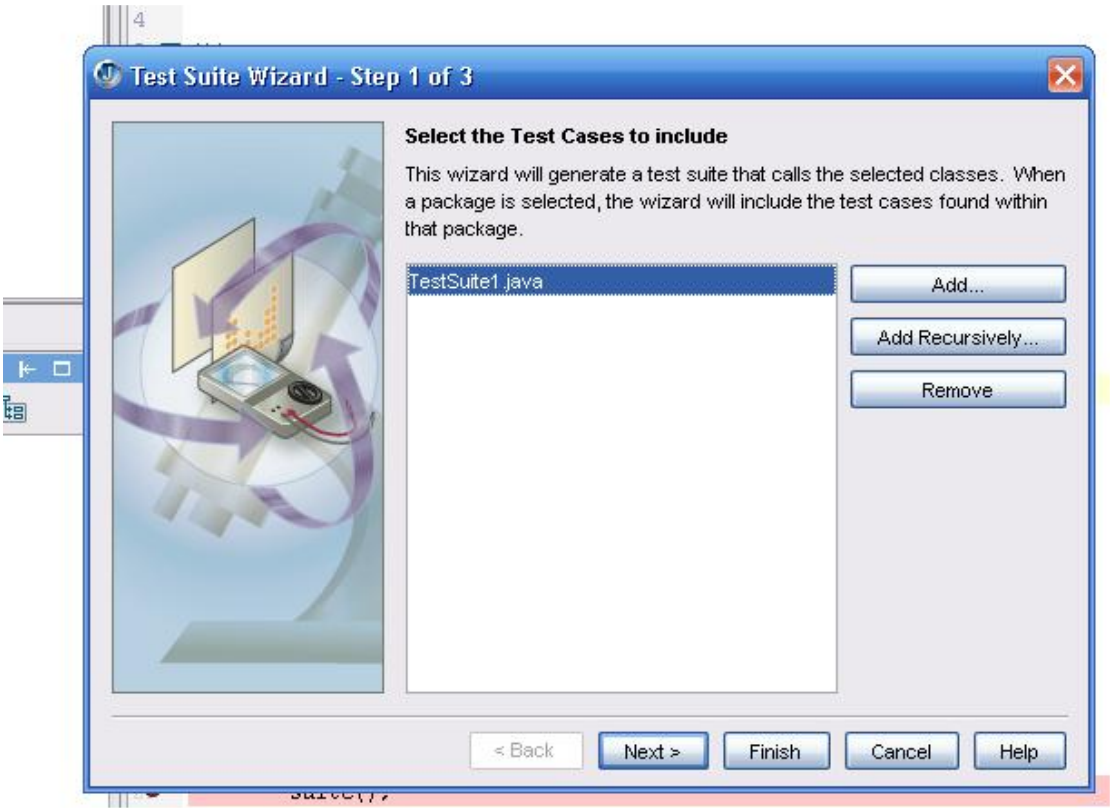
Test Case, Test Suite, JDBC Fixture, JNDI Fixture, Comparision Fixture, Custom Fixture, Micro Test Cas, Micro Test Builder.

در زیر شمایی از ویزارد ابتدایی را می‌بینید.

Copyright (c) 1997-2004 Borland Software Corporation. All Rights Reserved

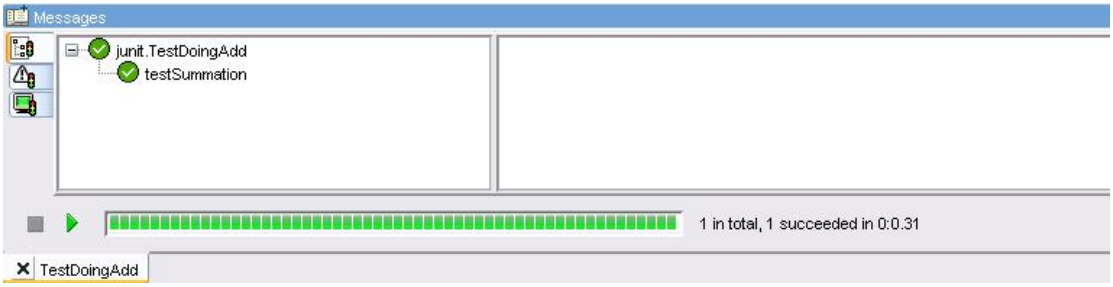


پس از انتخاب یکی از این کلاس‌ها، با پیش‌آمدن ویزاردی شبیه ویزارد پایین مراحلی طی می‌شود که طی آن کلاسی که مورد تست قرار می‌گیرد، متدهای مورد تست، احیاناً Runner این تست و... انتخاب می‌شوند.



