Version 0.2 User's Guide

Rygaard Technologies, LLC

26 November, 2018

"Undercamber" and the airfoil logo are trademarks of Rygaard Technologies, LLC.

Rygaard Technologies, LLC reserves the right to make changes to this document and its contained information without notice. This document does not represent any commitment on the part of Rygaard Technologies, LLC.

Rygaard Technologies, LLC makes no warranty of any kind regarding the information in this document. No statement or representation in this document may be deemed a warranty by, or lead to liability of, Rygaard Technologies, LLC. Rygaard Technologies, LLC shall not be held liable for any damages whatsoever resulting from the information in this document.

The Undercamber™ software is licensed under this agreement:

Copyright 2018 Rygaard Technologies, LLC

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
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. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS 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 COPYRIGHT HOLDER OR 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.

Contents

Introduction 5
System Requirements 5
Background 6
Undercamber Goals 6
Two Pass System 6
A Quick Introduction to Programming Tests in Undercamber 7
Quick Example 7
Multipass System 9
Example Configurator 9
Running the Example 9
Test Outcome 12
Test Failure 12
Negative Tests 13
`TestManager.addException(…)` 13
The Sequence of Operations 15
Different JVMs 16
Subtest Execution Sequence 16
Concurrent Execution 17
Specifying Thread Count 18
Mixing Concurrent and Sequential Execution 18
Concurrent Data Access 20
Internal Undercamber Data 20
User-Implemented Data 20
Watchdog Thread 21
Important Notes on Concurrent Tests 21
Completion Callback Listeners 22
Requirements Verification 23
Implementing the `Requirement` Interface 23
Integration with Requirements Tracking Systems 24
Prerequisites 25
Prerequisite Types 27
Previously Satisfied Conditional Prerequisites 29
Prerequisites Must Precede Dependents 30
Forcing Prerequisites 30
Test Uniqueness 31
Test Parameters 32
Specifying Test Parameters 32
Accessing Test Parameters 32
Tags 33
Results Directory 34
Output Root Directory 34
Output Subdirectory 34
User Directory 34
The GUI 35
The Test Selection Window 35
Run Individual Tests 36
Run a Test Set 37
View Test Set Details 38
Selecting Branches 40
Show Dependencies 41
Requirements 42
Tags Column 43
Expand and Collapse Branches 44
Tag List 45
Requirements List 48
Skipping the Selection Window 48
The Results Screen 49
Results Summary 50
Show Errors 51

"Undercamber" and the airfoil logo are trademarks of Rygaard Technologies, LLC

Requirements Results 53
Dependencies 54
Test Details 55
Tags 56
Expand and Collapse Branches 57
View Test Set Details 58
Requirements Tab 60
Unsupportive Tests Tab 61
Tags Window 62
Skipping the Results Window 63
The Configurator 64
The Minimum 64
Output Directory 64
String Expansion in the Configuration 65
Show or Hide the Results Screen 65
Different JVM Versions 66
Java Parameters 66
com.undercamber.Path 67
Test Parameters in the Configurator 68
Setting Environment Variables 70
Important Notes on Environment Variables 70
Thread Pool Size 71
Command Line Arguments 72
Examples 74
Environment Variables 75
A More Complete Example 76
The UUT 76
Top Level Test 77
Addition Checker 77
Division Checker 78
The Configurator 80
Running the Example 80
Legal Notices 81

Introduction

Undercamber was born out of these frustrations with existing test frameworks:

- Byzantine annotations.
- Running a subset of the test suite requires that the tester either mind-meld with the annotations or reprogram the test code.
- Adding, updating, and removing tests requires tuning and re-tuning annotations and/or configuration files.
- There is no clean way to dynamically discover at run-time what set-up needs to be performed.
- Implementing verification strategies not anticipated by the framework is difficult or impossible.
- Testing on multiple versions of the JVM is not well supported.
- The order in which tests are executed is not well defined, even if the tests are run sequentially.
- Long learning curve.

Undercamber™ is a software test automation framework, with these advantages:

- Short learning curve.
- Self-configuring GUI, with no GUI programming, allowing users to graphically select which subset of tests to run.
- Easily select subsets of tests from the command-line, without a GUI.
- Dynamically determine at run-time what setup is required for each test.
- Programmed via a simple Java API, without a new scripting environment, without annotations, and without configuration files.
- Because Undercamber is an API, test developers can implement complex verification strategies in a way that is natural and intuitive to a Java developer.
- Unit tests, regression tests, system tests, and acceptance tests are managed in the same way, in the same framework, with a common configuration.
- The system automatically reconfigures itself as tests are added and removed, without any re-tuning.
- Run tests in multiple versions of the JVM in a single run.
- Sophisticated networks of prerequisites and dependents are easy to implement and maintain.
- Simple, intuitive control of concurrency.
- Predictable and repeatable execution order, as defined by the test developer (except for tests running concurrently).
- Trap most test configuration issues at compile time.
- Trap all remaining test configuration issues immediately on launch.

This document assumes the reader is familiar with the Java programming language. Understanding the discussion and examples requires that the reader previously understand Java concepts, including language features added in Java Version 8.

System Requirements

Undercamber™ will run on any machine with Java SE Version 8 or later installed. Undercamber is 100% Java, and does not use any exotic capabilities, meaning it should run on a wide variety of platforms. It has been tested on Linux, Macintosh, and Windows.

Background

Undercamber was developed to address the needs of automated software testing in an agile environment. Some of the issues it addresses include:

- In an agile environment, a developer is typically working on just one aspect of the product while debugging or adding features. To support this, the developer needs just a small portion of the full test suite to check the development at hand. Running the entire regression test suite typically would be overkill and could be much too slow.
- An agile team (or any other team) should not spend time supporting the test framework. A test framework should self-configure as tests are updated, removed and added, instead of requiring the developers to update scripts, annotations, and configurations to support the tests.
- In a customer-driven environment, the application requirements can change over time. This in turn means the test suite must support quick refactoring with minimal effort.
- A regression test suite can take a very long time to complete, so running separate tests concurrently in separate threads can accelerate the testing.
- The test system needs to be tightly integrated into the development tools and the development workflow, so that programmers can remain focused on their tasks at hand, without jumping between environments and without learning a new environment.

Undercamber Goals

- Different development team members can easily run different sections of the tests in an ad-hoc fashion without reprogramming.
- Easily set up and manage a sophisticated and dynamic network of prerequisites and dependents, so that the test suite automatically discovers at run time what setup should be performed prior to running the verification tests.
- Support development tests, unit tests, regression tests, system tests, and acceptance tests in the same framework, with the same code.
- Support testing on multiple versions of the JVM in a single run.
- The test configuration and execution should be managed programmatically at run-time without annotations and without scripts. This gives test developers the full flexibility of the Java programming language to implement whatever test techniques and strategies might be needed.
- The test configuration should be easy to set up and require minimal maintenance, even while the test suite evolves and grows.
- Work in Java. This is the environment with which Java developers are most familiar. It also allows the test system to be tightly integrated into IDEs, development environments, and build systems.
- Exploit Java language features to trap most testing problems at compile-time, and trap all remaining test configuration problems immediately at startup.
- Easily set up a multi-threaded system to accelerate the test suite, and to test the thread safety of the application under test.

To achieve these goals, these features were designed into Undercamber:

- A GUI with a selection window to choose which tests to run.
- Hierarchical test structure provides natural grouping so that sets and subsets of tests can be easily selected in the selection window.
- The selection window is driven by the test source code, so that:
    - There is no need to re-program a GUI when tests are added, removed, or modified,
    - There is no need to update configuration files, scripts, or annotations as tests are added, removed, or modified.
- Easily run any subset of the tests from the command line, without any GUI.
- Easily set up and maintain a complex and dynamic network of prerequisites.
- A simple API that requires very little learning curve.
- A simple test flow that is easy to learn and maintain.
- An intuitive GUI to displays the results.

Two Pass System

To achieve the goals outlined above, Undercamber takes a unique approach to testing: It is a two-pass system. The first pass runs very quickly, and Undercamber uses this pass to discover the test structure, then uses this information to drive the selection window (if it is displayed). On the second pass, the selected tests are executed.

A Quick Introduction to Programming Tests in Undercamber

From Undercamber's point of view, each test implements the `TestUnit` interface:

```
package com.undercamber;

public interface TestUnit
{
   void runTest( TestManager testManager )
      throws Throwable;
}
```

In practice, most tests will not explicitly implement this interface. Instead, `TestUnit` is a functional interface for use in Lambda expressions.

Quick Example

In order to illustrate a test example, consider this "application" for testing:

```
package com.undercamber.codesamples.intro;

public class MyBank
{
   private java.util.Map<String,Integer> _accounts;

   public MyBank()
   {
      _accounts = new java.util.HashMap<String,Integer>();
   }

   public void createAccount( String name )
      throws Throwable
   {
      if ( _accounts.get(name) != null )
      {
         throw new IllegalArgumentException( "Account \"" + name + "\" already exists" );
      }

      _accounts.put( name, 0 );
   }

   public int deposit( String name,
                       int    amount )
      throws Throwable
   {
      Integer currentBalance;

      currentBalance = _accounts.get( name );

      if ( currentBalance == null )
      {
         throw new IllegalArgumentException( "Account \"" + name + "\" not found" );
      }

      if ( (currentBalance+amount) < 0 )
      {
         throw new IllegalArgumentException( "Overdraft" );
      }

      _accounts.put( name, (currentBalance+amount) );

      return currentBalance + amount;
   }
}
```

This illustrates a basic test system in Undercamber for the above “application”:

```
package com.undercamber.codesamples.intro;

import com.undercamber.*;

public class MyTests implements TestUnit
{
   public void runTest( TestManager testManager )
      throws Throwable
   {
      boolean verify;
      MyBank  bank;

      verify = testManager.initialize();
      if ( verify )
      {
         bank = new MyBank();
      }
      else
      {
         bank = null;
      }
      testManager.addSubtest( tm -> testAccountCreation(tm,bank) );
      testManager.addSubtest( tm -> testDeposit(tm,bank) );
   }

   void testAccountCreation( TestManager testManager,
                             MyBank      bank )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         bank.createAccount( "Bill" );
      }
   }

   void testDeposit( TestManager testManager,
                     MyBank      bank )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         bank.deposit( "Bill", 100 );
      }
      testManager.addSubtest( tm -> testWithdraw(tm,bank) );
   }

   void testWithdraw( TestManager testManager,
                      MyBank      bank )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         bank.deposit( "Bill", -50 );
      }
   }
}
```

Notice these aspects of the tests:

- Only the top-level test must explicitly implement the `TestUnit` interface.
- The top-level test must have a valid no-argument constructor.
- Each test is passed an instance of `TestManager`.

- Each test must call `TestManager.initialize()` (or one of its overloads) exactly once.
- Each test conditionally runs its verification checks.
- Tests can add subtests by calling `TestManager.addSubtest(…)`.

Multipass System

Undercamber is a two-pass test system. In the first pass, Undercamber only discovers the test structure without performing any actual verification, and uses this information to build the selection window. In the second pass, Undercamber reruns the tests with verifications.

In the first pass, `TestManager.initialize()` (or one of its overloads) returns `false`. Review the above code: In the first pass, the actual verifications are not performed.

In the second pass, `TestManager.initialize()` (or one of its overloads) returns `true`. Review the above code: The verifications are run in the second pass.

Example Configurator

In Undercamber a configuration class describes the top-level test. For the above example, this could be used:

```
package com.undercamber.codesamples.intro;

public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   final public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "BankTestExample" );
      configurator.setResultsRootDirectoryName( "C:\\UndercamberDemos" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();
      testSetBuilder.setTestSetName( "MyTests" );
      testSetBuilder.setClass( MyTests.class );
      testSetBuilder.createTestSet();
   }
}
```

As illustrated above, the configuration class must implement `com.undercamber.ConfigurationCallback`. The example configuration above specifies:

- The test suite is named “BankTestExample”.
- Undercamber will work in the `C:\UndercamberDemos` directory. By default, Undercamber will create a subdirectory below this directory, with a name based on the date and time. Update this line of the configuration class as needed for your particular environment (More on this later, in “String Expansion in the Configuration”, under “The Configurator”, below).
- There is one test set. There can be more than one test set, as described below under “The Configurator”.
- The test set starts with the `com.undercamber.codesamples.intro.MyTests` class.
- The test set is named “MyTests”. Undercamber uses this name to create persistence files.

A few other details can be specified in the configuration class, as described below under “The Configurator”.

In Undercamber, the configuration can be set up quickly and requires very little maintenance.

Running the Example

To run the above example:

1. Make sure Java 8 or later is installed and in your path.
2. Make sure the Undercamber library file is in your classpath (either the JAR file or the class files).
3. Enter the source code above into `MyBank.java`, `MyTests.java`, and `ConfigurationCallback.java`, in an appropriate subdirectory below the classpath. Compile them.
4. At a command line, type:

```
java com.undercamber.Undercamber -config com.undercamber.codesamples.intro.ConfigurationCallback
```

5. Undercamber will display the selection screen:

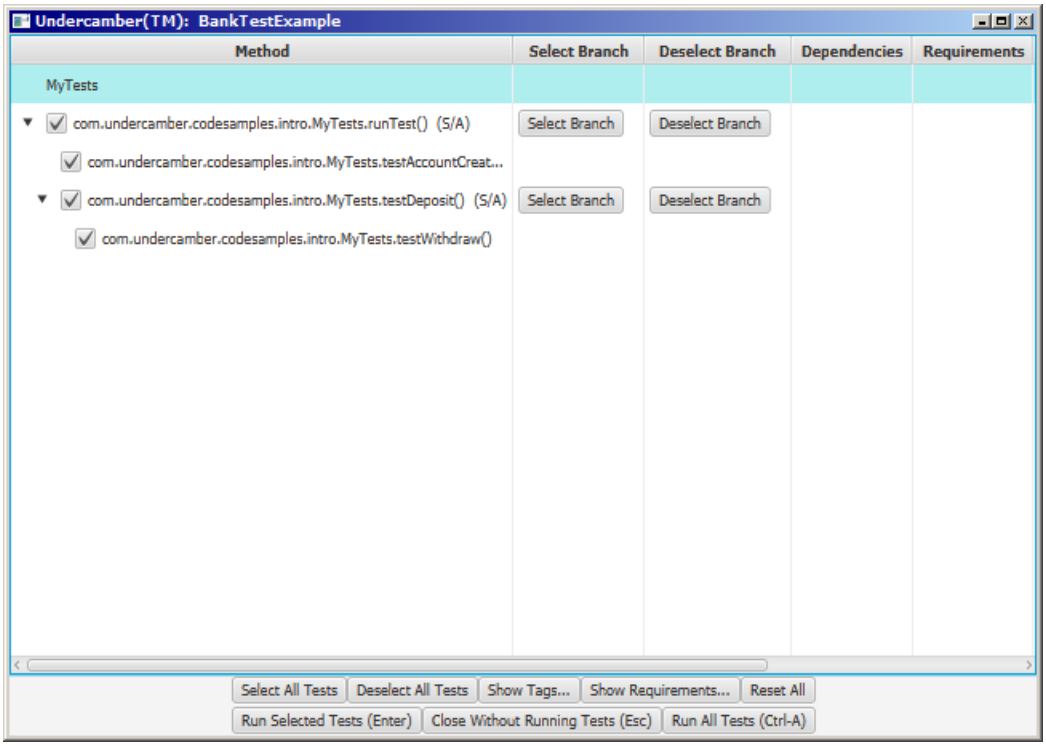

Figure 1: The Undercamber Selection Window

Notice that the tree of tests in the GUI is automatically configured according to the hierarchy specified in the calls to `TestManager.addSubtest(…)`. This screen is described in more detail in "The GUI", below. For now, feel free to explore; it won't 'break' anything.

6. Select which subtests to run.
7. Press `Enter`.
8. Undercamber will run the selected tests.
9. Undercamber will create these reports:

File	Description
`FinalReport.txt`	A report showing the results of the tests that were executed.
`FullReport.txt`	A report showing the results of all of the tests.
`Summary.txt`	A brief report showing the test outcome.
`Analysis.txt`	A report showing possible problems with the test setup.
`TestReport.xml`	An XML file fully describing the test and its results.
`TestReport.xsd`	The XML schema for `TestReport.xml`.

These files will be written to a subdirectory below the `C:\UndercamberDemos` directory (which is specified by the configurator as shown above). Undercamber will generate the subdirectory name from the current date and time.

If Undercamber detects deadlocks it will also create informational files in the `Deadlock` subdirectory, as described below in "Watchdog Thread".

10. Undercamber shows the results window:

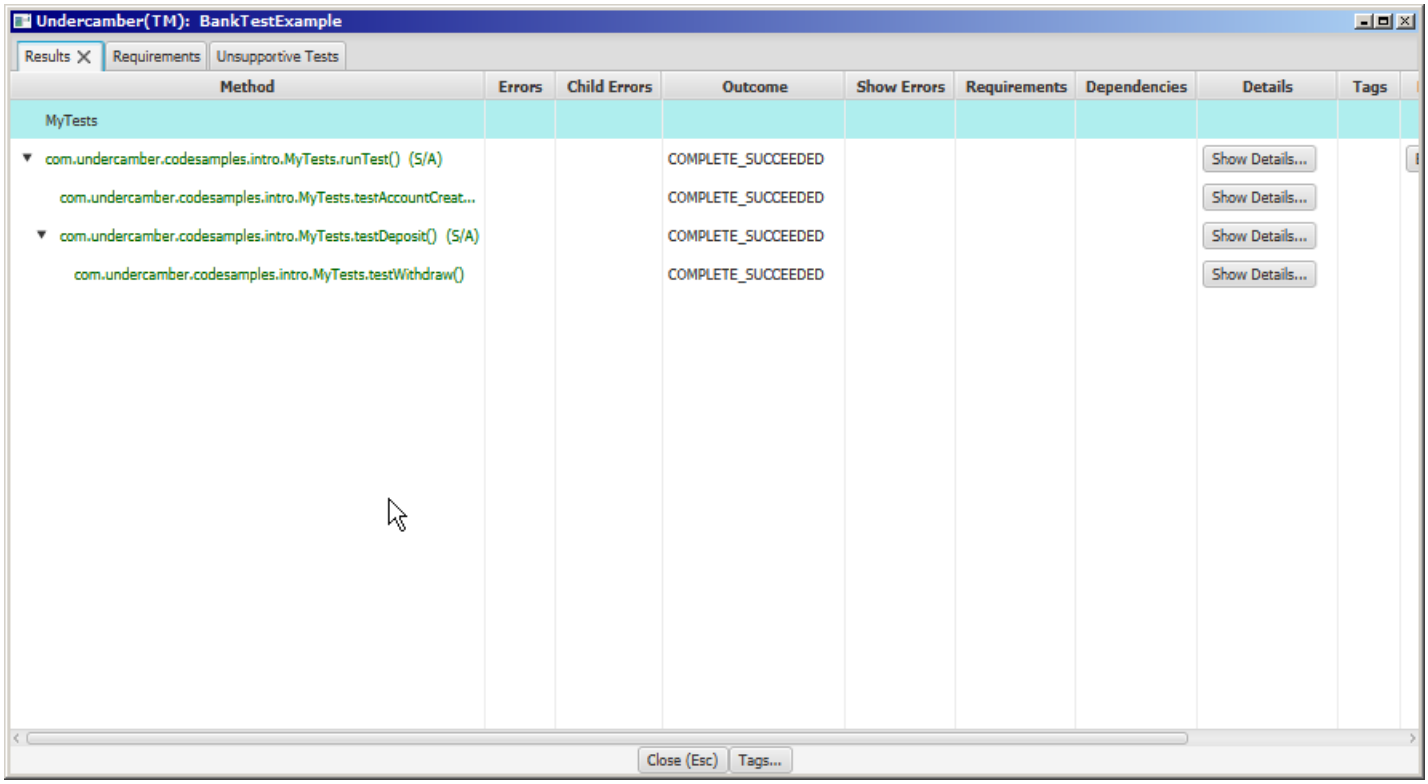

Figure 2: The Undercamber Results Window

This screen is described in more detail in "The GUI", below. For now, feel free to explore; it won't 'break' anything.

11. Press the escape key to close Undercamber.

Note: Undercamber creates a subdirectory named `.Undercamber` in the user's home directory. This directory is used for persistence so that the user's GUI choices are remembered between runs. This directory is quietly managed by Undercamber, so test developers and end users do not need to interact with it. If this directory becomes corrupted for some reason, Undercamber will re-build it using defaults.

The example in "A More Complete Example", below, illustrates a more sophisticated example.

Test Outcome

If a test runs without any exceptions, Undercamber assumes the test passed. If an exception is thrown, Undercamber assumes the test failed.

Each test will have one of several outcomes (enumerated in the `TestState` enum):

Outcome	Description
NOT_RUN	Test did not run.
UNINITIALIZED	The test failed to call `TestManager.initialize(…)`.
INITIALIZED	Initialized but terminated due to an unexpected error. This should not occur under normal operation.
RUNNING_SUBTESTS	Unexpected error while waiting for subtests to complete. This should not occur under normal operation.
SKIPPED_DUE_TO_PARENT_ERROR	Skipped because an ancestor failed. When a test fails, its subtests might or might not run, depending on the configuration.
SKIPPED_DUE_TO_PREREQUISITE_ERROR	Skipped because one or more prerequisites failed.
SKIPPED_DUE_TO_SIBLING_ERROR	Skipped because the parent test specified `SubtestSequencingMode.SEQUENTIAL_ABORT_SEQUENCE_ON_ERROR` and a prior sibling test failed. This is described in more detail below in "Concurrent Execution".
SKIPPED_BY_USER	Skipped because the user did not request it.
COMPLETE_SUCCEEDED	Ran to completion without any exceptions.
COMPLETE_FAILED	Ran to completion, but one or more exceptions were encountered.

