TESTING

The first release of the CommonPoint system supports testing in three areas:

- **The Test framework** supports execution, logging, and evaluation of tests.
- **Utility Tests** use the Test framework to implement standard tests.
- **Assertions** provide a mechanism for asserting invariants in a program and generating exceptions when these invariants aren't met.

This section focuses primarily on the Test framework.

Overview of the Test framework

The Test framework provides a consistent structure for running, maintaining, documenting, and reusing test code. Test code is not only separate from the code being tested but also completely independent of the user interface. This allows you to test objects that aren't affected by changes to the user interface early in the development process. The Test framework also provides convenience classes for testing standard CommonPoint behavior such as copying and streaming.

The Test framework makes it possible to link tests to specific parts of your code, as suggested by Figure 154, and run them with a testing utility provided by Taligent. The RunTest utility is used for this purpose with the CommonPoint reference release, which is hosted on AIX. Taligent plans to provide equivalent utilities for use with versions of CommonPoint that run on different hosts.

The TTest class, which is the core of the Test framework, provides a uniform protocol for

- Invoking tests
- Reporting test results
- Connecting tests to the class being tested
- Setting up tests and cleaning up afterward
- Combining tests
- Logging and timing

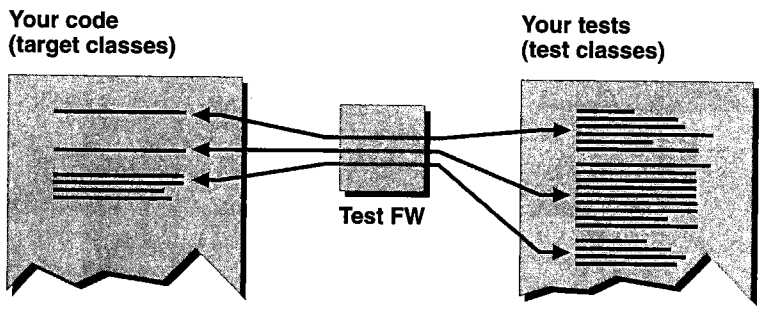


FIGURE 154 THE TEST FRAMEWORK PROVIDES A CONSISTENT INTERFACE BETWEEN YOUR CODE AND YOUR TESTS

To use the Test framework, you write a test class derived from TTest that evaluates the behavior of the class you are testing, called the *target class.* When you run the test, the test class creates an instance of the target class and then compares its actual behavior with the expected behavior. If the results are the same, the test passes.

A *decision function* is the code you write that determines whether the code being tested is behaving as you expect. For example, if an object simply returns a value sent to it, a decision function might compare a value sent to the object with the value the object returns and make sure the two are equal.

Here is an example of a decision function that compares the value for a variable with an expected length:

```
if (maxlength != fExpectedLength)
    SetSuccess(false);
```

If the two values don't match, the function passes a value of `false` to SetSuccess, which means that the test failed.

A single TTest class can exercise a single decision function or can parse the input to select a subset of decision functions, as suggested by Figure 155.

You decide how much testing a single TTest class needs to perform. If a single TTest turns up several defects, the scope of the test is too large.

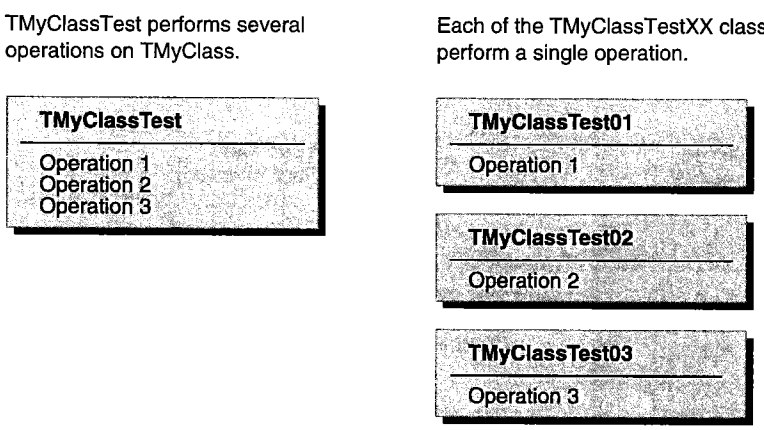


FIGURE 155 EACH OF YOUR TEST CLASSES CAN PERFORM MULTIPLE OPERATIONS

The Test framework makes it possible to organize decision functions into tests with standard interfaces that can be run repeatedly to create regression test suites. Such test suites can be run automatically and don't require special knowledge of the code being tested. The tests return information about the success or failure of the decision functions.

For example, you might find that for an instance of TSampleObject, which contains SetString and GetString functions, you can perform several different operations to test the functions. Each operation is cast as a decision: does the operation work correctly? The outcome of the test operation must be `true` or `false`, indicating the success of the associated test.