پس از تکمیل این گزینه‌ها، کلاسی که از کلاس گزینه‌ی مورد نظر ارث برده است، با موارد انتخابی ساخته می‌شود. اینک می‌توان تغییرات لازم را در source code آن اعمال نمود و آن را تا تبدیل شدن به یک test case یا test suite کامل، که همه‌ی دامنه‌های مهم را چک کند، پیش برد. در نهایت پس از اجرای هر تست پنجره‌ای در قسمت

پایین Jbuilder editor ظاهر می‌شود، که چگونگی انجام عمل تست و نتایج آن را گزارش می‌دهد. کـه نمونـه‌ای از آن را در زیر می‌بینید.



نکاتی درباره‌ی Junit

Junit برای فراخوانی متدها و مقایسه‌ی نتایجی که برمی گردانند با نتایج مطلوب طراحـی شـده‌اسـت. امـا همیـشه بیش‌تر متدها side effect و تأثیرات جانبی دارند.

برای متدهایی که خروجی دارند، باید خروجی را قیاس کرد.

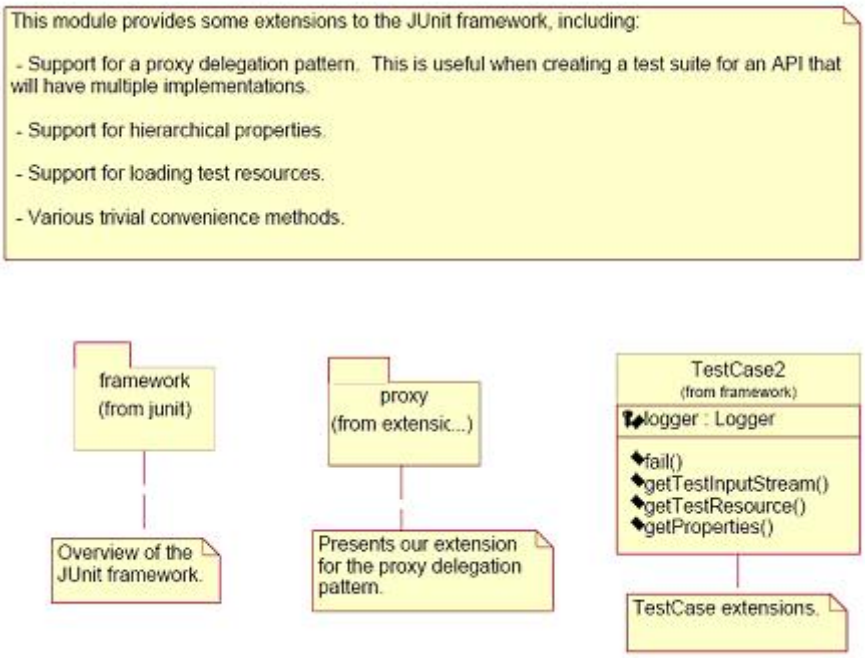
برای متدهایی که حالت شیئ را تغییر می‌دهند، باید کدی داشته‌باشید که حالت را نیز چک کند. یک ایده‌ی خـوب این است که در هر قسمت، یک self-test برای چک کردن validity شیء بنویسیم.

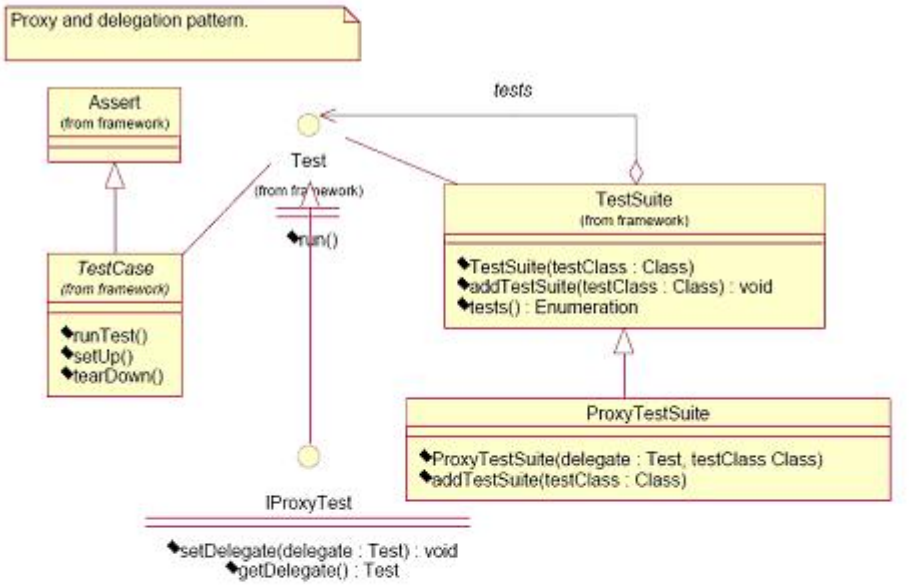
ضمناً سعی کنید متدها را تا حد ممکن بدون تأثیرات جانبی بنویـسید. ایـن امـر بـه reusability یـک کـلاس یـا component کمک شایانی می‌کند.