The test outcome is reported to:

- The command line during execution,
- The results window after execution (if it is displayed), and
- The `FinalReport.txt` and the `TestReport.xml` files.

Test Failure

Any test that throws an exception is considered a failing test. Consider this example:

```
void runTest( TestManager testManager )
   throws Throwable
{
   boolean verify;
   int     result;

   verify = testManager.initialize();

   if ( verify )
   {
      try ( FileInputStream fileInputStream = new FileInputStream("InputData.dat") )
      {
         try ( DataInputStream dataInputStream = new DataInputStream(fileInputStream) )
         {
            result = dataInputStream.readInt();
            if ( result != 10 )
            {
               throw new Exception( "Incorrect result" );
            }
         }
      }
   }
}
```

This example will be considered a failing test if any of the following occur:

- The file `InputData.dat` does not exist,
- The user does not have read access to `InputData.dat`,
- The file `InputData.dat` is less than 4 bytes long (the size of an `int`), or
- The value in `InputData.dat` is not 10.

If the test does not throw any exceptions (and does not call `TestManager.addException(…)` as described below), the test is considered a success.

Negative Tests

Consider the case where the correct behavior of the UUT (Unit Under Test) is to throw an exception. This can be verified using an appropriate try/catch block:

```
void myTest( TestManager testManager,
             BankAccount account )
   throws Throwable
{
   boolean verify;

   verify = testManager.initialize();

   if ( verify )
   {
      try
      {
         account.withdraw( 1000000 );
         throw new Exception( "Did not get expected OverdraftException" );
      }
      catch ( OverdraftException expected )
      {
      }
   }
}
```

In this example, if the call to `BankAccount.withdraw(…)` throws an `OverDraftException`, the test passes. Similarly, if the call to `BankAccount.withdraw(…)` does not throw an exception, the test fails.

`TestManager.addException(…)`

Undercamber maintains a list of errors for each test, and a test is considered a failure if there is one or more entries in the list of errors. If a test throws an exception, the thrown exception is appended to the list. To add an exception to the list without throwing it, call `TestManager.addException( Throwable )`.

In the above example, the test aborts if the negative test fails. To continue testing after a verification fails, use `TestManager.addException(…)`:

```
void myTest( TestManager testManager,
             BankAccount account )
   throws Throwable
{
   boolean verify;

   verify = testManager.initialize();

   if ( verify )
   {
      try
      {
         Account.withdraw( 1000000 );
         testManager.addException( new Exception("Did not get expected OverdraftException") );
      }
      catch ( OverdraftException expected )
      {
      }
   }
}
```

`TestManager.addException(…)` can be used anywhere the test should not be aborted when a verification fails:

```
void myTest( TestManager testManager,
             Bank        bank )
   throws Throwable
{
   boolean verify;
   Account account;
   int     balance;

   verify = testManager.initialize();

   if ( verify )
   {
      try
      {
         bank.createAccount( "Hugh G. Kwazion" );
      }
      catch ( IllegalArgumentException namingConflict )
      {
         testManager.addException( namingConflict );
      }

      try
      {
         balance = bank.deposit( "Hugh G. Kwazion", 100 );
      }
      catch ( Exception exception )
      {
         testManager.addException( exception );
      }

      if ( balance != 100 )
      {
         testManager.addException( new Exception("Incorrect balance") );
      }

      try
      {
         bank.deposit( "Hugh G. Kwazion", -150 );
         testManager.addException( new Exception("Did not get expected overdraft") );
      }
      catch ( IllegalArgumentException expected )
      {
      }
   }
}
```

The example in “A More Complete Example”, below, contains other examples of calls to `TestManager.addException(…)`.

The Sequence of Operations

This illustrates the sequence of activities in a single Undercamber run:

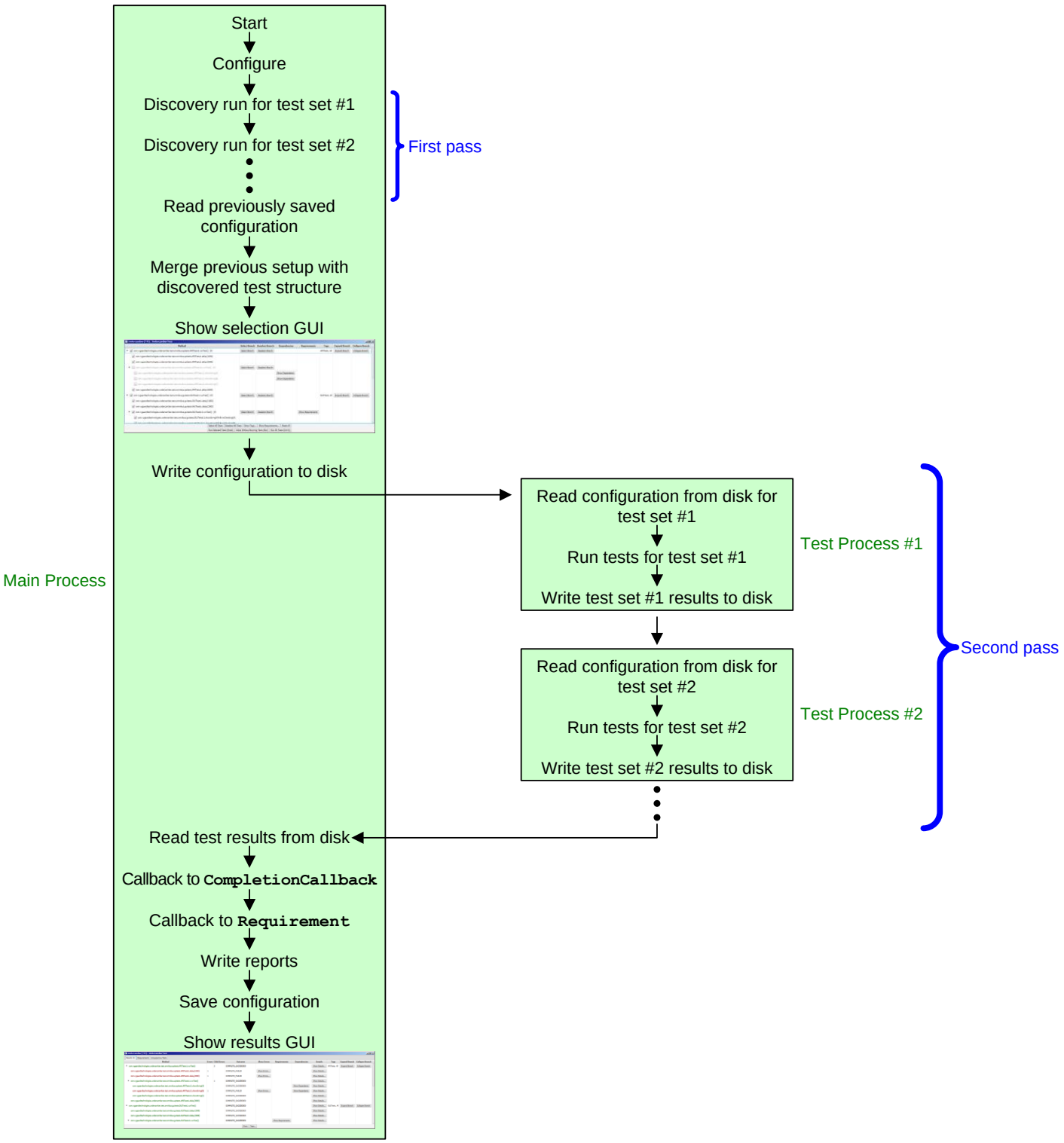

Figure 3: Sequence of Operations

Undercamber makes 2 passes: The first pass discovers the test structure to drive the selection window, without running any verifications. The second pass runs the actual verifications.

There can be one or more test sets. During the second pass, when the actual verifications are run, each test set is run in a separate JVM, in a separate process. This scheme allows different sets of tests to run in different environments, such as a different JVM version and/or different classpath for each test set. The test sets are specified by the Undercamber™ configuration, as described below in "The Configurator".

During the first pass, all of the tests are run concurrently, using as many threads as are available in the thread pool. During the second pass, each test is run sequentially or concurrently, as defined by the test developer (See "Concurrent Execution", below).

Different JVMs

During the second pass, the tests can be run in different versions of the JVM, as specified by the configurator. A test developer should understand some of the consequences of using different versions of the JVM in different test sets.

In the first pass, the default JVM (whatever JVM is used to invoke Undercamber) is used. During this pass, all tests from all test sets are called, even though no verification is performed. All of these calls are made in the same JVM. This implies that all of the classes will be loaded into the default JVM.

Loading all of the classes into the default JVM can lead to difficulties. In particular, if some test sets have features that are not supported by the default JVM, then those classes might not load during the first pass.

The best strategy to manage this issue is to:

1. Of the JVMs used during the second pass, select the latest version JVM.
2. Use the selected JVM to launch Undercamber.

To configure Undercamber to use different JVMs, see "Different JVM Versions", below, under "The Configurator".

Subtest Execution Sequence

Undercamber always runs subtests after the parent tests have completed. The call to `TestManager.addSubtest(…)` does not execute the subtest immediately; it only queues the subtest for later execution.

Consider this code fragment:

```
public void runTest( TestManager testManager )
   throws Throwable
{
   boolean verify;

   testManager.addSubtest( tm -> subtest1(tm) );
   testManager.addSubtest( tm -> subtest2(tm) );

   verify = testManager.initialize();
   if ( verify )
   {
      new FileOutputStream( "Data.dat" );
   }
}
```

In this example, the subtests will be executed *after* the file is created, even though the calls to `TestManager.addSubtest(…)` precede the verification checks.

Concurrent Execution

Running tests concurrently is easy in Undercamber. `TestManager.initialize(…)` has overloaded versions that can specify whether the subtests are run sequentially or concurrently during the second pass:

```
public class MyTests2
   implements com.undercamber.TestUnit
{
   public void runTest( com.undercamber.TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize( com.undercamber.SubtestSequencingMode.CONCURRENT );
      if ( verify )
      {
         // Perform checks here
      }
      testManager.addSubtest( tm -> mySubtest1(tm) );
      testManager.addSubtest( tm -> mySubtest2(tm) );
   }

   void mySubtest1( TestManager com.undercamber.testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         // Perform checks here
      }
   }

   void mySubtest2( com.undercamber.TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         // Perform checks here
      }

      testManager.addSubtest( tm -> mySubtest21(tm) );
      testManager.addSubtest( tm -> mySubtest22(tm) );
   }

   void mySubtest21( com.undercamber.TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         // Perform checks here
      }
   }

   void mySubtest22( com.undercamber.TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         // Perform checks here
      }
   }
}
```

In this example, `mySubtest1` and `mySubtest2` are run concurrently. `mySubtest21` and `mySubtest22` are run sequentially because subtests are run sequentially by default.

There are three choices for subtest sequencing (enumerated in the `SubtestSequencingMode` enum):

Mode	Description
`CONCURRENT`	Run the subtests concurrently.
`SEQUENTIAL_ABORT_SEQUENCE_ON_ERROR`	Run the subtests sequentially. If one of the subtests fails, do not run the remaining subtests.
`SEQUENTIAL_CONTINUE_ON_ERROR`	Run the subtests sequentially. Continue running subtests if one fails.

If the concurrency is not specified in the call to `TestManager.initialize(…)`, Undercamber will default to `SubtestSequencingMode.SEQUENTIAL_ABORT_SEQUENCE_ON_ERROR`.

Specifying Thread Count

Undercamber uses a thread pool for concurrent tests. There are several ways to specify the size of the Undercamber thread pool:

1. On the command line, using the `-threadCount` flag. See "Command Line Arguments", below.
2. In the configurator, by specifying the thread count in a call to `Configurator.setPass1ThreadCount(…)`. See "Thread Pool Size", below, under "The Configurator".
3. In the configurator, by specifying the thread count in a call to `TestSetBuilder.setPass2ThreadCount(…)`. See "Thread Pool Size", below, under "The Configurator".
4. In the `UNDERCAMBER_THREAD_COUNT` environment variable. Set this variable to the number of threads Undercamber should have in its thread pool.
5. If the thread pool size is not specified by one of the above methods, Undercamber will use the number of processor cores available to the JVM.

If the thread pool size for the first pass (the discovery pass) is specified multiple ways, Undercamber uses this search precedence:

1. The command line, then
2. The value specified in `Configurator.setPass1ThreadCount(…)`, then
3. The `UNDERCAMBER_THREAD_COUNT` environment variable, then
4. The number of available processor cores.

If the thread pool size for a second pass is specified multiple ways, Undercamber uses this search precedence:

1. The command line, then
2. The value specified in `TestSetBuilder.setPass2ThreadCount(…)`, then
3. The value specified in `Configurator.setPass1ThreadCount(…)`, then
4. The `UNDERCAMBER_THREAD_COUNT` environment variable, then
5. The number of available processor cores.

Mixing Concurrent and Sequential Execution

As illustrated above, it is easy to mix sequential and concurrent execution. In a sequentially executed series of tests, each test is held until all pending concurrent tests are completed, in a rendezvous technique.

For example, consider this system:

```
public class MixedConcurrency
   implements com.undercamber.TestUnit
{
   public void runTest( com.undercamber.TestManager testManager )
      throws Throwable
   {
      testManager.initialize();
      testManager.addSubtest( tm -> subtest1(tm) );
      testManager.addSubtest( tm -> subtest2(tm) );
   }

   private void subtest1( com.undercamber.TestManager testManager )
      throws Throwable
   {
      testManager.initialize( com.undercamber.SubtestSequencingMode.CONCURRENT );
      testManager.addSubtest( tm -> subtest11(tm) );
      testManager.addSubtest( tm -> subtest12(tm) );
   }

   private void subtest11( com.undercamber.TestManager testManager )
      throws Throwable
   {
      testManager.initialize();
   }

   private void subtest12( com.undercamber.TestManager testManager )
      throws Throwable
   {
      testManager.initialize( com.undercamber.SubtestSequencingMode.CONCURRENT );
      testManager.addSubtest( tm -> subtest121(tm) );
      testManager.addSubtest( tm -> subtest122(tm) );
   }

   private void subtest121( com.undercamber.TestManager testManager )
      throws Throwable
   {
      testManager.initialize();
   }

   private void subtest122( com.undercamber.TestManager testManager )
      throws Throwable
   {
      testManager.initialize();
   }

   private void subtest2( com.undercamber.TestManager testManager )
      throws Throwable
   {
      testManager.initialize();
   }
}
```

This illustrates the sequence of execution for the above example:

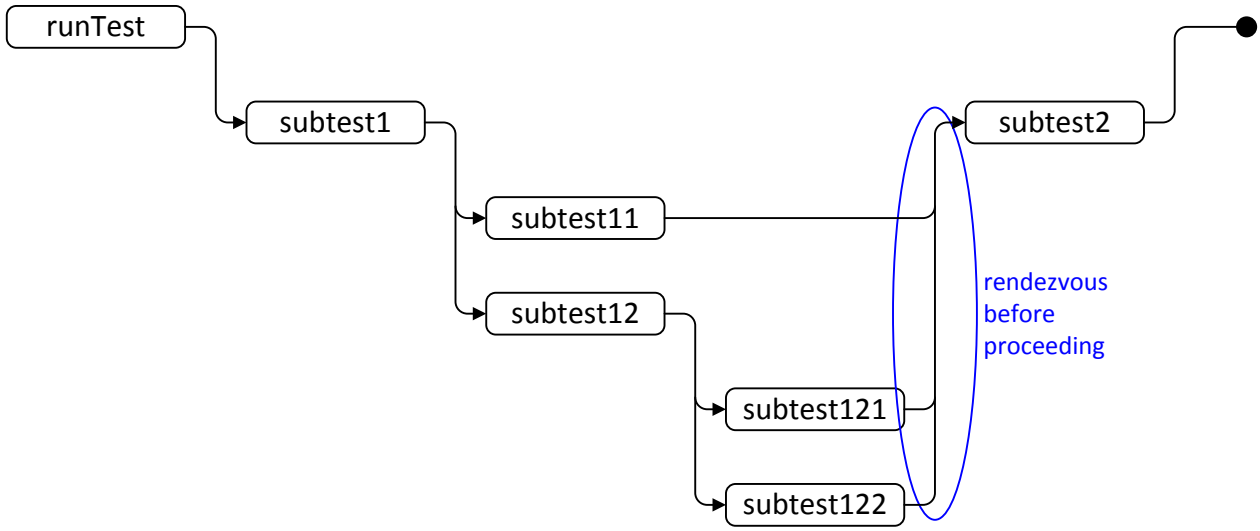

Figure 4: Execution Path

As illustrated above:

- `subtest1` and `subtest2` are executed sequentially,
- `subtest121` and `subtest122` are executed concurrently,
- `subtest12` and the `subtest121`/`subtest122` path are executed sequentially,
- `subtest11` is executed concurrently with the `subtest12`/`subtest121`/`subtest122` path, and
- `subtest2` is not executed until all of `subtest11`, `subtest121`, and `subtest122` have completed.

In the above example, if `subtest11`, `subtest12`, `subtest121`, or `subtest122` fails, then `subtest2` will still be executed. (This behavior can be changed using `Prerequisites`, as described below in "Prerequisites").

Concurrent Data Access

Internal Undercamber Data

Undercamber itself is thread safe. A test developer can call into the Undercamber API from multiple threads without worrying about corrupting Undercamber's internal data (of course, all of the other restrictions and requirements described in this document still apply).

Undercamber protects all of its internal data with exactly one monitor. When making callbacks into user code (such as with `Requirement` or `CompletionCallback`, as described below), Undercamber is not holding its internal monitor. This ensures Undercamber cannot contribute to a deadlock.

Because Undercamber is thread safe, its API can be called from threads not created by Undercamber. This can be useful for testing callbacks from a multi-threaded UUT.

User-Implemented Data

Undercamber does not protect any data implemented by test developers from corruption due to concurrent access.

A common use case for Undercamber's multithreading facility is to concurrently run unrelated tests, to accelerate the tests. In this case, data corruption is not a concern.

Because Undercamber is an API, a test developer can use the multithreading facility for other use cases, such as testing the UUT for thread safety. In these situations, the test developer will need to assess data corruption on a case-by-case basis and handle it accordingly. If data corruption is a concern, it can be addressed using the various concurrency facilities in the standard JVM or in third-party libraries.

When implementing techniques to protect data from concurrent access, the test developer should keep in mind that Undercamber uses a thread pool. Techniques implemented by the test developer should be carefully designed to ensure there is no starvation of threads.

Watchdog Thread

In addition to the test threads managed by Undercamber's internal thread pool, Undercamber also launches a watchdog thread to look for deadlocks. It runs in this infinite loop:

1. Call `java.lang.management.ThreadMXBean.findDeadlockedThreads()`.
2. If deadlocked threads are detected in the previous step:
    1. Display a deadlock report to `System.out`.
    2. If necessary, create the deadlock reporting file. This file is in the Undercamber results directory, in the "Deadlock" subdirectory. The name of the file is the name of the test set, followed by "`.txt`".
    3. Append a report to the deadlock reporting file.
3. Sleep for 10 seconds.
4. Return to Step 1.

Important Notes on Concurrent Tests

- If a series of tests are scheduled to run sequentially, using either `SEQUENTIAL_ABORT_SEQUENCE_ON_ERROR` or `SEQUENTIAL_CONTINUE_ON_ERROR`, the tests might not run on the same thread. In a sequential series, a test will not be launched until its prior test completes, even though they might be on different threads.
- Threads waiting for a rendezvous, as described above in "Mixing Concurrent and Sequential Execution", are made available to the thread pool for running other tests. This helps ensure liveness of the Undercamber system.
- During the first pass (the discovery pass), Undercamber runs all tests as if `SubtestSequencingMode.CONCURRENT` were passed to every call to `TestManager.initialize(…)`, and the sequencing actually specified in the calls to `TestManager.initialize(…)` is ignored. This helps ensure the discovery phase is quick. The second pass is run according the sequencing specified by the test developer in the calls to `TestManager.initialize(…)`.
- When Undercamber traps an exception (either by a thrown exception or by a call to `TestManager.addException(…)`), its stack trace will be automatically written to `java.lang.System.out`. Unlike a normal call to `printStackTrace()`, the stack trace will be contiguous on the screen, even if multiple threads are writing stack traces to the screen. The stack traces will not be interleaved with other output and with other stack traces.

Completion Callback Listeners

Undercamber has a mechanism to notify listeners when all tests are complete. A completion callback listener should implement the `CompletionCallback` interface:

```
package com.undercamber;

public interface CompletionCallback
{
   void testComplete( java.util.List<TestData> testDataRoots );
}
```

To create a completion listener, create a listener that implements the `CompletionCallback` interface, then register it with `TestManager.addCompletionCallback(…)`. After all of the tests are complete, Undercamber will call all registered listeners.

This callback can be used for any purpose, including:

- Recording results in a custom format.
- Establishing a connection to a requirements database.
- Consolidating test results for requirements that are validated by multiple tests.
- Integration into third-party tools.

