The SysUnit Manual

SysUnit Project
www.sysunit.org

July 9, 2003

Contents

List of Figures

Chapter 1

Introduction

JUnit is a framework ostensibly useful for testing individual units of a system. The framework is more generally useful for testing at different levels, not simply the unit level. JUnit has been integrated into build systems such as Jakarta-Ant and Apache-Maven. The integration with build tools allows the tests to be run during the course of every developer build, ensuring integrity of the system at any time.

Unit tests are typically created to test a very small, isolated portion of the system, while *system tests* test larger grained chunks of an application. System tests may test complex iteractions between many units simultaneously. The concurrency required by many system tests is not directly supported with the base JUnit framework. SysUnit adds to the base JUnit framework to support concurrent and distributed system testing. By building upon JUnit, SysUnit allows system tests to be run at the same time, using the same mechanisms, as an existing base of unit tests. From the point-of-view of Ant or Maven, a SysUnit system test is simply yet-another-test.

Chapter 2

System Testing

Unit tests are by definition tests that verify the operation of a *unit* of code. There is no absolute strict definition of unit but most view a unit to be equivalent to a single implementation class. In some instances, though, a unit may be composed of multiple classes within a package.

When testing the interactions of multiple units, you have graudated to *system testing*, which may verify more complex behaviour of the system. Many times system testing requires concurrently executing several bits of code, synchronizing their execution, and verifying the results across the entire environment.

> **Editor's Note**
>
> *More description about how to write good system-tests is needed here. Examples, etc.*

Chapter 3

The SystemTestCase Class

3.1 Overview

When using the base JUnit framework, each unit test is represented by a single method of the form textXXX() on a subclass of TestCase. In this way, each subclass of TestCase may actually contain several tests.

Unlike the base JUnit framework, a SysUnit test class creates only a single test. A system test is created by subclassing the SystemTestCase class. JUnit's textXXX() idiom is *not* used when creating tests, since SysUnit creates a test-per-class, instead of a test-per-method.

Each portion of the system that interacts within a test is a TBean, which may be interpreted as either a *test bean* or a *thread bean.* A TBean is similar to the standard Runnable interface, with a few modifications to assist with testing. When a system test is executed, all of the test's TBean instances are executed in parallel. The SysUnit framework handles the spawning of the threads necessarily for concurrently executing the code.

There are two different ways to create TBean instances for a test:

1. TBean factory method, using tbeanXXX().

2. Thread factory method, using threadXXX().

3.2 TBean Factory Method

To configure a unit of code to execute concurrently, a *TBean factory method* may be used. It is simply a method that begins with the prefix tbean and returns a TBean. When the test class is executed, each TBean factory method will be called to generate the set of TBean instances used for the test.

```
public class MySystemTest
  extends SystemTestCase
{
    public TBean tbeanOrderAcceptor()
    {
        return new OrderAcceptorTBean();
    }

    public TBean tbeanInventoryVerifier()
    {
        return new InventoryVerifierTBean();
    }
}
```

3.3 Thread Factory Method

To aide in creating concurrent tests without the need of embodying each unit of code within a `TBean`, SysUnit supports the notion of a *thread factory method.* In addition to the `TBean` factory method for creating executable segments of code, any method that has the prefix of `thread` will also be run concurrently.

> **Note**
>
> *The term "thread" is used through this manual to denote a general thread of execution, and not necessarily instances of `java.lang.Thread`, though they may certainly be implemented using the `Thread` class under the covers.*

```
public class MySystemTest
  extends SystemTestCase
{
    public void threadOrderAcceptor()
        throws Exception
    {
         // the logic for running the order acceptor
    }

    public void threadInventoryVerifier()
        throws Exception
    {
         // the logic for running the inventory verifier
    }
}
```

3.4 Synchronization

Many times, while multiple segments of code are running in independent threads of execution, it is desirable to synchronize at a point. In multi-threaded programming, this is considered a *barrier*, which is a point in the code that causes the threads to block and wait until all threads reach the same point, and then continue independently once unblocked.

> **Note**
>
> *The term “synchronization” implies coordination, not unlike how secret-agents will synchronize their watches to ensure activities happen at the same time. SysUnit’s usage of “synchronization” should not be confused with Java’s* `synchronized` *keyword, which actually ensures that multiple things* do not *occur at the same time.*

3.4.1 Synchronizer

The basis for synchronizing across multiple threads (local or distributed) is the `Synchronizer`, which describes a single `sync(...)` method.

```
public interface Synchronizer
{
    void sync(String tbeanId,
              String syncPointName)
        throws SynchronizationException, InterruptedException;
}
```

The `Synchronizer` provides a means for blocking at a named barrier, or *sync-point* until all other threads of execution block at a sync-point or have completed. The `tbeanId` parameter identifies the `TBean` requesting the synchronization, while the `syncPointName` identifies the particular sync-point to wait for. The `tbeanId` is typically derrived from the `tbeanXXX()` and `threadXXX()` method names, after removing the prefix.

The `syncPointName` is any arbitrarily chosen identifier for the synchronization rendezvous. The following example shows three `TBean` instances synchronizing at a sync-point named “after-initialization”

```
    synchronizer.sync( "FirstTBean",
                       "after-initialization" );
```

```
    synchronizer.sync( "SecondTBean",
                       "after-initialization" );
```

```
    synchronizer.sync( "ThirdTBean",
                       "after-initialization" );
```

The `Synchronizer` guarantees that all three threads will reach the call to `sync(...)` and wait until the others have also reached that point. All threads will then simultaneously unblock, allowing execution to continue. Any number of sync-points may be used through-out a test in order to allow coordination of multiple phases.