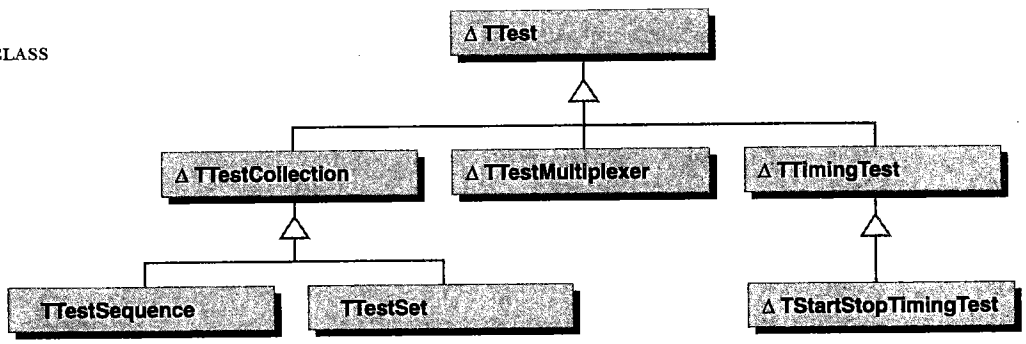
Test framework classes

TTest is an abstract base class. Figure 156 shows some of the TTest-derived classes provided by the Test framework.

TTest includes a Test member function that you must override with the code for a decision function. When you declare a derived class of TTest to be a friend (in C++) of the target class, the derived class can access all the private interfaces of the target class for internal testing purposes.

The sections that follow describe the roles of some of the classes derived from TTest.

FIGURE 156
TEST FRAMEWORK CLASS HIERARCHY



Test collection classes

The Test framework includes derived classes of TTest that allow you to group tests:

- **TTestCollection**, an abstract base class derived from TTest, contains a collection of TTest instances. The tests in the collection run sequentially to determine the success or failure of the entire group. TTestCollection has two concrete derived classes:
  - **TTestSequence** runs its subtests in a fixed sequence.
  - **TTestSet** contains an unordered set of subtests that can be shuffled to vary the order in which the subtests run.
- **TTestMultiplexer** derives from TTest and supports multiple decision functions applied to a single test target. It does so by means of a key-value dictionary in which each value is a decision function and the corresponding key is some text used to identify that function. TTestMultiplexer allows you to fill in the dictionary with appropriate key-value pairs. You can then specify keys from the command line when you run the text to invoke particular decision functions. This makes it possible to rerun a test by using different combinations of decision functions without having to recompile the test.

Timing tests

The Test framework provides two classes for timing tests:

- **TTimingTest** is an abstract base class that provides a basic guide for tests that measure the time a specific operation takes to complete.
- **TStartStopTimingTest** is an abstract base class derived from TTiming Test. It supports stopping and starting the timer several times.

Utility Tests

Utility Tests test an entire hierarchy of classes that share a common protocol. There are five concrete Utility Test classes:

- **TCopyTestOf** tests the copying of the target using the global Copy function.
- **TStreamTestOf** tests the streaming of the target.
- **TFlattenResurrectTestOf** uses the global Flatten and Resurrect functions to test the flattening and resurrecting of the target.
- **TPrimitiveComparisonTestOf** tests canonical member functions: the constructor, destructor, copy constructor, and assignment operator.
- **TWellBehavedObjectTestOf** tests the copying, flattening, and streaming of the target by creating TCopyTestOf, TStreamTestOf, and TFlattenResurrectTestOf tests for the target and running all of them.

Related classes

The Test framework uses several classes derived from TStandardText for buffering and logging purposes. All output from a test is buffered in the test object and, if the test is logged, in the log. TTieredText and TTieredTextBuffer allow you to organize test output in a hierarchy that makes it possible to control the level of detail displayed both during the test and when retrieving test output.

- **TTieredText** is the class of objects collected by TTieredTextBuffer. It is a derived class of TStandardText and allows you to define the relative importance of information in a test, from single-line headlines to debugging details. Once you have designed tiers for particular kinds of output, you can filter the level of information displayed as the test runs and when you retrieve the test output from a test log.
- **TTieredTextBuffer** behaves like the C++ ostream class. It contains << operators for all basic types. Unlike the ostream class, TTieredTextBuffer keeps a collection of all text sent to it. Other features of TTieredText include the following:
  - Echoing of text to a destination you specify
  - Filtering output so that detailed information is suppressed or displayed
  - Flushing text beyond a certain level of detail from the buffer.