When developing `CompletionCallback` listeners, the developer should be aware of its lifecycle:

1. During the first pass:
    - Undercamber captures and retains the `CompletionCallback` objects passed to `TestManager.addCompletionCallback(…)`.
2. During the second pass:
    - Undercamber captures and records the test results.
    - Undercamber ignores the `CompletionCallback` objects passed to `TestManager.addCompletionCallback(…)` since they were saved in the first pass.
3. After all tests are run, Undercamber calls the `CompletionCallback.testComplete( TestData )` listeners:
    - `testComplete` is called on the `CompletionCallback` objects retained from the first pass.
    - The argument to `CompletionCallback.testComplete( TestData )` contains the results recorded in the second pass.
    - The listeners will be called in the same order in which they were registered.

This lifecycle is illustrated in “Figure 3: Sequence of Operations”, above.

Note that if concurrently running tests register callbacks, then there might be uncertainty in the order in which the callbacks are registered and notified.

Requirements Verification

A test developer can reference requirements by implementing the `Requirement` interface:

```
package com.undercamber;

public interface Requirement
   extends Comparable<Requirement>
{
   String getRequirementID();
   void testComplete( TestData testData );
   TestState getCompletionState();
   String getDescription();
   String getResultsText();
   default int compareTo( Requirement that )
   {
      return getRequirementID().compareTo( that.getRequirementID() );
   }
}
```

Undercamber provides a very basic implementation of the `Requirement` interface in `SimpleRequirement`, as described in the JavaDocs.

In Undercamber, a test can declare that it verifies a requirement in the call to `TestManager.initialize(…)`. This code fragment shows how a test declares that it verifies a requirement:

```
void myTest( TestManager testManager )
   throws Throwable
{
   boolean verify;

   verify = testManager.initialize( new SimpleRequirement("1.2","Graceful failure") );
   if ( verify )
   {
      // Perform checks here
   }
}
```

After running tests, Undercamber generates a text report and an XML document describing the verification results for each test, and presents the verification results in the results GUI (if it is displayed).

Implementing the `Requirement` Interface

When developing implementations of the `Requirement` interface, the developer should be aware of how Undercamber handles Requirements:

1. During the first pass:
   - Undercamber captures and retains the `Requirement` objects passed to `TestManager.initialize(...)`.
   - Undercamber captures and retains the `CompletionCallback` objects passed to `TestManager.addCompletionCallback(…)`.
2. During the second pass:
   - Undercamber captures and records the test results.
   - Undercamber ignores the `Requirement` objects passed to `TestManager.initialize(…)` since they were saved in the first pass.
   - Undercamber ignores the `CompletionCallback` objects passed to `TestManager.addCompletionCallback(…)` since they were saved in the first pass.
3. After the tests are run, Undercamber notifies all `CompletionCallback` objects previously passed to `TestManager.addCompletionCallback(…)`. This can be used, for example, to set up connections to a requirements database.
4. After the completion callbacks, Undercamber calls the `Requirement.testComplete( TestData )` callbacks:
   - `testComplete` is called on the `Requirement` objects retained from the first pass.
   - The argument to `Requirement.testComplete(…)` contains the results recorded in the second pass.

In order to make the discovery pass (the first pass) run quickly, it is recommended that any time-consuming tasks in `Requirement` objects be deferred until the calls to either `CompletionCallback.testComplete(…)` or `Requirement.testComplete(…)`.

Integration with Requirements Tracking Systems

There are two mechanisms for integrating Undercamber with third-party requirements tracking systems:

1. Undercamber generates an XML file, `TestReport.xml`, and its schema file, `TestReport.xsd`, that fully describe the tests, the test results, and the requirements. This can be parsed to discover the test results.
2. A developer can develop a custom implementation of the `Requirement` interface. If needed, this can be supported by implementations of the `CompletionCallback` interface, as described above in "Completion Callback Listeners".

Prerequisites

It is easy to set up and maintain a complex and dynamic structure of prerequisites and dependents in Undercamber.

Any test can have prerequisites, which must be successfully run in advance. In Undercamber, a prerequisite is just another test; the only difference is that it must be completed prior to running its dependent tests.

In Undercamber:

- Because a prerequisite is just another test, it can, in turn, have its own prerequisites.
- Prior to running a test, Undercamber will run all of that test's prerequisites. If a prerequisite fails, any dependent tests will not be run.
- When a test is selected in the selection window, all of its prerequisites will also be automatically selected.
- When a test is deselected in the selection window, all of its dependent tests will also be automatically deselected.

Undercamber supports both explicit and implicit prerequisites:

- A test's parent is implicitly considered a prerequisite. By recursion, a test's parent's parent is also implicitly considered a prerequisite, and all ancestors are implicitly considered prerequisites.
- If a parent test specifies that the subtests are run with `SubtestSequencingMode.SEQUENTIAL_ABORT_SEQUENCE_ON_ERROR` (either explicitly or by default), then each subtest implicitly has its prior 'siblings' as prerequisites. `SubtestSequencingMode` is described above in "Concurrent Execution".
- A prerequisite can be explicitly specified in the call to `TestManager.initialize(…)`.

To explicitly specify a prerequisite, create a `Prerequisite` object and pass it to the `TestManager.initialize(…)` method:

```
import com.undercamber.*;

public class MyTests3
   implements TestUnit
{
   public void runTest( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         // Perform checks here
      }
      testManager.addSubtest( tm -> mySubtest1(tm) );
      testManager.addSubtest( tm -> mySubtest2(tm) );
   }

   void mySubtest1( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         // Perform checks here
      }
   }

   void mySubtest2( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize( SubtestSequencingMode.SEQUENTIAL_ABORT_SEQUENCE_ON_ERROR);
      if ( verify )
      {
         // Perform checks here
      }

      testManager.addSubtest( tm -> mySubtest21(tm) );
      testManager.addSubtest( tm -> mySubtest22(tm) );
   }

   void mySubtest21( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      // Make mySubtest1 a prerequisite
      verify = testManager.initialize( new Prerequisite("MyTests3","mySubtest1") );
      if ( verify )
      {
         // Perform checks here
      }
   }

   void mySubtest22( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         // Perform checks here
      }
   }
}
```

In this example, `mySubtest21` has these prerequisites:

Prerequisite	Explanation
`mySubtest1`	Explicitly specified
`mySubtest2`	It is a parent
`runTest`	It is an ancestor

In this example, `mySubtest22` has these prerequisites:

Prerequisite	Explanation
`mySubtest21`	`mySubtest2` runs its subtests with `SubtestSequencingMode.SEQUENTIAL_ABORT_SEQUENCE_ON_ERROR`
`mySubtest2`	It is a parent, and it is a prerequisite of `mySubtest21` (which is itself a prerequisite of `mySubtest22`)
`mySubtest1`	It is a prerequisite of `mySubtest21` (which is itself a prerequisite of `mySubtest22`)
`runTest`	It is an ancestor, and it is a prerequisite of `mySubtest21` (which is itself a prerequisite of `mySubtest22`)

In the selection window, when selecting a test, all of that test's prerequisites (if any) are automatically also selected. Similarly, when deselecting a test, all of that test's dependents (if any) are automatically also deselected.

The `Prerequisite` constructor has several overloads to specify:

- Whether the prerequisite's subtests should also be considered prerequisites (by default, subtests are considered prerequisites),
- How to handle the situation where multiple tests match the specified signature, and
- The type of the prerequisite, as described in the next subsection.

The `Prerequisite` constructors are described in the Undercamber JavaDocs.

Prerequisite Types

Undercamber has three types of prerequisites, enumerated in the `Prerequisite.Type` enum:

Type	Description
`FIXED`	This prerequisite must always be run before running the dependent test.
`CONDITIONAL_NOT_PREVIOUSLY_SATISFIED`	A conditional prerequisite that has not been previously satisfied prior to running the test suite.
`CONDITIONAL_PREVIOUSLY_SATISFIED`	A conditional prerequisite that has been satisfied prior to running the test suite. For this type of prerequisite, Undercamber will not automatically schedule the prerequisite to run unless it is scheduled by some other mechanism (more on this below).

To specify a conditional prerequisite, use a different overload of the `Prerequisite` constructor:

```
import com.undercamber.*;

public class ConditionalPrerequisite
   implements TestUnit
{
   public void runTest( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize( SubtestSequencingMode.SEQUENTIAL_CONTINUE_ON_ERROR );
      if ( verify )
      {
         // Perform checks here
      }
      testManager.addSubtest( tm -> timeConsumingSetup(tm) );
      testManager.addSubtest( tm -> mySubtest(tm) );
   }

   void timeConsumingSetup( TestManager testManager )
      throws Throwable
   {
       boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         new FileOutputStream( "DataFile.txt" ).close();  // Generate the file
      }
   }

   void mySubtest( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      if ( runPrerequisites() )
      {
         verify = testManager.initialize( new Prerequisite("MyTests4",
                                                            "timeConsumingSetup”,
                                                            Prerequisite.Type.CONDITIONAL_NOT_PREVIOUSLY_SATISFIED) );
      }
      else
      {
         verify = testManager.initialize( new Prerequisite("MyTests4",
                                                            "timeConsumingSetup”,
                                                            Prerequisite.Type.CONDITIONAL_PREVIOUSLY_SATISFIED) );
      }

      if ( verify )
      {
         // Perform checks here
      }
   }

   private boolean runPrerequisites()
   {
      return !( new File("DataFile.txt").exists() );
   }
}
```

With this strategy, when `mySubtest` is scheduled to run, `timeConsumingSetup(…)` might or might not also be automatically scheduled to run, depending on whether or not the data file previously exists. (Naturally, `timeConsumingSetup(…)` can be selected to run in the GUI even if the file already exists.)

Previously Satisfied Conditional Prerequisites

At first glance, the third type of prerequisite, `CONDITIONAL_PREVIOUSLY_SATISFIED`, can be confusing. When declared this way, the prerequisite is not run, so it seems like it is not a prerequisite at all. To describe the purpose of this category, it is necessary to understand some background:

- Undercamber supports requirements verification. A test can declare that it verifies a requirement, through the call to `TestManager.initialize(…)`. This is described above in "Requirements Verification".
- Ideally, every test either verifies a requirement, or supports a verification test. A test can support a requirement by either (a) verifying the requirement, or (b) being a prerequisite for a test that verifies a requirement.
- After running a test suite, Undercamber generates a report describing an analysis of the test suite. In particular, it makes a list of tests that do not support the verification of some requirement.

The third category of prerequisite, `CONDITIONAL_PREVIOUSLY_SATISFIED`, aids in generating the analysis report. If a test is a prerequisite for a verifying test, but the prerequisite is conditional and has already been satisfied before running Undercamber, then that test will still get credit for supporting a Requirement regardless of whether it is actually scheduled to run or not. In the above example, in class `MyTests4`, the test `timeConsumingSetup(…)` will be credited for supporting a `Requirement`, even if it does not run.

In the Undercamber API, `Prerequisite.Type` provides a shortcut to simplify the above example:

```
import com.undercamber.*;

public class SimplifiedConditionalPrerequisite
   implements TestUnit
{
   public void runTest( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize( SubtestSequencingMode.SEQUENTIAL_CONTINUE_ON_ERROR );
      if ( verify )
      {
         // Perform checks
      }
      testManager.addSubtest( tm -> timeConsumingSetup(tm) );
      testManager.addSubtest( tm -> mySubtest(tm) );
   }

   void timeConsumingSetup( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         new FileOutputStream( "DataFile.txt" ).close();  // Generate the file
      }
   }

   void mySubtest( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize( new Prerequisite("MyTests5",
                                                         "timeConsumingSetup”,
                                                         Prerequisite.Type.getConditionalType(runPrerequisites())) );
      if ( verify )
      {
         // Perform checks here
      }
   }

   private boolean runPrerequisites()
   {
      return !( new File("DataFile.txt").exists() );
   }
}
```

Prerequisites Must Precede Dependents

Naturally, a prerequisite must be scheduled to execute before the test that lists the prerequisite. If a prerequisite listed in the call to `TestManager.initialize(…)` is scheduled to execute after the test, Undercamber will throw an exception and the test suite will not be run.

Also, if a test and its prerequisite are scheduled to run concurrently, Undercamber will throw an exception.

Forcing Prerequisites

To run all prerequisites, including previously satisfied prerequisites, use either the `-forcePrerequisites` or the `-fp` command-line options. This is described in “Command Line Arguments”, below.

Test Uniqueness

By default, Undercamber gives each test a "name" which is the fully qualified method name. This might not result in a unique "name" for each test for several reasons:

- The method might be overloaded.
- The method might be called multiple times, possibly from different locations.

In most cases, this ambiguity is not a problem. However, this lack of uniqueness can cause difficulties when:

- Reading reports,
- Specifying a prerequisite, or
- Specifying a test to run from the command line (see "Command Line Arguments", below).

To resolve this, pass a unique String (an 'argument') to `TestManager.initialize(…)`:

```
import com.undercamber.*;

public class MyTests5
   implements TestUnit
{
   public void runTest( TestManager testManager )
      throws Throwable
   {
      testManager.initialize( SubtestSequencingMode.SEQUENTIAL_CONTINUE_ON_ERROR );
      testManager.addSubtest( tm -> mySubtest(tm,13) );
      testManager.addSubtest( tm -> mySubtest(tm,14) );
   }

   void mySubtest( TestManager testManager,
                   int         expectedResult )
      throws Throwable
   {
      boolean verify;
      int     result;

      verify = testManager.initialize( Integer.toString(expectedResult) );
      if ( verify )
      {
         result = getResult();

         if ( result != expectedResult )
         {
            throw new Exception( "Incorrect result:  expected = " + expectedResult + ", found = " + result );
         }
      }
   }

   private boolean getResult()
   {
      return 14;
   }
}
```

When an 'argument' is passed to `testManager.initialize(…)` as shown above, the argument becomes part of the test 'name'. In this case, the test can be referenced as a `Prerequisite` by using an overloaded constructor that includes an `argument` parameter. In `MyTests5` above, the first of the two subtests can be uniquely referenced, for example, using

```
new Prerequisite( "com.undercamber.MyTests5", "mySubtest", "13" );
```

It can be uniquely referenced from the command line using the `-test2` flag or the `-test4` flag, as described below in "Command Line Arguments".

The example in "A More Complete Example", below, is another illustration of passing an 'argument' to make a test signature unique.

Test Parameters

Undercamber requires that the top-level test in a test set have a viable no-argument constructor. Therefore, there is no direct way to pass parameters to the tests, but Undercamber has mechanisms to pass parameters to the tests.

Specifying Test Parameters

Undercamber has two mechanisms to specify what parameters, if any, should be passed to the tests:

1. Through the Configurator, using:
   - `TestSetBuilder.appendTestParameter(…)`,
   - `TestSetBuilder.prependTestSetParameter(…)`,
   - `TestSetBuilder.appendTestSetParameterPair(…)`, or
   - `TestSetBuilder.prependTestSetParameterPair(…)`.

   See "Test Parameters in the Configurator", below, for a detailed explanation.

2. On the command line, using either a `-p` or a `-pp` flag. See "Command Line Arguments", below for a more detailed explanation.

In the configurator, each test set process can be given a different set of parameters to be passed into the test system. Test parameters specified on the command line are passed to all test set processes. Test parameters specified in the configurator are appended to the test parameters specified on the command line.

Accessing Test Parameters

To access the test parameters from within the test code, use one or more of these methods:

- `TestManager.getParameters()`
- `TestManager.getFollowingParameterAsString(…)`
- `TestManager.getFollowingParameterAsInteger(…)`
- `TestManager.getFollowingParameterAsLong(…)`
- `TestManager.getFollowingParameterAsDouble(…)`
- `TestManager.getFollowingParameterAsFloat(…)`
- `TestManager.containsParameter(…)`

These methods are described in the Undercamber JavaDocs.

Tags

When testing software, it is often convenient to have a pre-canned subset of tests that can be run from the command line (or from a script), without going through a GUI. This capability is required in automated test systems. To support this, Undercamber has *Tags*, which is basically a name that can be attached to individual tests. A test can have zero or more tags associated with it, and different tests can share a Tag.

A tag is applied using an overloaded version of `TestManager.initialize(…)`:

```
import com.undercamber.*;

public class MyTests6
   implements TestUnit
{
   public void runTest( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();
      if ( verify )
      {
         // Perform checks here
      }
      testManager.addSubtest( tm -> mySubtest1(tm) );
      testManager.addSubtest( tm -> mySubtest2(tm) );
   }

   void mySubtest1( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize( new Tag("TestB") );
      if ( verify )
      {
         // Perform checks here
      }
   }

   void mySubtest2( TestManager testManager )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize( new Tag("TestA") );

      if ( verify )
      {
         // Perform checks here
      }
   }
}
```

The `Tag` constructor has an overload to specify whether to include or not include all subtests of the tagged test, as described in the Undercamber JavaDocs. The single-argument constructor includes subtests by default.

When invoking Undercamber, the command line can specify zero or more tag names. When invoked this way, Undercamber will run all of the tests with the specified tags, without displaying the selection window. To run `MyTest6.mySubtest2(...)` from the command line without displaying the selection window, use the `-tag1` or `-tag2` command line option:

```
java com.undercamber.Undercamber -tag1 TestA
```

When specifying a tag from the command line, Undercamber will run the specified tests and their prerequisites, without displaying the selection window. The `-tag1` and `-tag2` command-line options are described below in “Command Line Arguments”.

The example in “A More Complete Example”, below, also illustrates the use of tags.

Results Directory

Undercamber places its results files in a user-specified root output directory. Each time Undercamber runs, it creates a new subdirectory below the user-specified root directory and places the results files there.

Output Root Directory

The root output directory used by Undercamber can be specified three different ways:

1. On the command line, using the `-rootDirectory` flag. See "Command Line Arguments", below.
2. In the configurator, using `Configurator.setResultsRootDirectory(…)` or `Configurator.setResultsRootDirectoryName(…)`. See "The Configurator", below.
3. In the `UNDERCAMBER_ROOT` environment variable.

If the output directory is specified in more than one way, Undercamber uses this search precedence:

1. The command line, then
2. The configurator, then
3. The `UNDERCAMBER_ROOT` environment variable.

If the output root directory is not specified, and exception is thrown.

Output Subdirectory

The results subdirectory created by Undercamber can be specified on the command line using the `-subdirectory` flag, as described below in "Command Line Arguments". If the subdirectory is not specified on the command line, Undercamber will generate a subdirectory name from the current date and time.

User Directory

Each time Undercamber runs, it creates a new, empty, and unique test directory that can be used by the test software. It can be used to save data files, write intermediate results, or any other purpose needed by the test software. Undercamber creates this directory and makes it available to the test software, but otherwise does nothing with it.

This directory is created below Undercamber's results subdirectory. To discover the working directory, use `TestManager.getUserWorkingDirectory()`, as described in the Undercamber JavaDocs

The example in "A More Complete Example", below, shows one strategy for using this directory.

The GUI

The Test Selection Window

The initial screen shown by Undercamber is the test selection window:

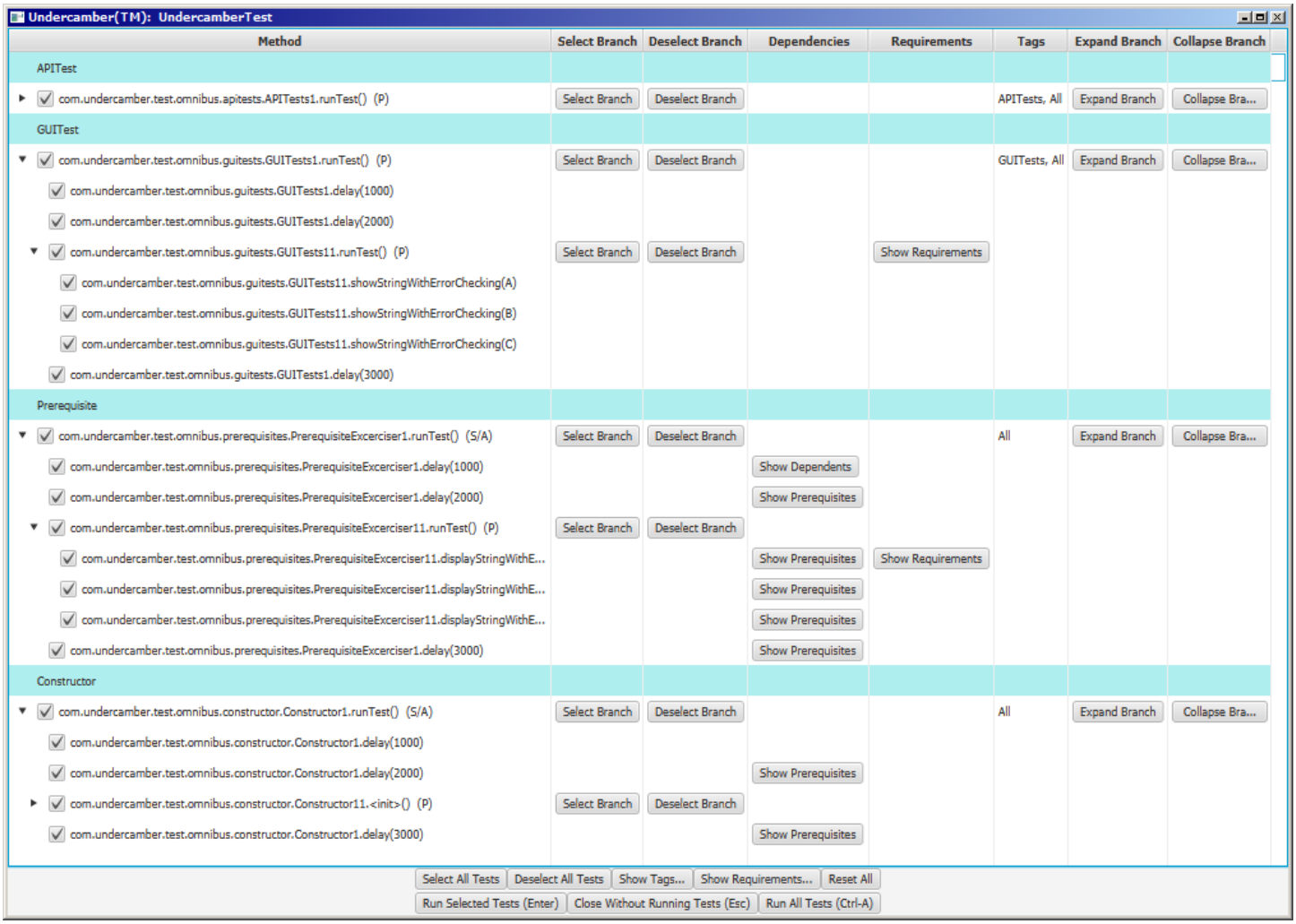

Figure 5: The Undercamber Selection Window

This window is used to select which tests to run.