3.4.2 SynchronizableTBean, AbstractSynchronizableTBean and TBeanSynchronizer

The base TBean interface does not allow for synchronization. The TBean subinterface SynchronizableTBean adds setSynchronizer(...) method which signals to the SysUnit framework that a specialized TBeanSynchronizer should be provided for synchronization within the TBean.

```
public interface SynchronizableTBean
    extends TBean
{
    void setSynchronizer(TBeanSynchronizer synchronizer);
}
```

The TBeanSynchronizer manages the tbeanId parameter required by Synchronizer.sync(...), providing a simple synchronization interface which requires only a syncPointName parameter.

```
public interface TBeanSynchronizer
{
    void sync(String syncPointName)
        throws SynchronizationException, InterruptedException;
}
```

The AbstractSynchronizableTBean provides a base for implementations of SynchronizableTBean to manage the TBeanSynchronizer and provides a convenience sync(...) method.

```
public class MyFirstTBean
    extends AbstractSynchronizableTBean
{
    public void run()
        throws Exception
    {
        doSomething();
        sync( "after-something" );
        doSomethingElse();
        sync( "after-something-else" );
    }
}
```

```
public class MySecondTBean
    extends AbstractSynchronizableTBean
{
    public void run()
        throws Exception
    {
        doDifferentSomething();
        sync( "after-something" );
        doDifferentSomethingElse();
        sync( "after-something-else" );
    }
}
```

In general, TBean implementations should extend the AbstractSynchronizableTBean abstract class to make writing test code simpler. Management of the Synchronizer and tbeanId will be managed automatically.

3.4.3 SystemTestCase.sync(...)

When using the threadXXX(...) form for creating implicit TBean instances, the SystemTestCase provides a simple sync(...) identical to the one provided by AbstractSynchronizableTBean.

```
public class MySystemTest
    extends SystemTestCase
{
    public void threadOne()
    {
        doSomething();
        sync( "after-something" );
    }

    public void threadTwo()
    {
        doDifferentSomething();
        sync( "after-something" );
    }
}
```

Chapter 4

Test Validation

4.1 Introduction

Code does not a test make. Only through validating the results of the code does a test gain true value. JUnit provides a range of assertion methods for validating the state of a test. Assertions are a way of validating that the code being tested produced the expected results.

4.2 Validation with `AbstractTBean`

The abstract convenience `TBean` implementations of `AbstractTBean` and `AbstractSynchronizableTBean` extend the base JUnit `Assert` class, allowing for all normal JUnit validation methods to be used.

```
public class MyFirstTBean
    extends AbstractSynchronizableTBean
{
    public void run()
        throws Exception
    {
        String result = doSomething();
        sync( "after-something" );
        assertEquals( "garbonzo",
                      result );
    }
}
```

Additionally, the `TBean` class provides for an `assertValid()` method to do validation after all threads have completed.

```
public class MyFirstTBean
    extends AbstractSynchronizableTBean
{
    public void run()
        throws Exception
    {
        doSomething();
    }

    public void assertValid()
        throws Exception
    {
        assertEquals( this.garbonzo,
                      "garbonzo"
    }
}
```

4.3 Validation with `threadXXX(...)` methods

Since `threadXXX(...)` methods appear on implements of `SystemTestCase` which itself is a subclass of JUnit's `TestCase`, all `Assert` methods are available.

```
public class MyFirstSystemCase
    extends SystemTestCase
{
    public void threadOne()
    {
        String result = doSomething();
        sync( "after-something" );
        assertEquals( "garbonzo",
                      result );
    }
}
```

Chapter 5

TBeanManager

5.1 Introduction

SysUnit tests are created to coordinate multiple threads of execution. There is no constraint that all threads of execution must execution within a single process or even a single machine. The `TBeanManager` is a pluggable strategy to allow for the creation, execution, synchronization and coordination of `TBean` instances in various environments. It is never necessary for test implementors to concern themselves with `TBeanManager` instances. It is purely an implementation class within the core of SysUnit.

The `TBeanManager` may be selected using the `org.sysunit.TBeanManager` system-property.

5.2 `LocalTBeanManager`

The `LocalTBeanManager` is the default `TBeanManager` and runs all system `TBean` instances within a single JVM.

5.3 `DistributedTBeanManager`

> **Implementor's Note**
>
> *So far, only `LocalTBeanManager` exists, due to previous problems attempting to use JBoss to create a `RemoteTBeanManager`. Will either attempt a go with mx4j or possibly just a custom-rolled test-cluster node daemon that provides classloader isolation.*

Appendix A

Licensing

A.1 SysUnit License

A.1.1 The License

Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain copyright statements and notices. Redistributions must also contain a copy of this document.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name `SysUnit` must not be used to endorse or promote products derived from this Software without prior written permission of The SysUnit Project. For written permission, please contact info@codehaus.org.
4. Products derived from this Software may not be called `SysUnit` nor may `SysUnit` appear in their names without prior written permission of The SysUnit Project. `SysUnit` is a registered trademark of The SysUnit Project.
5. Due credit should be given to The SysUnit Project.

THIS SOFTWARE IS PROVIDED BY THE SYSUNIT PROJECT AND CONTRIBUTORS **AS IS** AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE SYSUNIT PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

A.1.2 Summary

SysUnit is provided under a license similar to that used by The Apache Software Foundation. It is a commercial-friendly license in that it allows you to modify and distribute SysUnit in either source or binary form. While you are *encouraged* to contribute changes back to the project, you are by no means *required* to do so. The SysUnit license is not viral or infectious. It does not alter how you license you own product. If you have any questions regarding the licensing of SysUnit please contact `info@sysunit.org`.

Colophon

This manual was produced without the aide of Microsoft products. It was authored on a Linux laptop using XEmacs, LaTeX, `makeindex`, and `xdvi`.

Phish rocks.

Index