می‌توان عمل چاپ خروجی را به جای فرستادن به خروجی کنسول به هر پرینت‌استریم دیگری نیز فرستاد.

معماری Junit

در زیر می‌توانید، نمودارهای uml مربوط به معماری و ساختار کلاس‌های مربوط به Junit را ببینید.

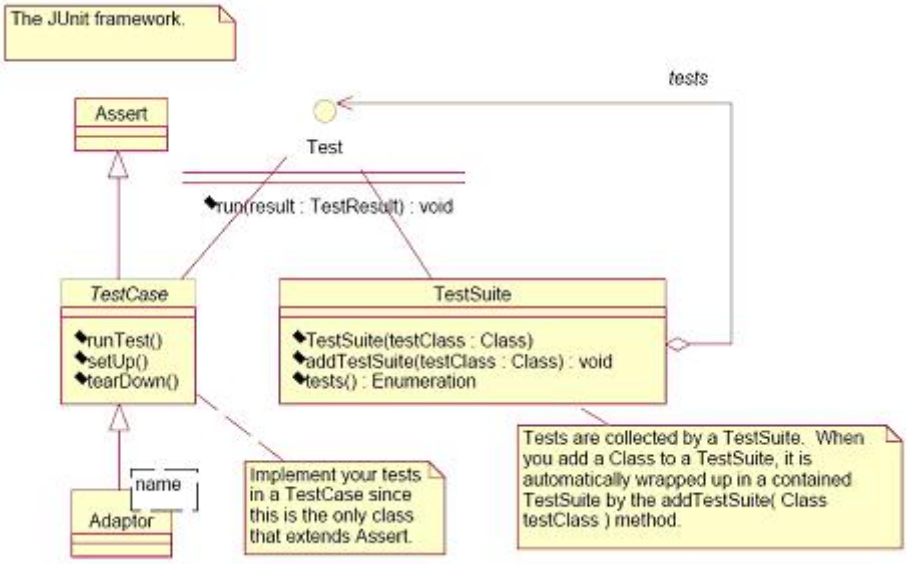


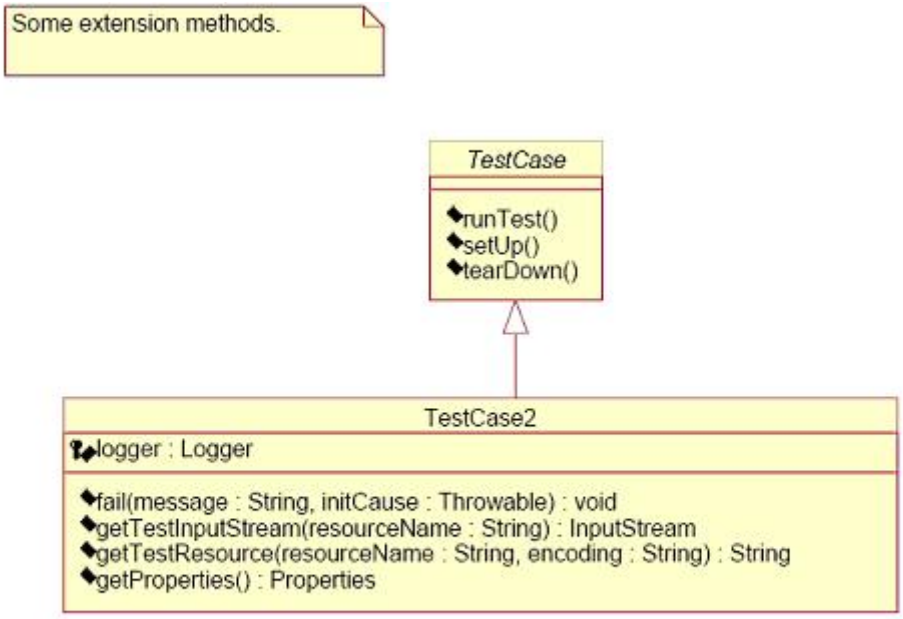


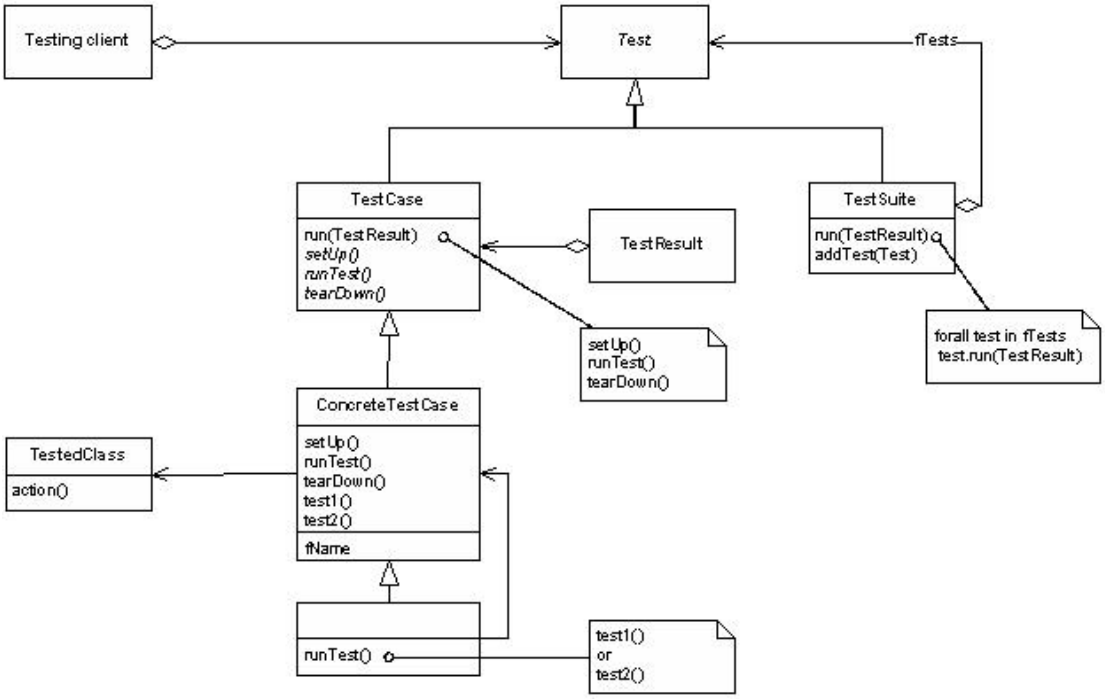
The use of the proxy and delegation pattern with JUnit is appropriate when you want to factor apart a test suite for some API and you have multiple implementations of that API with the consequence that implementation specific fixtures must be established for testing. Under these conditions you have a multiple inheritance problem since the API test suite needs to extend the implementation specific test cases, but the API test suites must be independent of the specific implementations.

The JUnit framework makes stateless invocations of tests. In order to solve the multiple inheritence problem, you need to be able to provide one item of state to each of the tests in the API test suite -- a delegate Test. Normally the delegate Test will expose some abstract behavior and the implementation specific TestCase will provide a concrete implementation of that behavior. The tests in the API test suite will implement the IProxyTest interface so that they can gain access to that delegate. The implementation specific test case extends TestCase.

The use of the proxy and delegation pattern makes it easy to repeat a test suite for different configurations as well. The delegate TestCase typically implements a method that reads Properties that are used to establish the configuration.







منابع و مآخذ

Junit - http://www.junit.org

Java Tools for Extreme Programming: Mastering Open Source Tools Including Ant, JUnit, and Cactus.

Beck, K., and Gamma, E., . JUnit Test Infected: Programmers Love Writing Tests,. *Java Report*, July 1998, Volume 3, Number 7. Available on-line at:

http://JUnit.sourceforge.net/doc/testinfected/testing.htm

Beck, K., and Gamma, E., .JUnit A Cook.s Tour,. *Java Report*, 4(5), May 1999. Available on-line at: http://JUnit.sourceforge.net/doc/cookstour/cookstour.htm

Beck, K., and Gamma, E., .JUnit Cookbook. Available on-line at http://JUnit.sourceforge.net/doc/cookbook/cookbook.htm

Beck, K., .Extreme Programming Explained,. Addison-Wesley, 2000.

Guttag, J.V., and Horning, J.J., .The Algebraic Specification of Abstract Data Types,. *Acta Informatica* 10 (1978), pp. 27-52.

J. Guttag, E. Horowitz, D. Musser, .Abstract