When a test is selected in this window, all of that test's prerequisites are also automatically selected. Similarly, when a test is deselected, all of that test's dependents are also automatically deselected.

Run Individual Tests

To directly run an individual test and its prerequisites, use the pop-up menu:

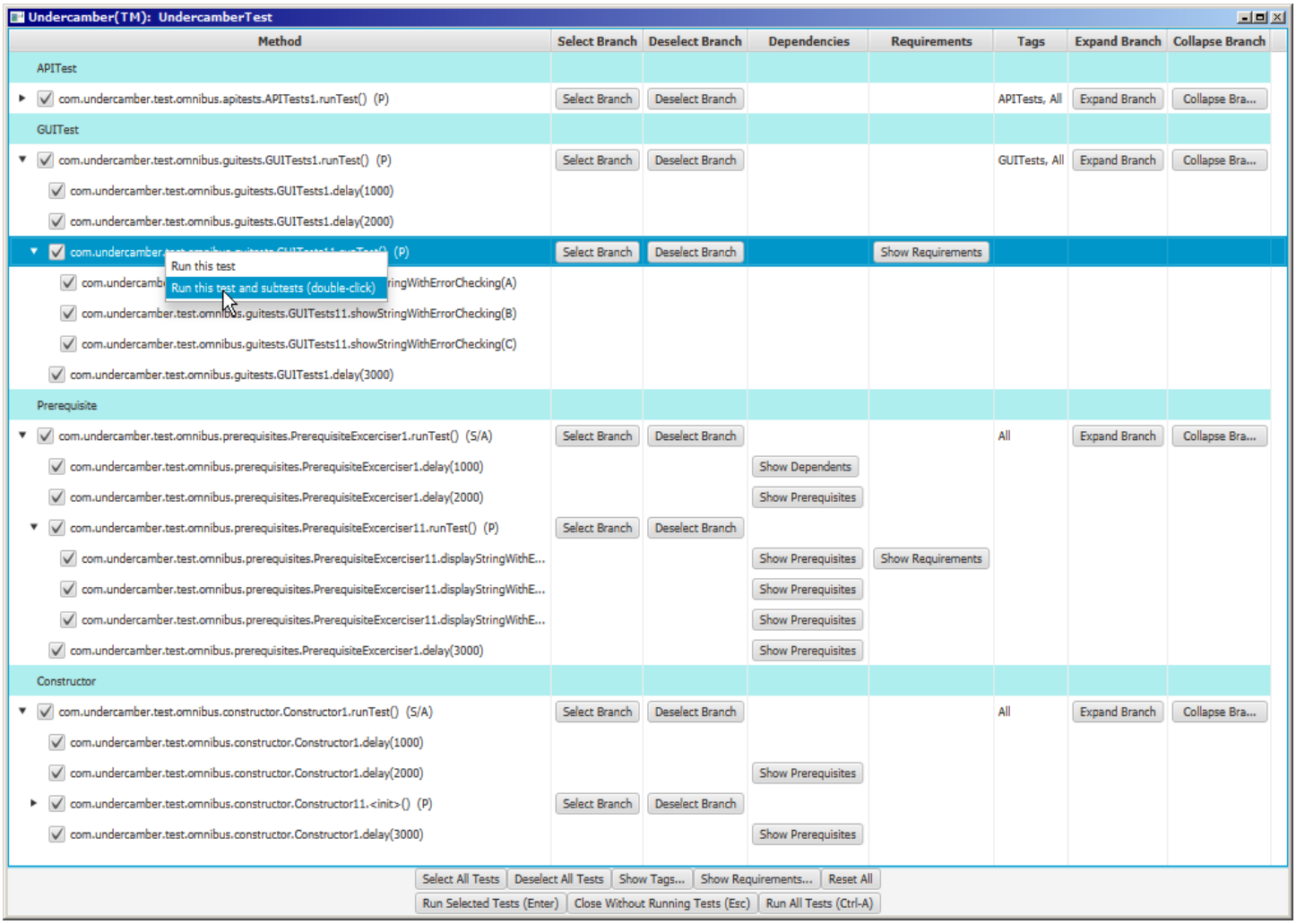

Figure 6: Select and Run Individual Tests

When a test is run using this pop-up menu, the check boxes in the GUI are ignored.

Run a Test Set

To run all of the tests in a test set, and all needed prerequisites, use the pop-up menu:

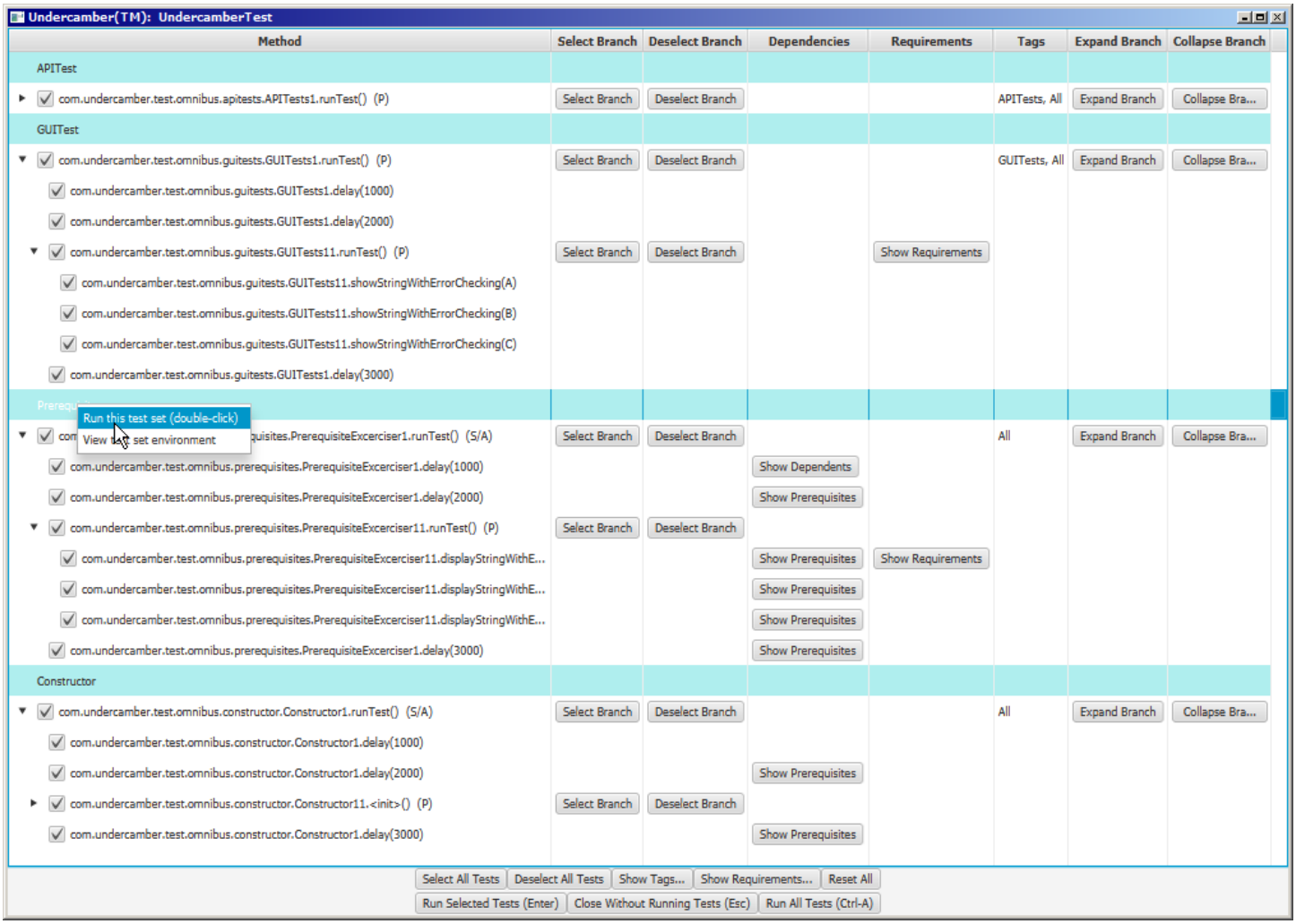

Figure 7: Test Set Pop-up Menu

View Test Set Details

To view the environment used in the second pass for a particular test set, use the pop-up menu:

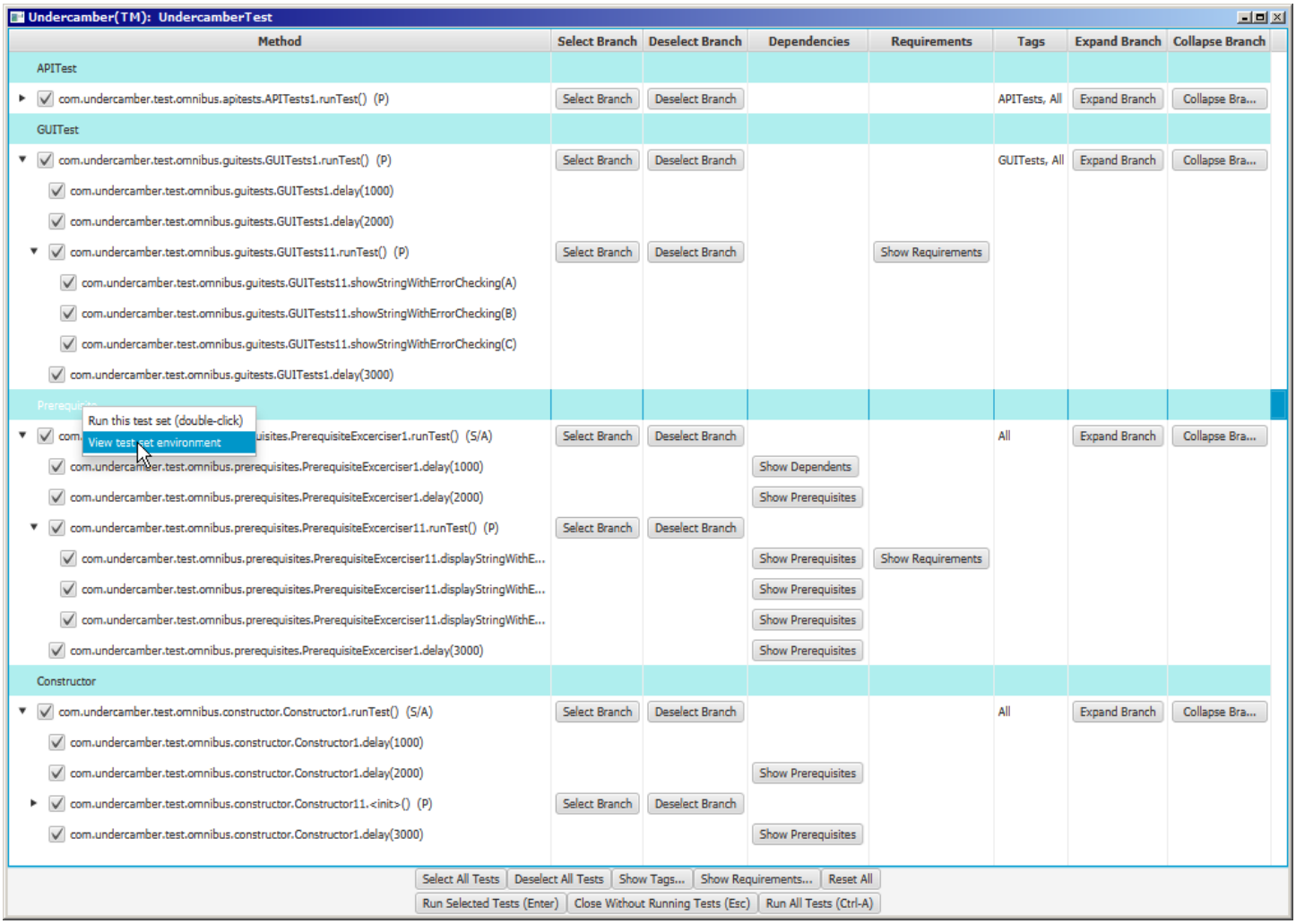

Figure 8: View Test Set Details

Undercamber shows the details of the test set:

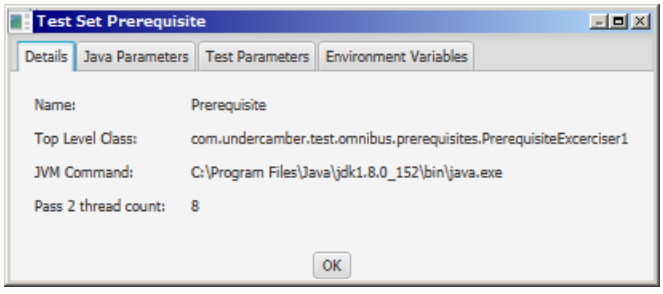

Figure 9: Test Set Details

The “Java Parameters” tab shows the parameters used to invoke the JVM for the second pass:

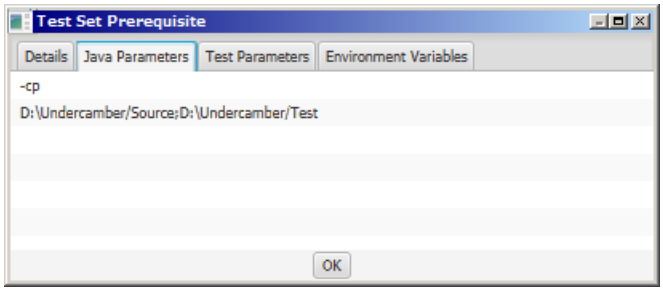

Figure 10: Pass 2 JVM Parameters

The “Test Parameters” tab shows the parameters passed to the tests:

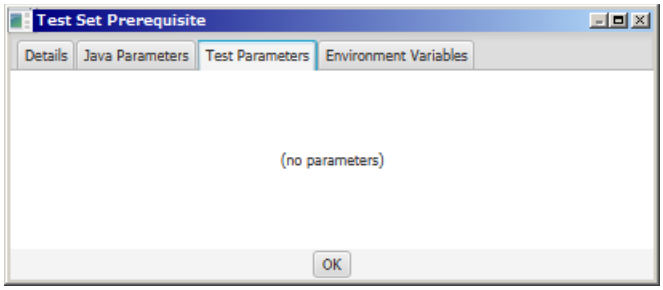

Figure 11: Test Parameters

The “Environment Variables” tab shows the envrionment variables used in the second pass:

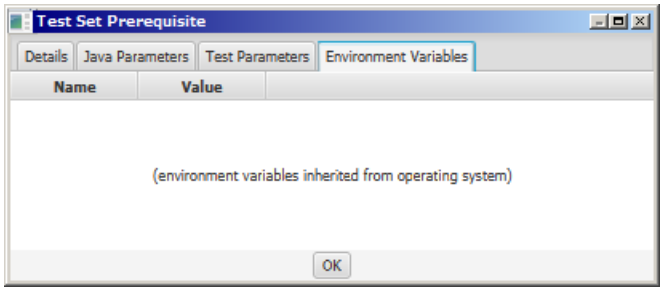

Figure 12: Pass 2 Environment Variables

Selecting Branches

To set or unset all of the selection boxes for a test and all of its subtests, use the buttons “Select Branch” and “Deselect Branch” columns:

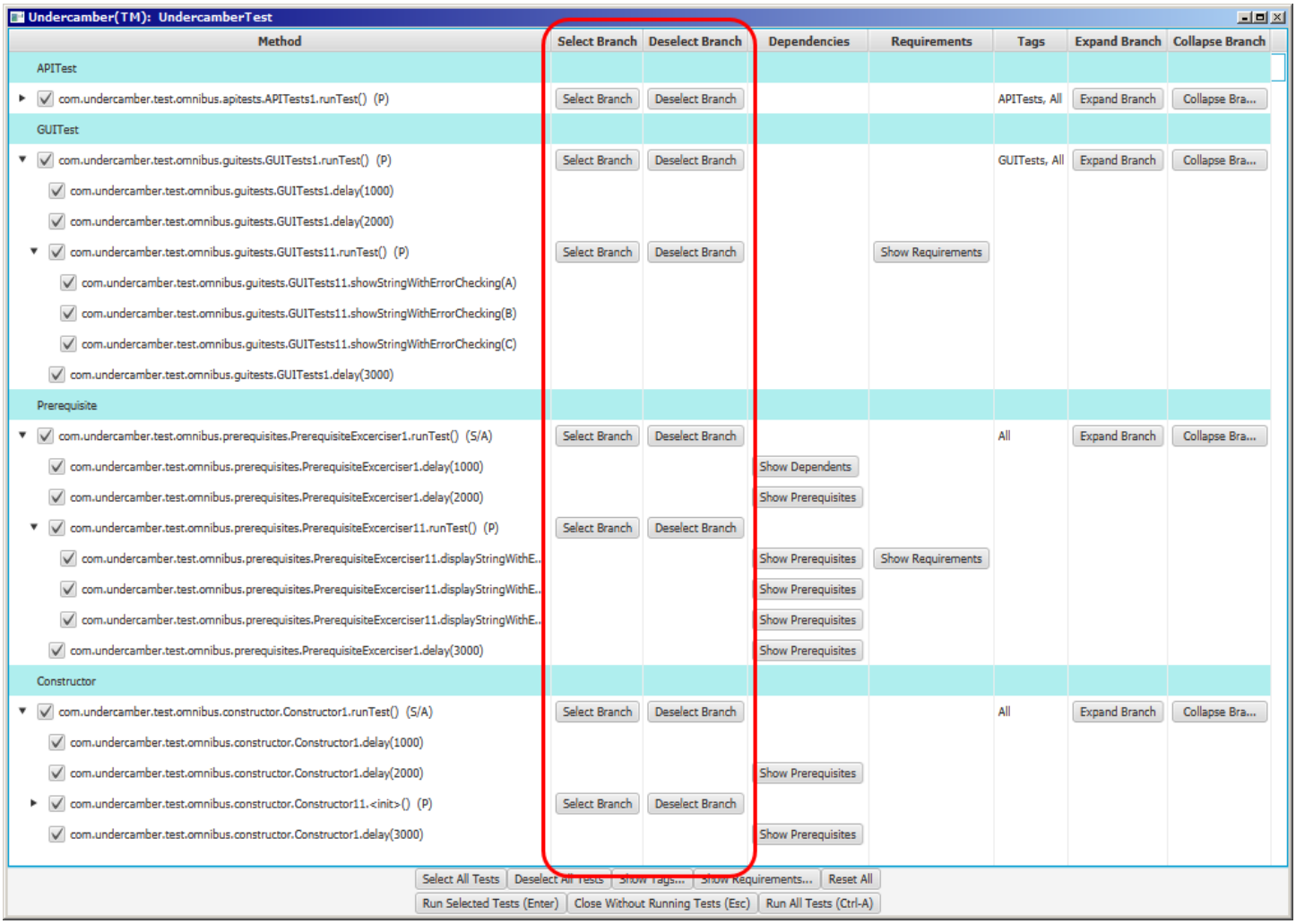

Figure 13: Select and Deselect Branches

Show Dependencies

To show the prerequisites, dependents, or both for a particular test, click on the button in the “Dependencies” column:

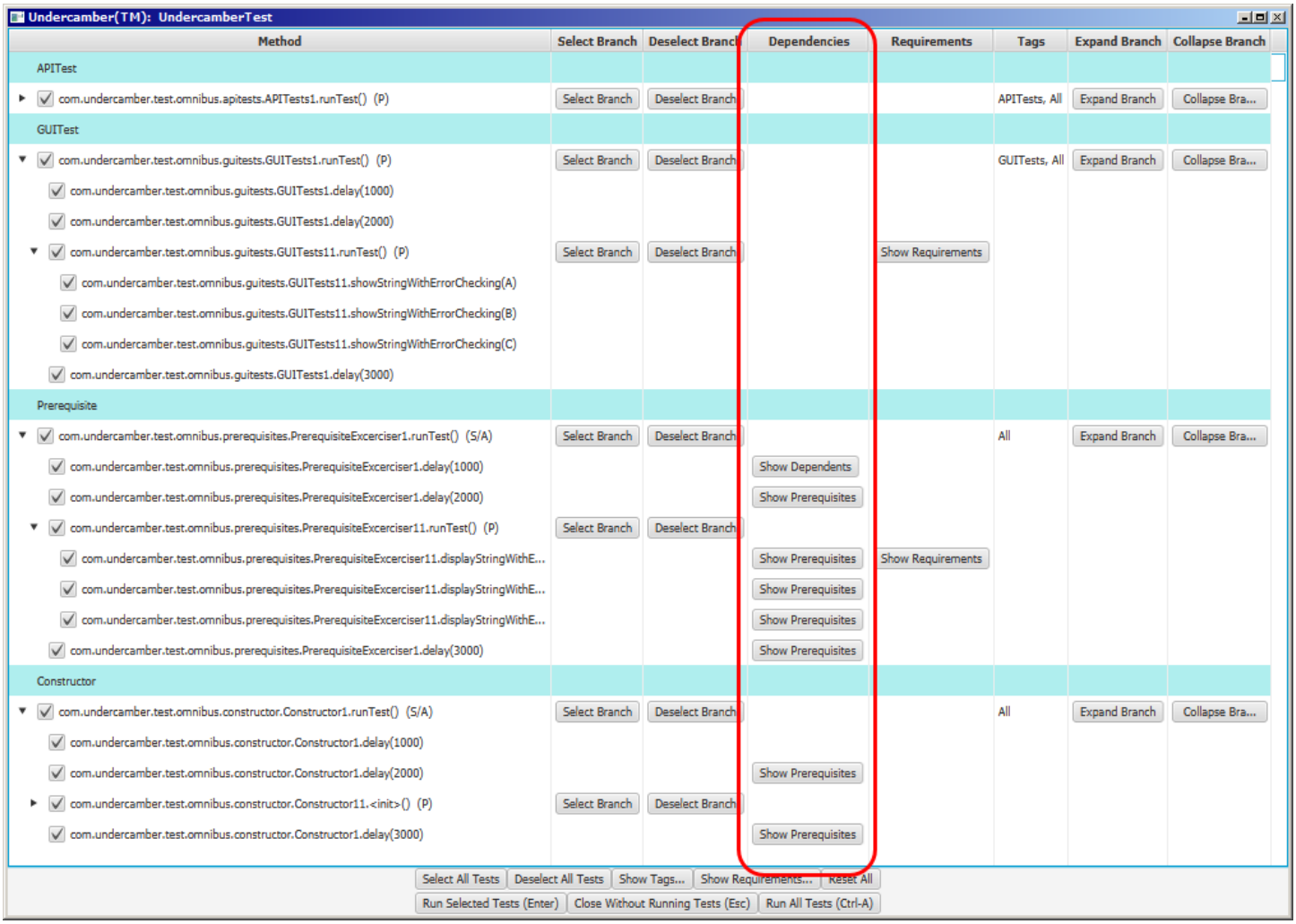

Figure 14: Show Dependencies

When one of these buttons is clicked, Undercamber shows a list of the dependencies:

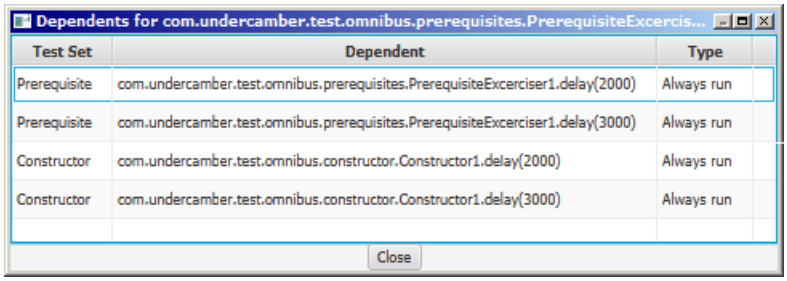

Figure 15: Dependency Window

Like the selection window, individual tests (and their prerequisites) can be run from the dependency window:

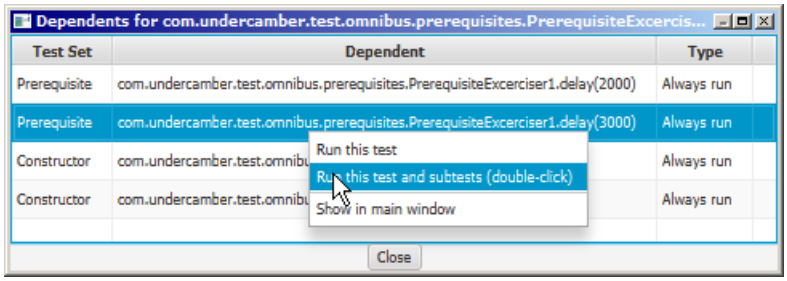

Figure 16: Popup Menu in the Dependency Window

When a test is run this way, the check boxes in the selection window are ignored.

As shown above, the popup menu in the Dependencies window has a "Show in main window" item. This item helps the user to find the selected test in the main selection window.

Requirements

To view the requirements validated by specific tests, use the buttons in the "Requirements" column:

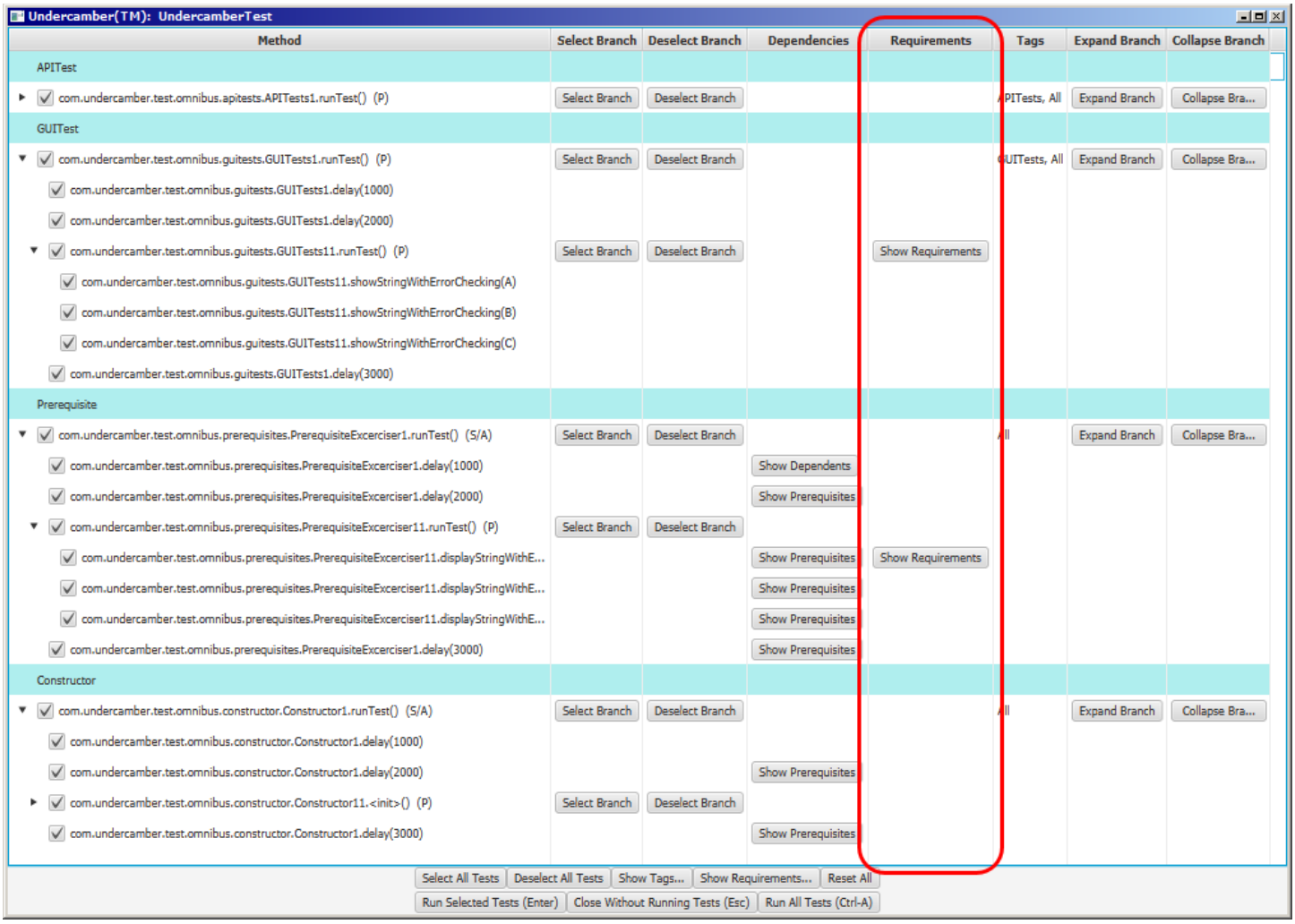

Figure 17: Requirements for Individual Tests

When one of these buttons is clicked, Undercamber displays a list of the requirements validated by the test:

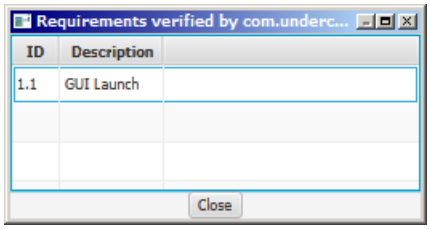

Figure 18: Requirements Validated by Specific Tests

Tags Column

The "Tags" column shows what tags are attached to each test:

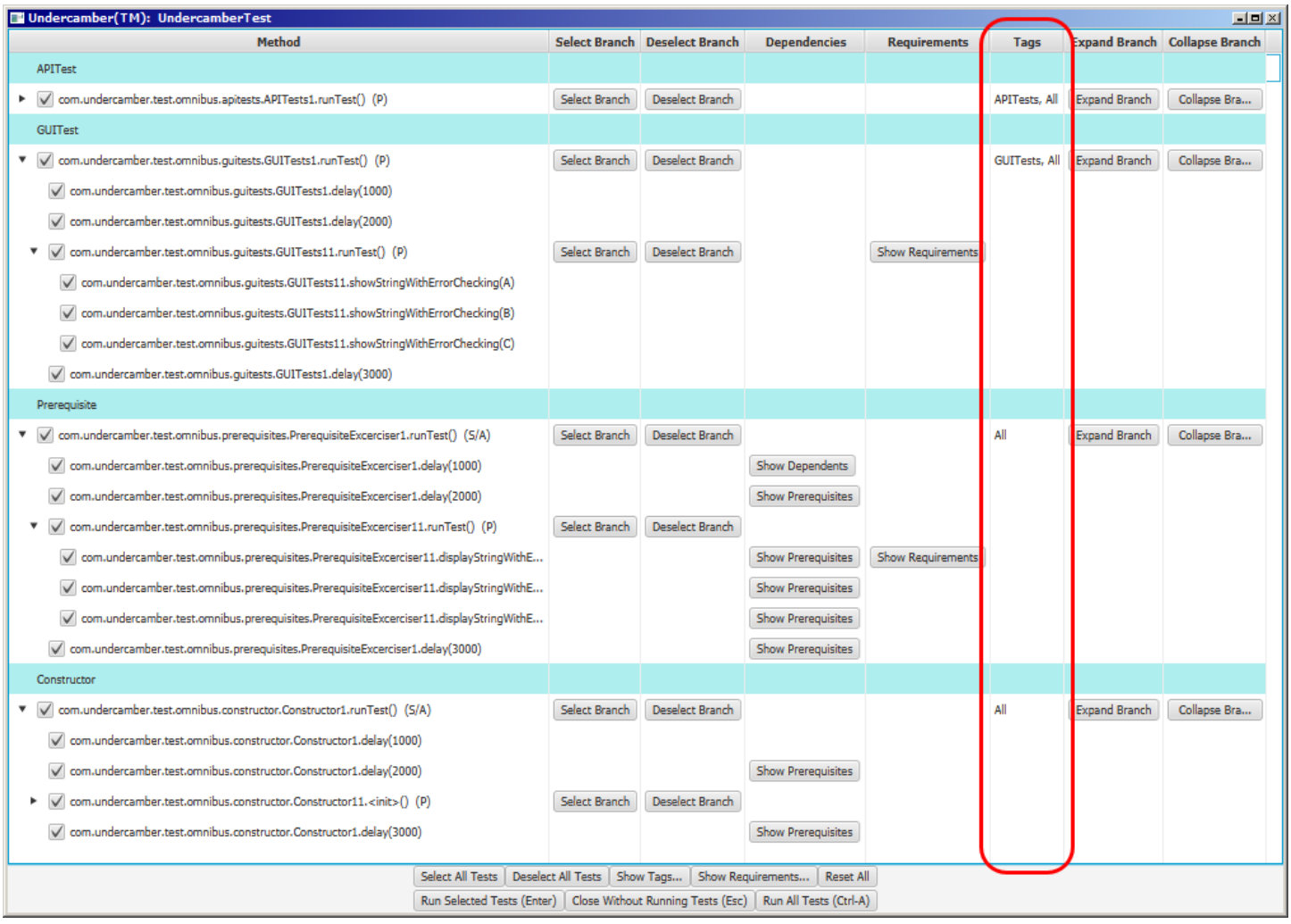

Figure 19: Tags Column

Expand and Collapse Branches

Entire branches of the test tree can be graphically expanded and collapsed using the buttons in the “Expand Branch” and “Collapse Branch” columns:

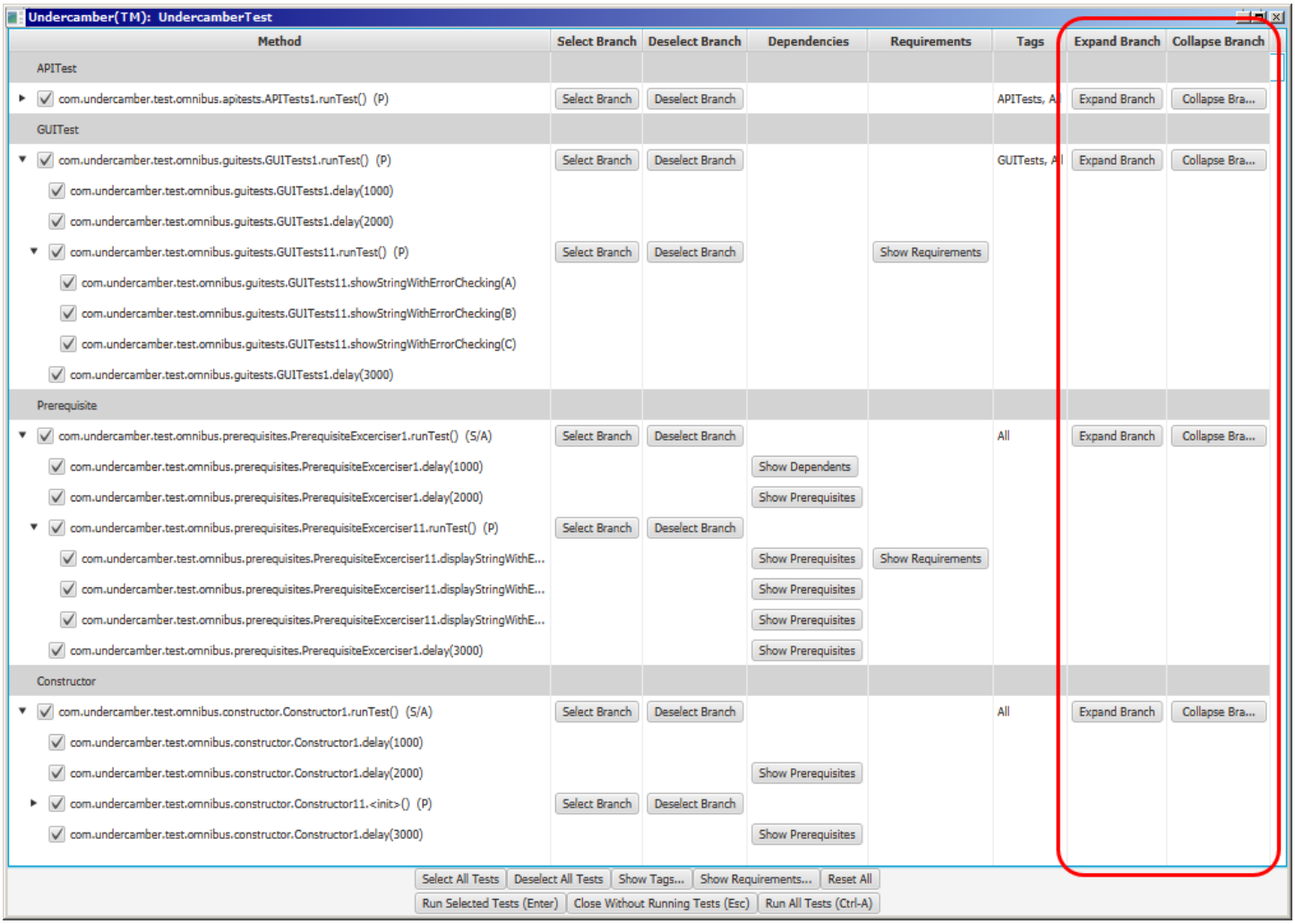

Figure 20: Expand and Collapse Branches

Tag List

To view a list of all of the tags in the test suite, click on the “Show Tags…” button:

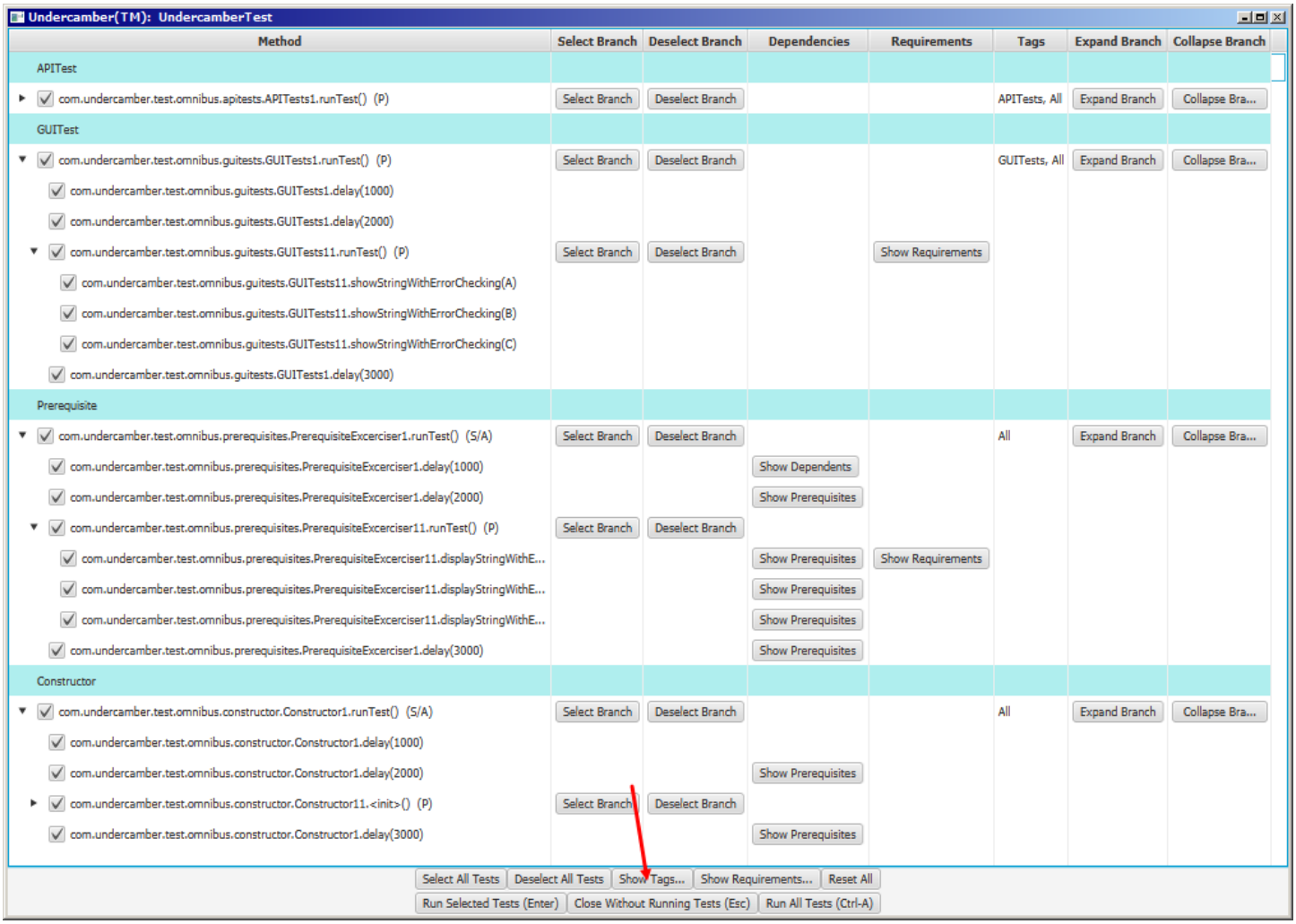

Figure 21: Show Tags Button

Clicking on the “Show Tags…” button shows the list of tags:

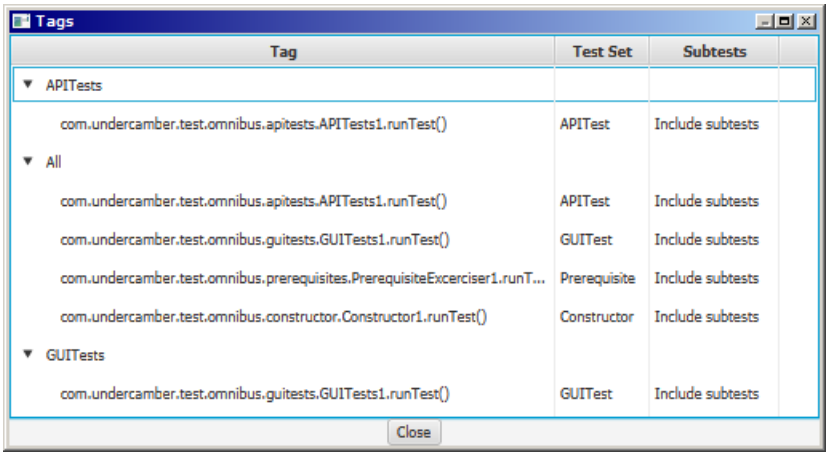

Figure 22: Tags List

To run one of the tests in the list, and its prerequisites, use the popup menu:

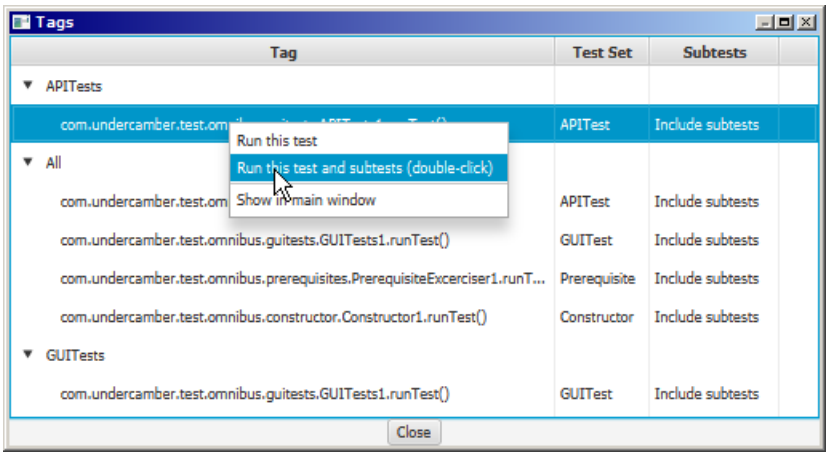

Figure 23: Test Popup Menu in Tags Window

When a test is run this way, the check boxes in the selection window are ignored.