  Each instance of TTest contains a TTieredTextBuffer to which derived classes can stream diagnostic text messages. TTest itself uses this mechanism to report progress and results.
- **TTextArgumentDictionary** parses a sequence of TText objects into pairs of keys and values. It allows you to parse the command line easily and quickly look up the arguments used by a particular test.

Creating a test

The simplest way to create a test class is to derive it directly from TTest, as suggested by Figure 157. In this example, TSampleObject is the target of the test.

FIGURE 157
TSAMPLEOBJECTLENGTHTEST CONTAINS CODE FOR TESTING TSAMPLEOBJECT

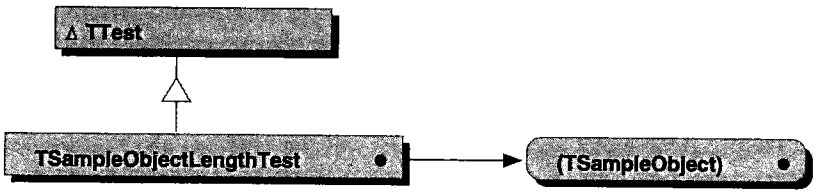
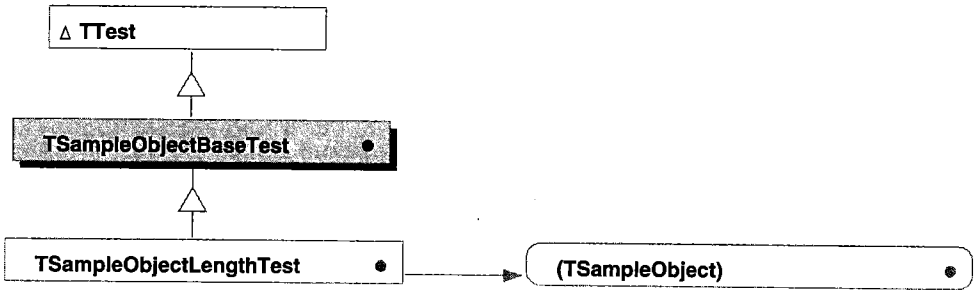


Figure 158 illustrates a more flexible approach that involves creating a base test class that contains common functions and data.

FIGURE 158
TSAMPLEOBJECTBASE CONTAINS CODE COMMON TO ALL TSAMPLEOBJECT TESTS



Each member function to be tested needs an instance of a TTest object, as shown by Figure 159. You can use a simple naming convention to clarify which test classes relate to the classes to be tested, such as TMyClassTest for TMyClass or TMyClassMyMethodTest for TMyClass::MyMethod.

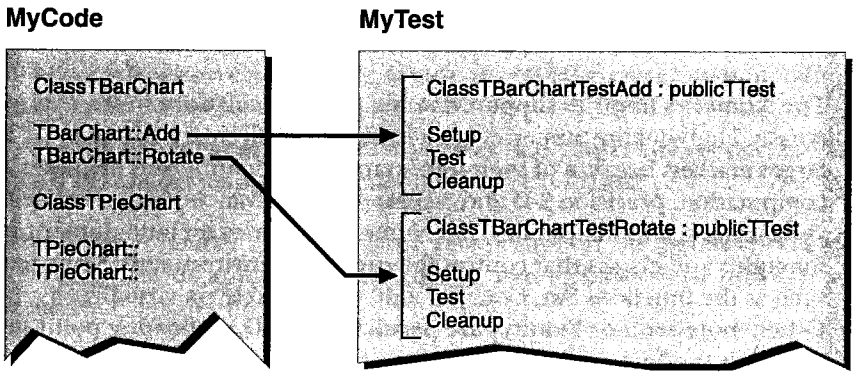


FIGURE 159 EACH MEMBER FUNCTION THAT YOU WANT TO TEST NEEDS AN INSTANCE OF A TTEST OBJECT

Combining tests

You can group tests in three ways:

- Combine tests that need to run together as a group into a suite. Each test suite has a script associated with it that runs all the tests.
- Create a derived class of TTestMultiplexer that contains multiple decision functions that can be applied selectively to a single test target.
- Use the TTestCollection derived classes TTestSet and TTestSequence to group related TTest classes in a single test. The resulting test passes only if all its subtests pass. This approach also allows you to use the same TTest classes individually.

TTestCollection derived classes allow you to:

- Define the order in which subtests execute
- Shuffle the subtests into a new random order
- Propagate inputs from the group to the subtests

You can also create tests that have different behavior depending on the outcome of other tests. For example, suppose TSecondTest works only if the system is in a state that is achieved only if TFirstTest is run and passes. If you place TFirstTest and then TSecondTest inside a TTestSequence, the subtests run in order when the TTestSequence executes. If a subtest fails, the remaining subtests do not run. This behavior can be switched on and off.