As shown above, the popup menu in the Tags window has a “Show in main window” item. This item helps the user to find the selected test in the main selection window.

To run all of the tests under a particular tag, along with their prerequisites, use the popup menu:

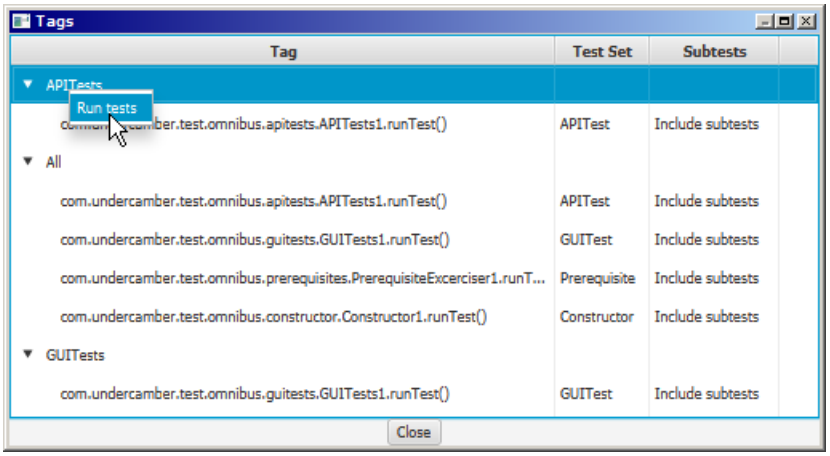

Figure 24: Tag Popup Menu

When tests are run this way, the check boxes in the selection window are ignored.

Requirements List

To view a list of all of the requirements specified in the test suite, click on the “Show Requirements…” button:

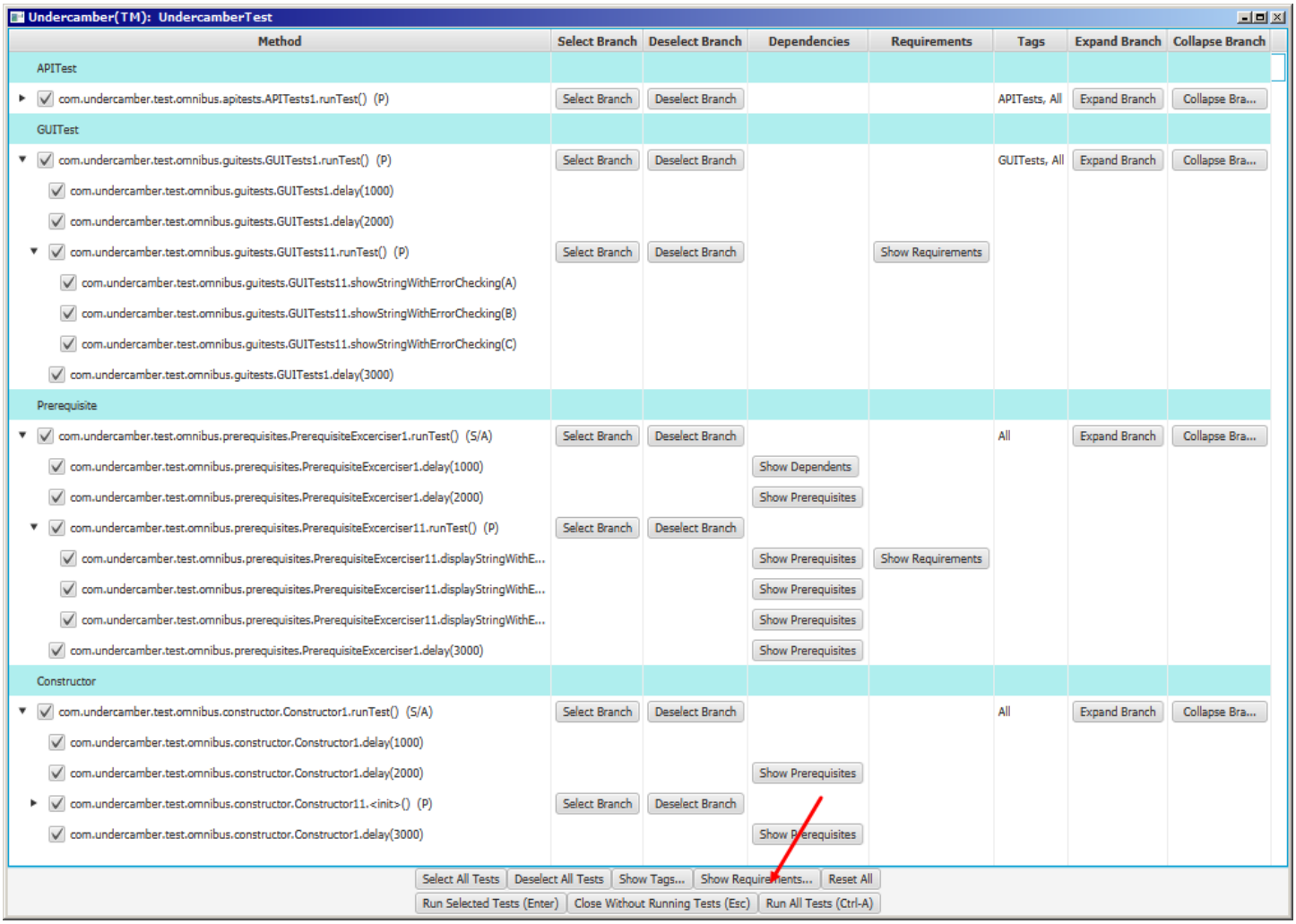

Figure 25: Requirements List Button

When the “Show Requirement…” button is clicked, Undercamber shows a list of all of the requirements in the test suite:

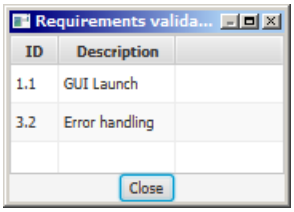

Figure 26: Requirements List

Skipping the Selection Window

The selection window will be skipped if any of these command-line options are specified:

- `-a`
- `-tag1`
- `-tag2`

- `-test1`
- `-test2`
- `-test3`
- `-test4`
- `-set`
- `-g`

These command-line options are described below in the section “Command Line Arguments”.

The Results Screen

After running the tests, Undercamber shows the results screen:

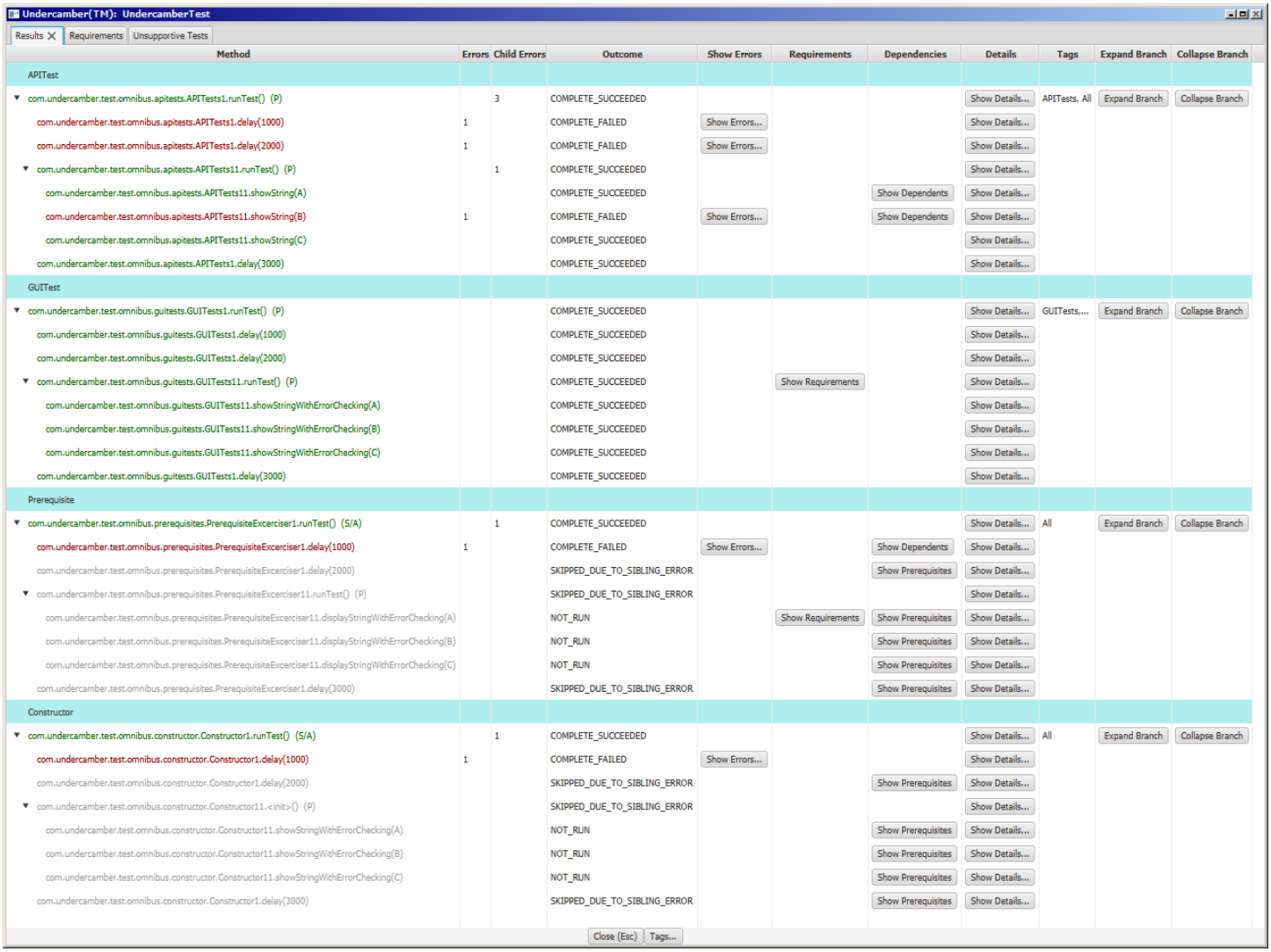

Figure 27: The Results Screen

In this screen, the tests listed in green passed, the tests listed in red failed, and the tests listed in gray were not run.

Results Summary

The “Errors”, “Child Errors”, and “Outcome” columns provide a synopsis of the test results:

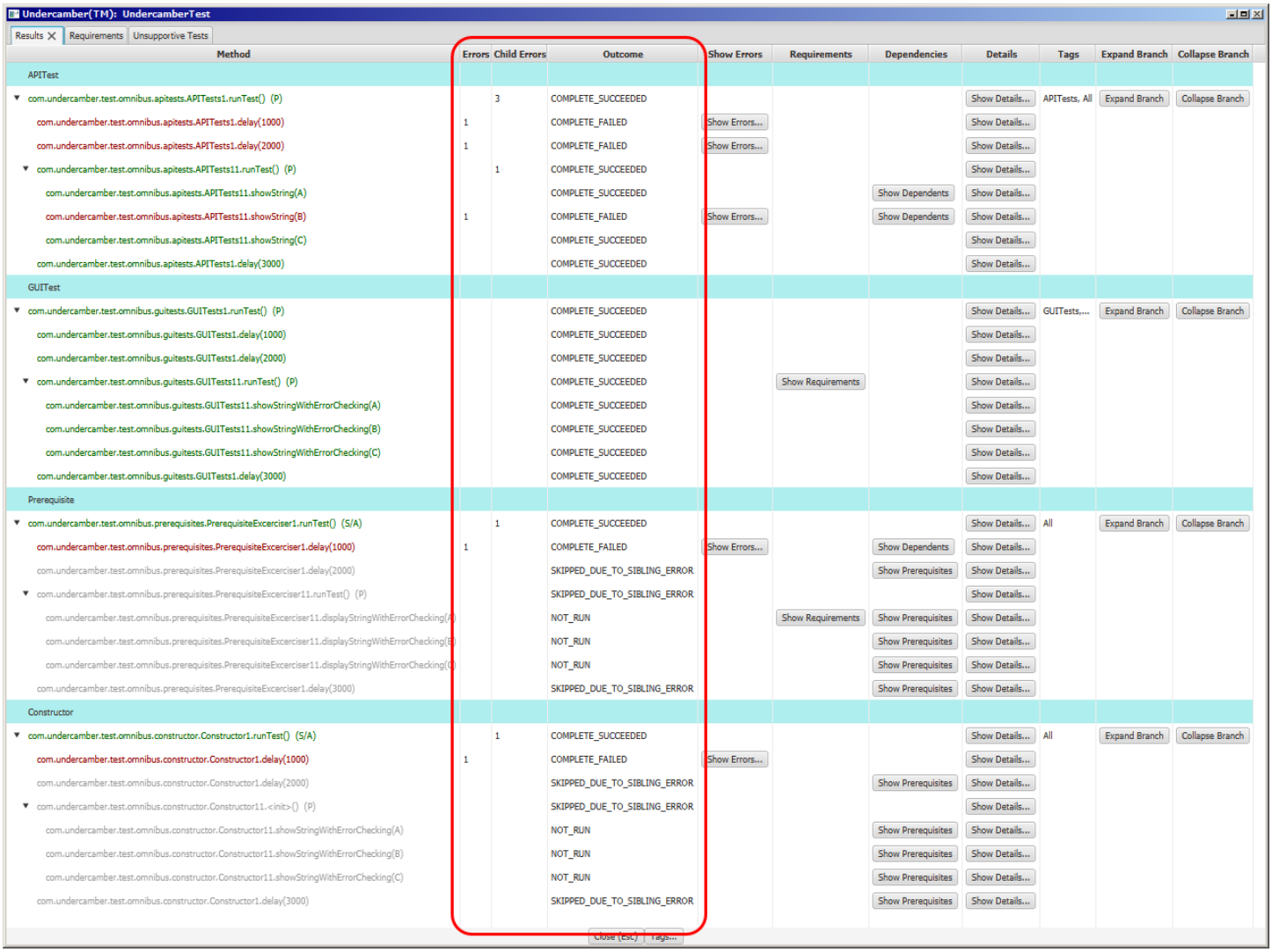

Figure 28: Synopsis Columns

Show Errors

To view the errors in a specific test, use the buttons in the “Show Errors” column:

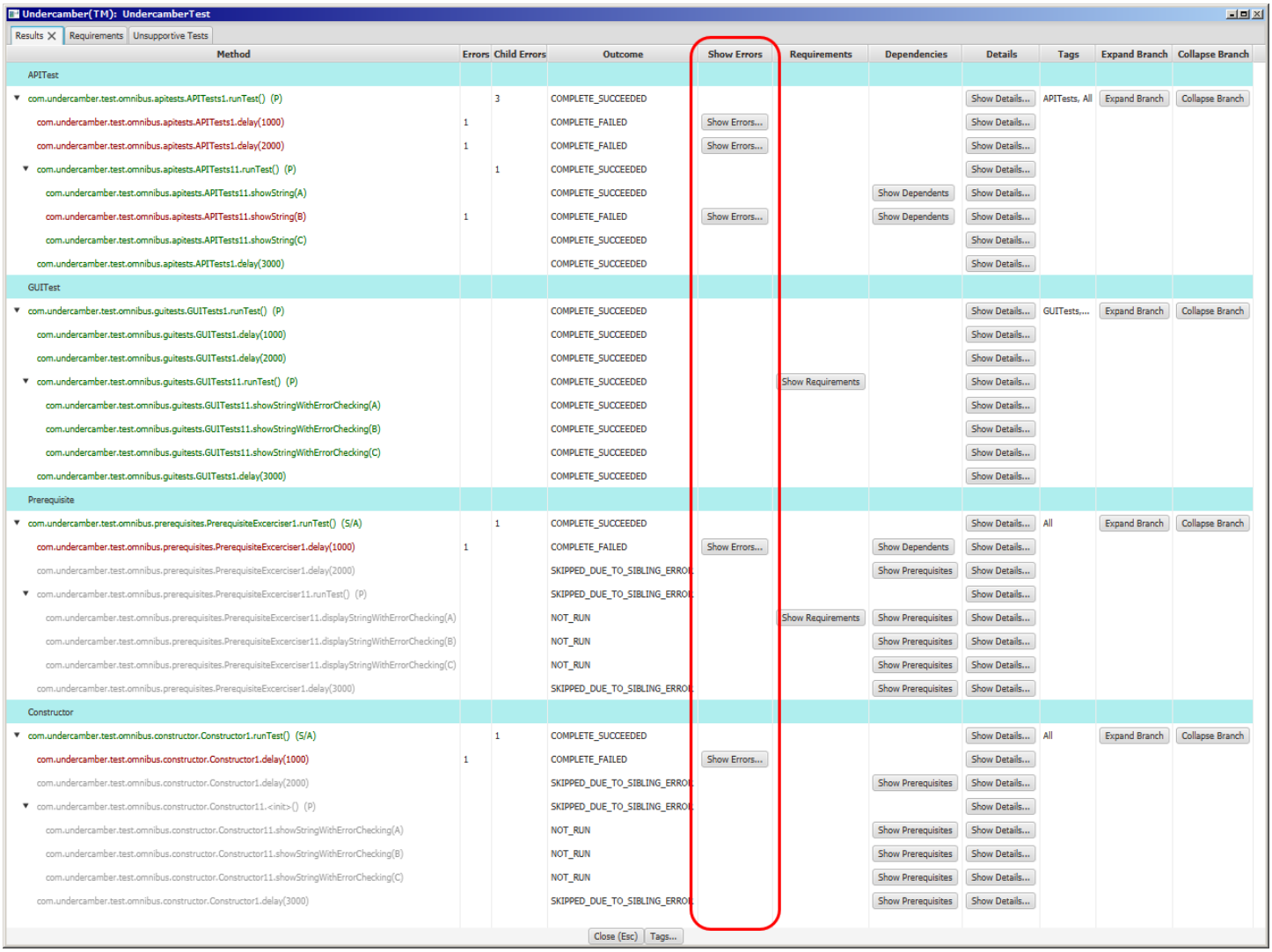

Figure 29: Show Errors Column

Pressing one of the buttons in the “Show Errors” column will display the errors for the test:

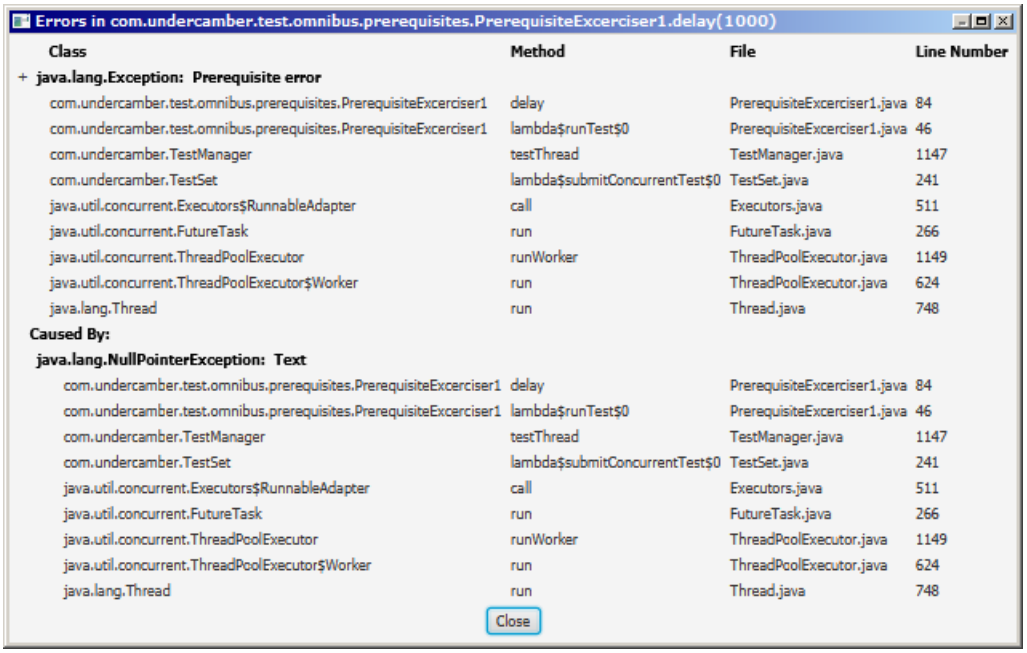

Figure 30: Error Listing

Requirements Results

To view the requirements validated by a test, use the “Requirements” column:

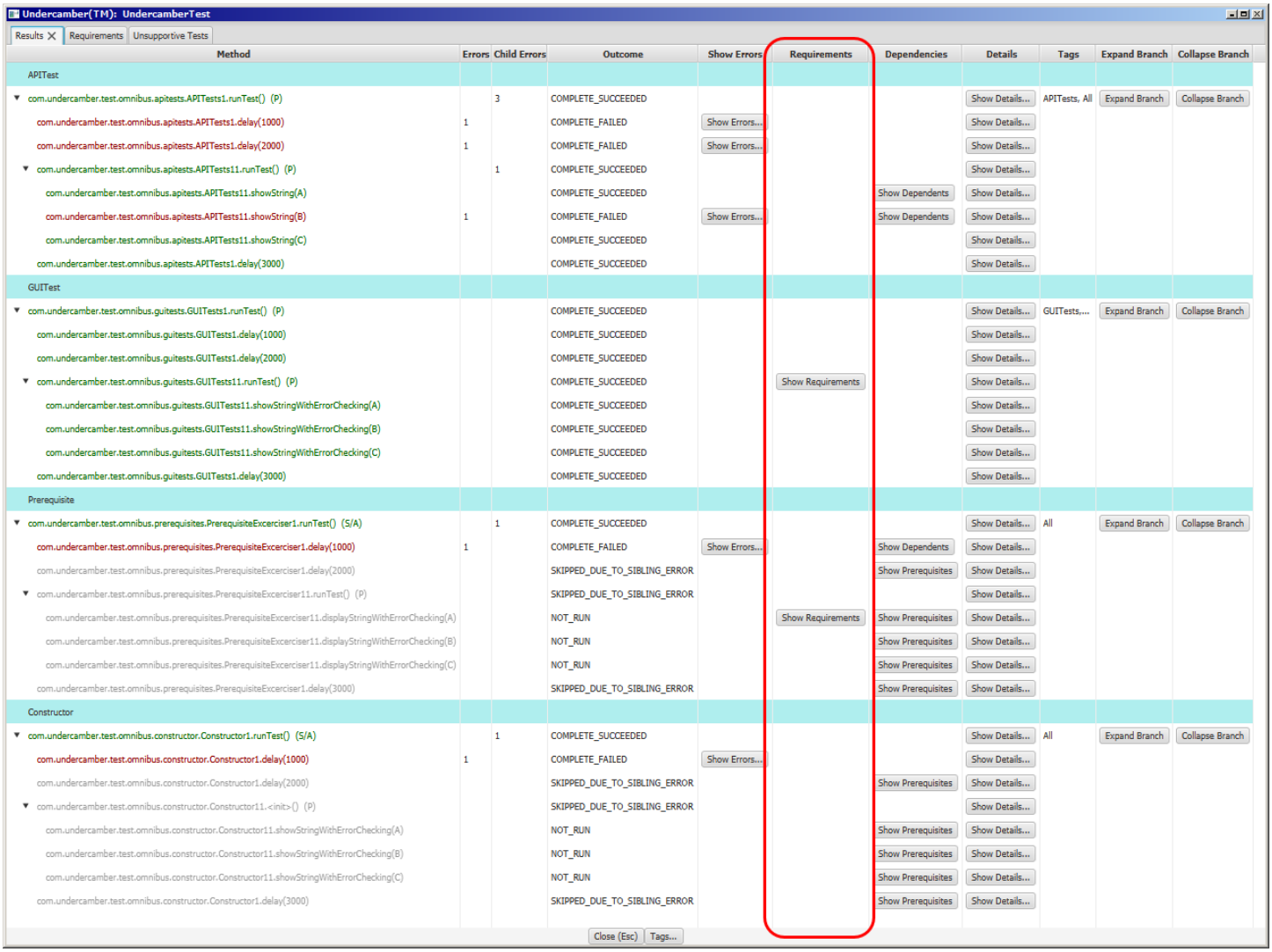

Figure 31: Requirement Results Column

Pressing one of the buttons in the “Requirements” column shows a list of requirements and their validation results:

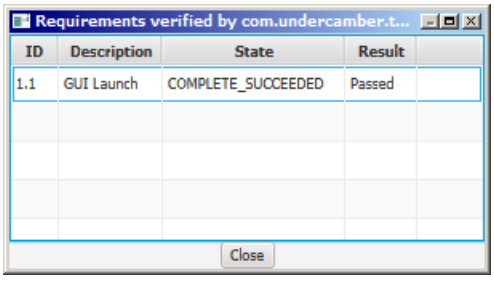

Figure 32: Requirements List

Dependencies

To view the prerequisites or dependencies for a test, use the “Dependencies” column:

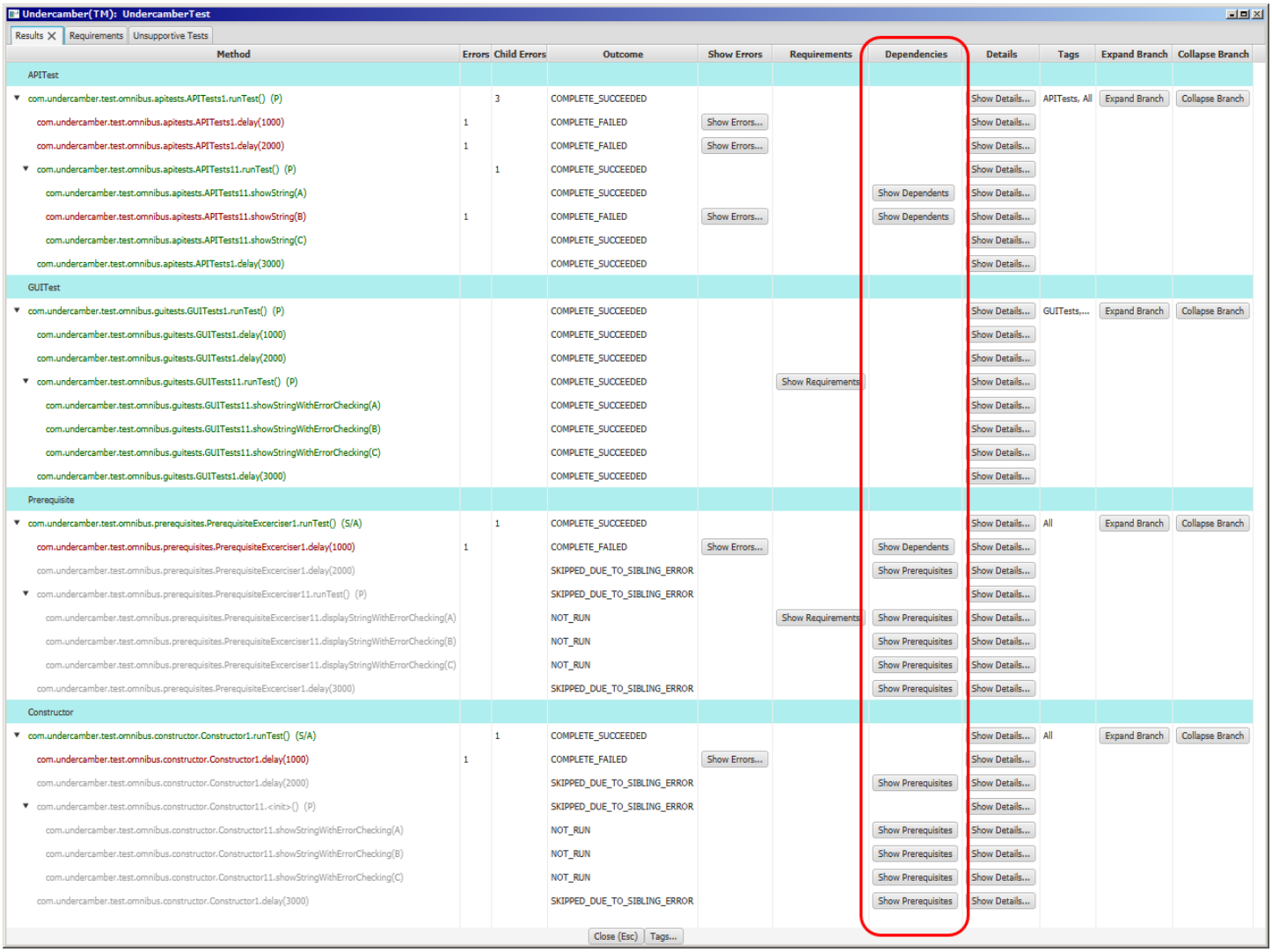

Figure 33: Dependencies Column

When one of the buttons in the “Dependencies” column is clicked, Undercamber shows the dependents and/or prerequisites for the test:

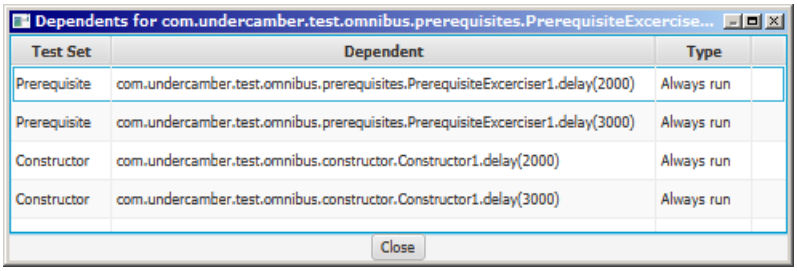

Figure 34: Dependencies Window

Prerequisites and dependents are described above in “Prerequisites”.

To help find a test in the main window, the dependencies window has a popup menu:

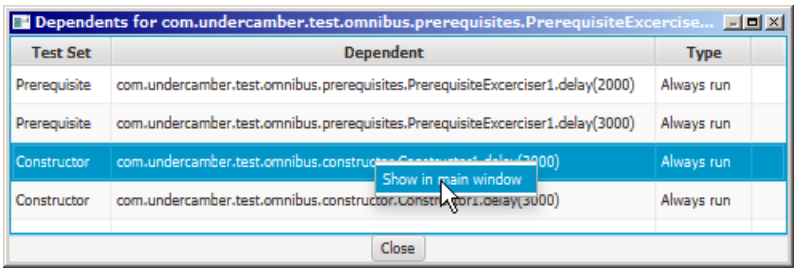

Figure 35: Popup Menu

Test Details

To view the detailed results of a test, use the buttons in the "Details" column:

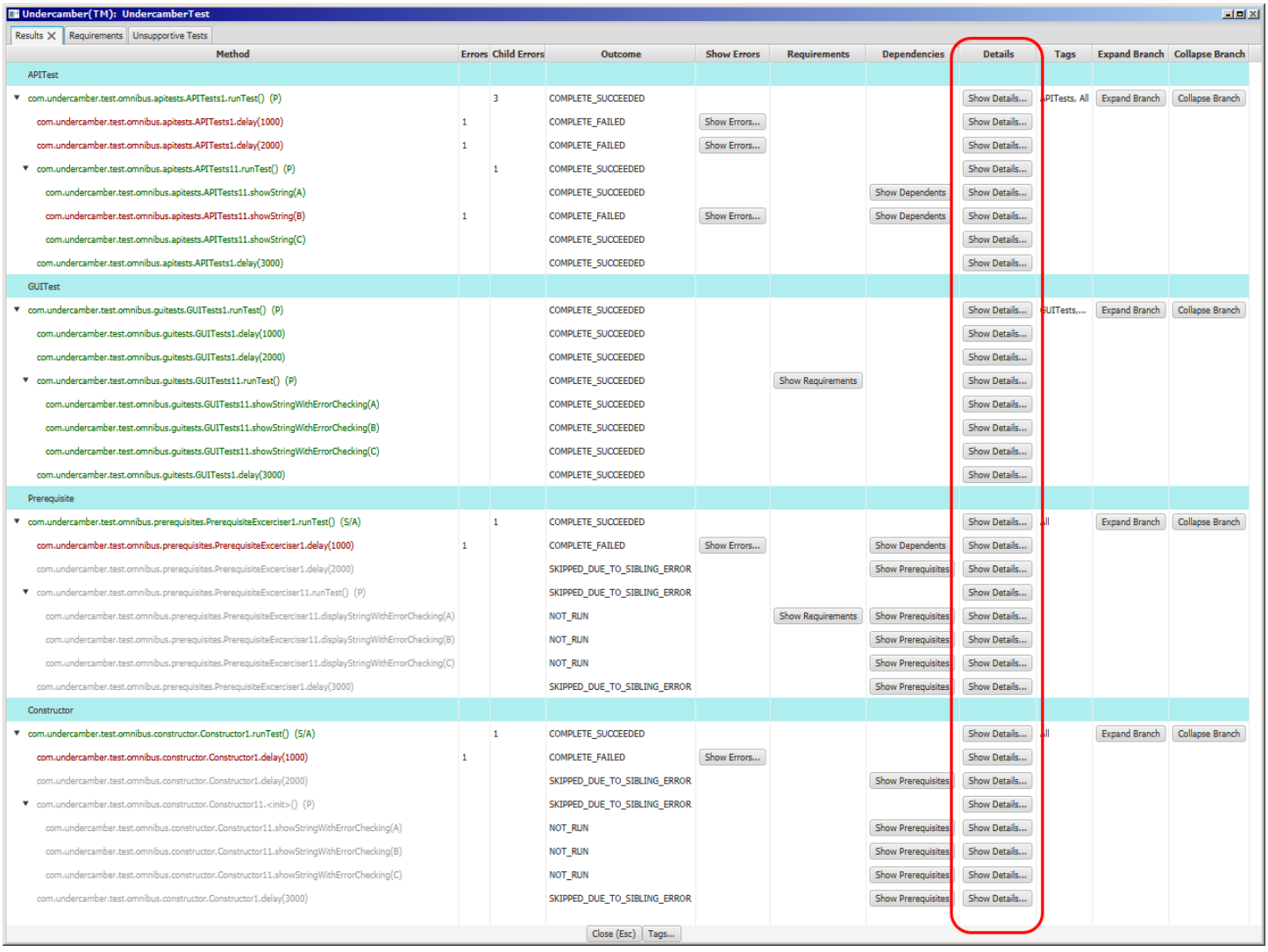

Figure 36: Test Details Column

When a button in the “Details” column is clicked, Undercamber shows the test details:

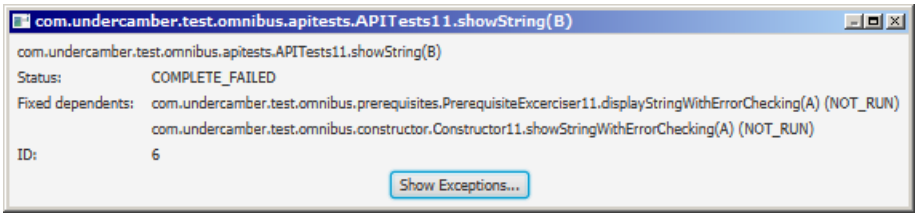

Figure 37: Test Details

The test status is described above in “Test Outcome”.

Prerequisites and dependents are described above in “Prerequisites”.

Tags

The “Tags” column shows the tags attached to each test:

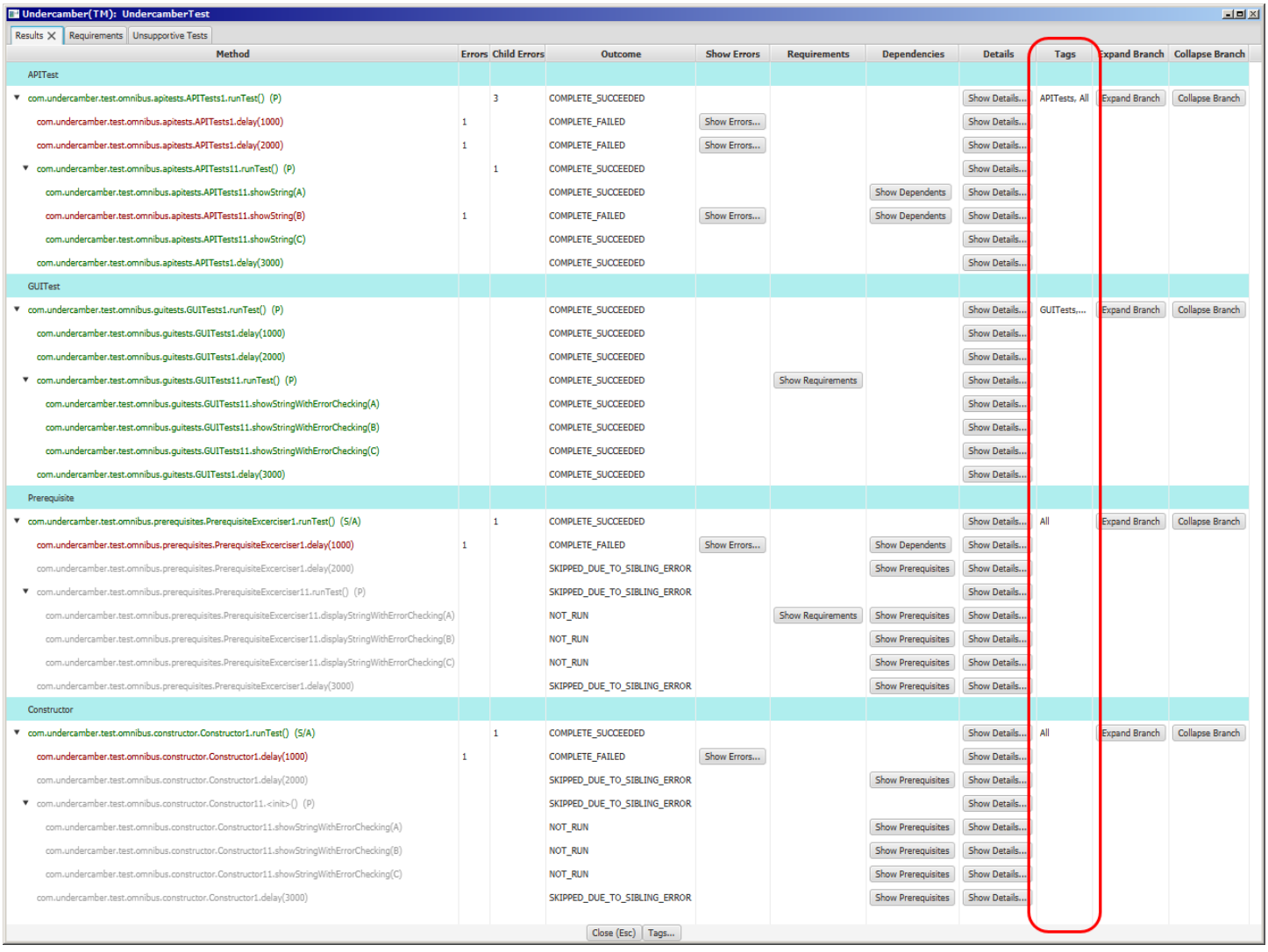

Figure 38: Tags Column

Expand and Collapse Branches

To graphically expand or collapse entire branches of the test set, use the buttons in the “Expand Branch” and “Collapse Branch” columns:

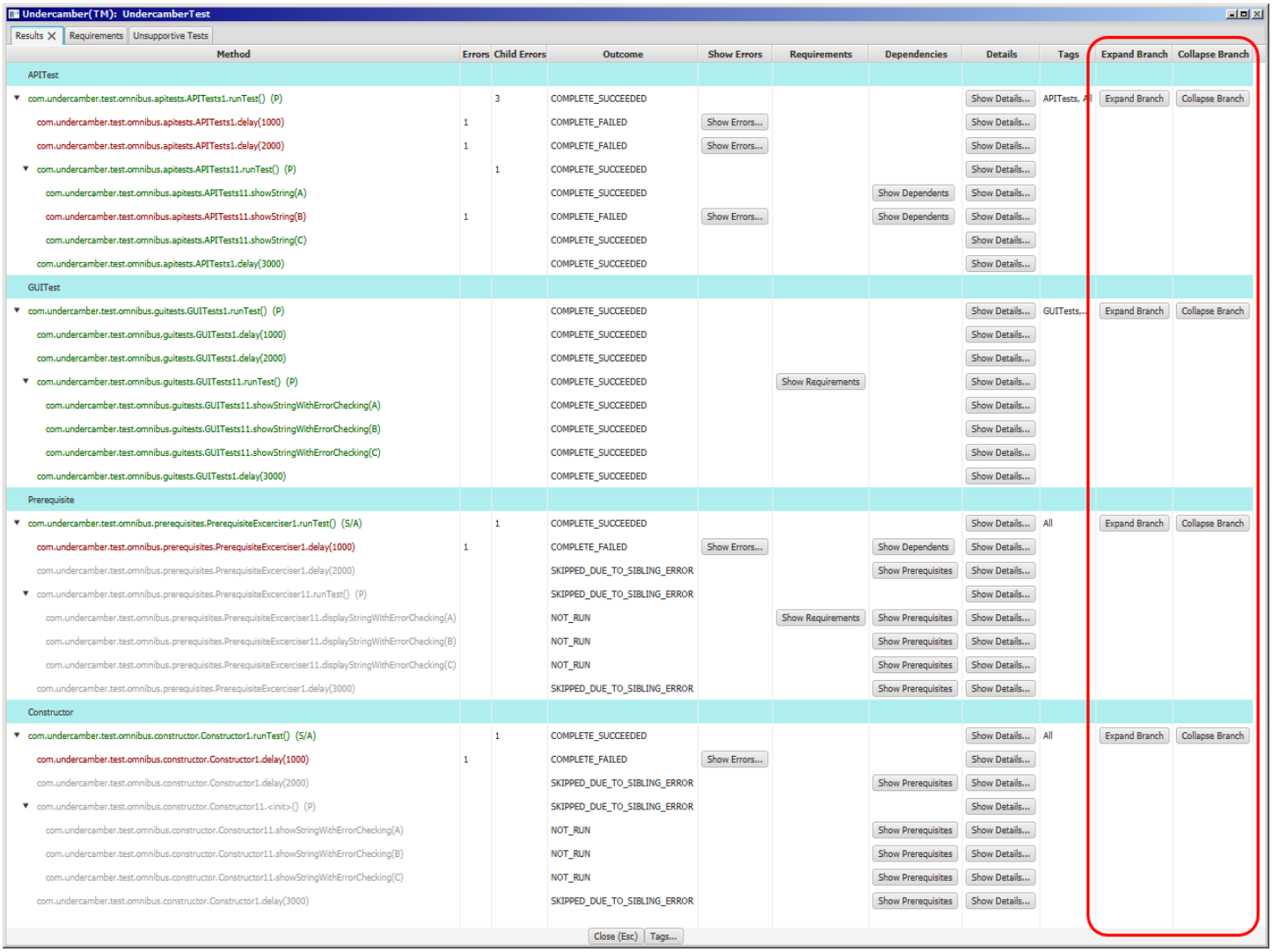

Figure 39: Expand and Collapse Branch Columns

View Test Set Details

To view the details of a test set, use the pop-up menu:

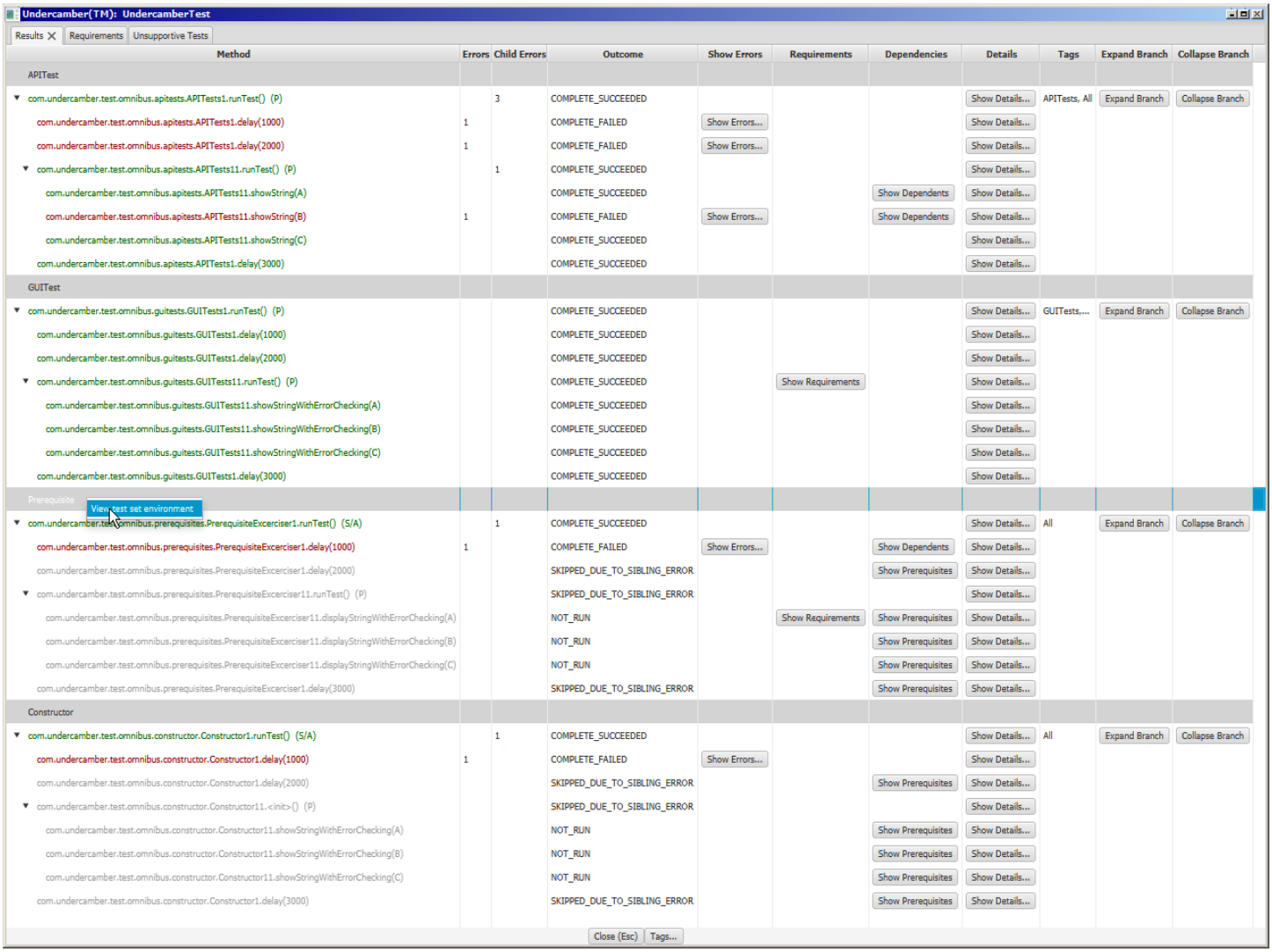

Figure 40: Test Set Pop-up Menu

Undercamber show the details of the test set:

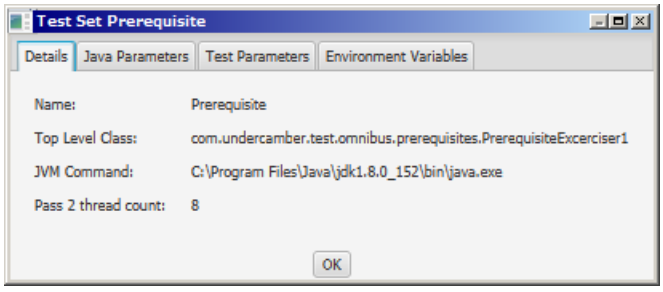

Figure 41: Test Set Details

The “Java Parameters” tab shows the parameters passed to the JVM for the second pass:

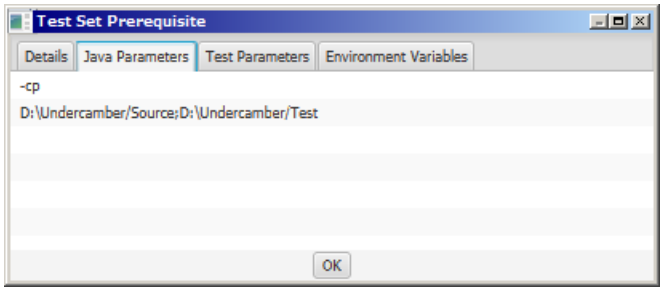

Figure 42: Pass 2 JVM Parameters

The “Test Parameters” tab shows the test parameters available to the test set:

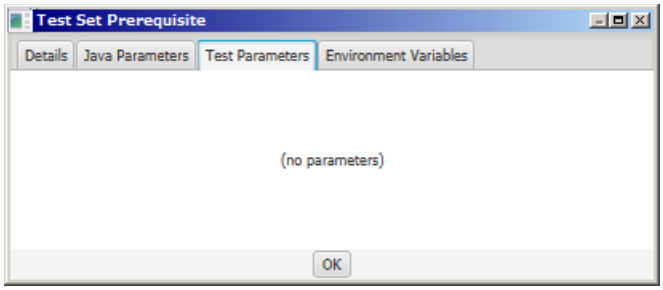

Figure 43: Test Parameters

The “Environment Variables” tab shows the environment variables used in the second pass:

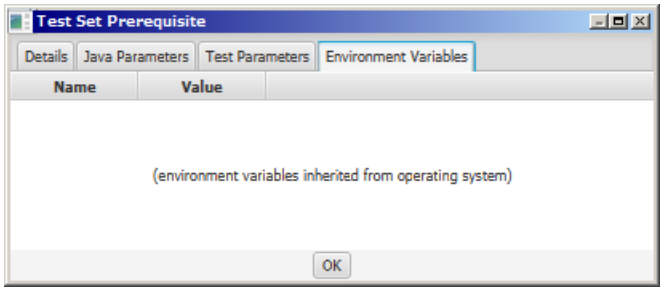

Figure 44: Pass 2 Environment Variables

Requirements Tab

Clicking on the “Requirements” tab at the top of the window shows the list of all of the requirements specified in the test suite:

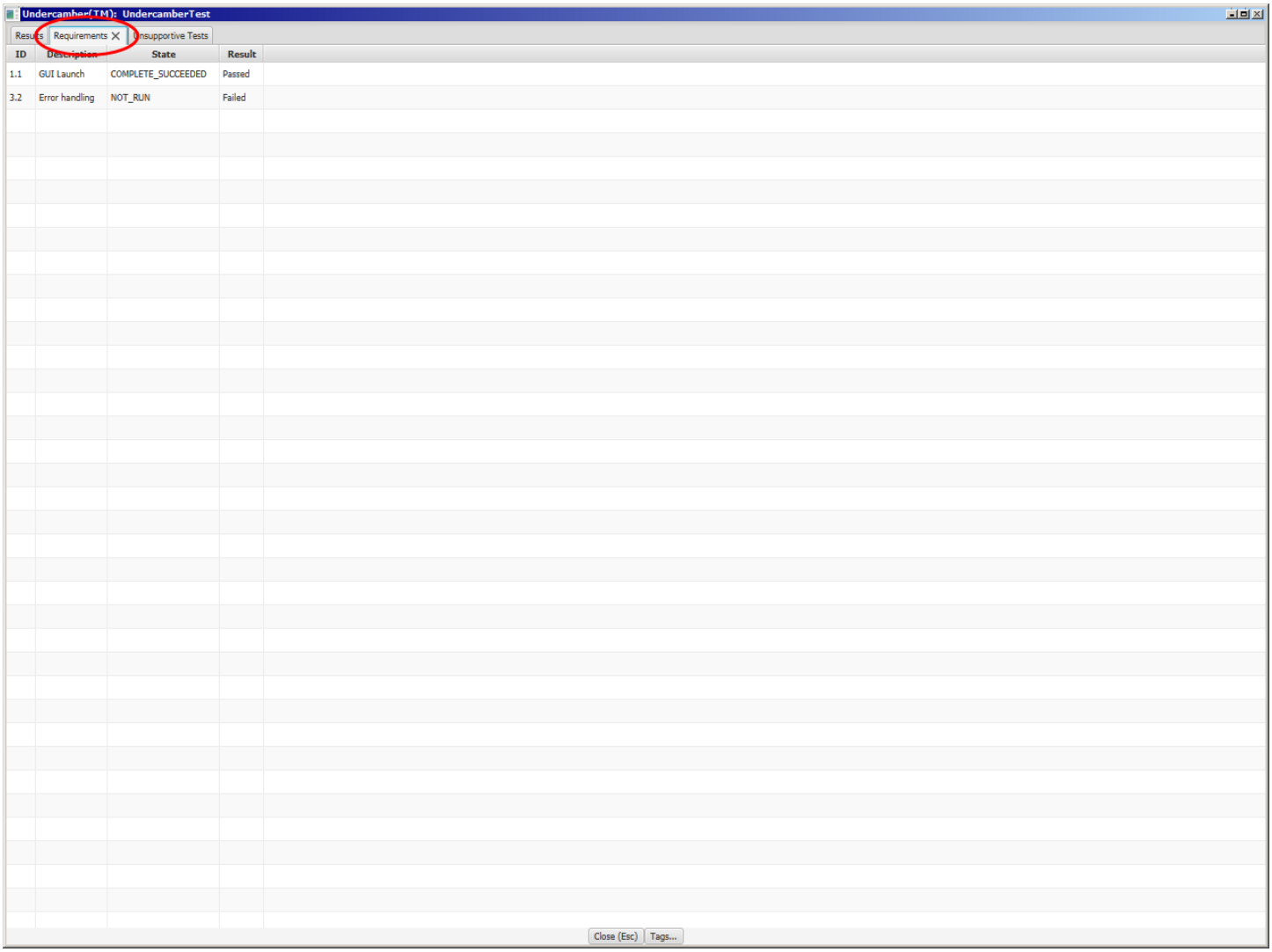

Figure 45: Requirements Tab

Unsupportive Tests Tab

Clicking on the “Unsupportive Tests” tab at the top of the window shows the list of all of the tests that do not support a requirement:

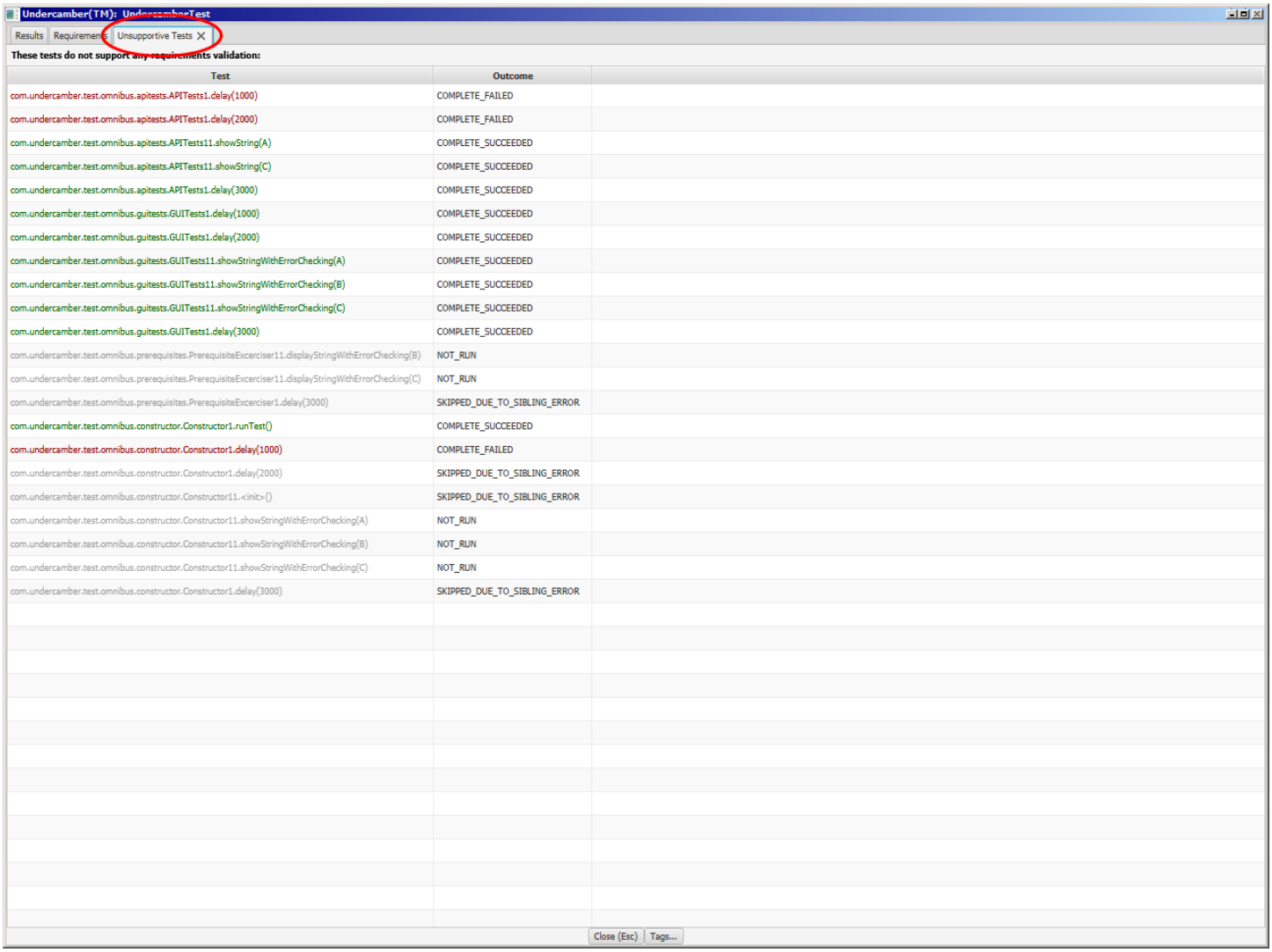

Figure 46: Unsupportive Tests Tab

Ideally, all tests support the validation of some requirement. This tab can be used to analyze the test suite for efficacy.

Requirements in Undercamber are described in more detail in “Requirements Verification”, above.

Tags Window

To show a list of all of the tags in the test suite, use the “Tags…” button:

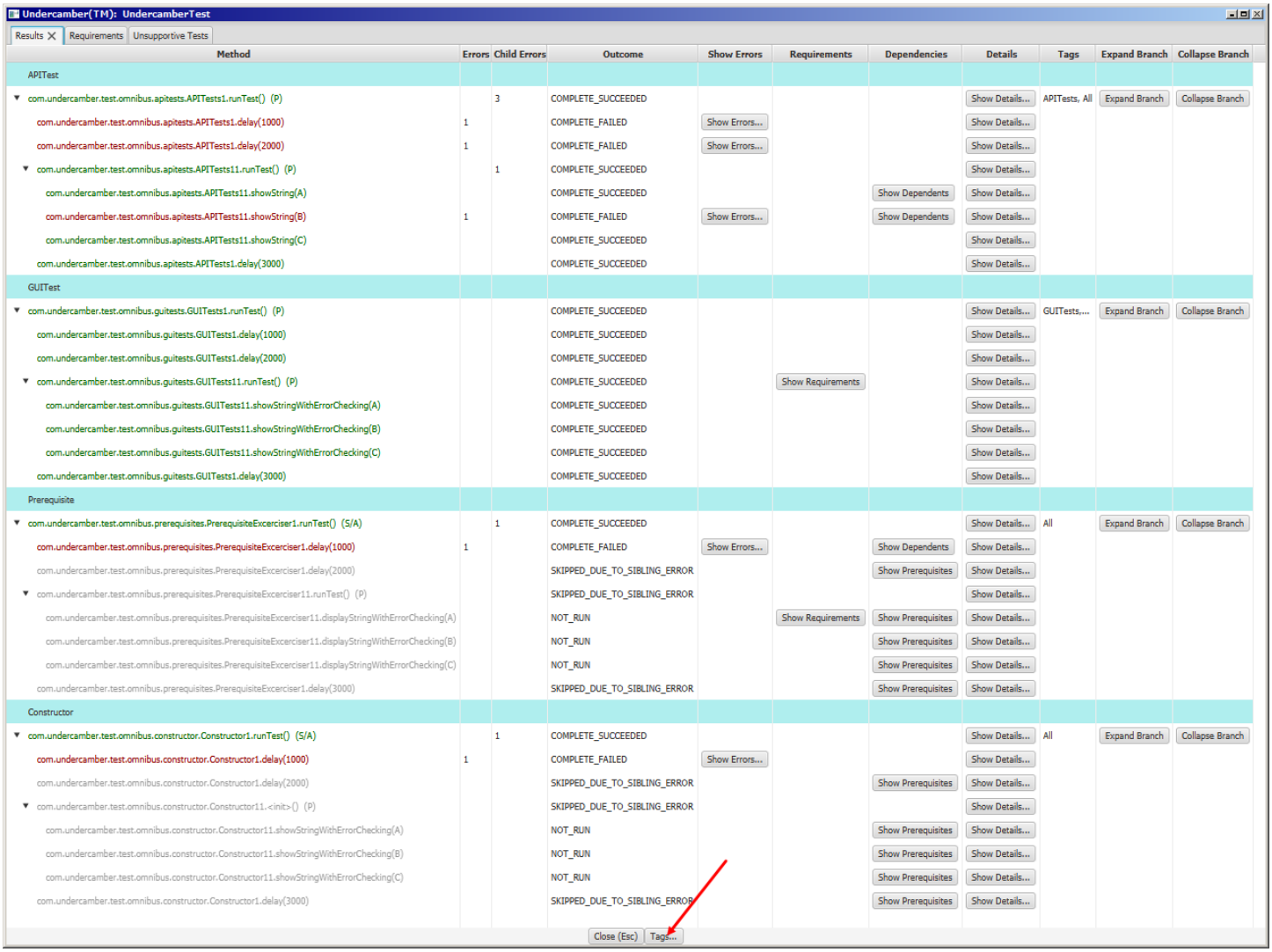

Figure 47: Tags Button

The “Tags…” button shows the list of tags:

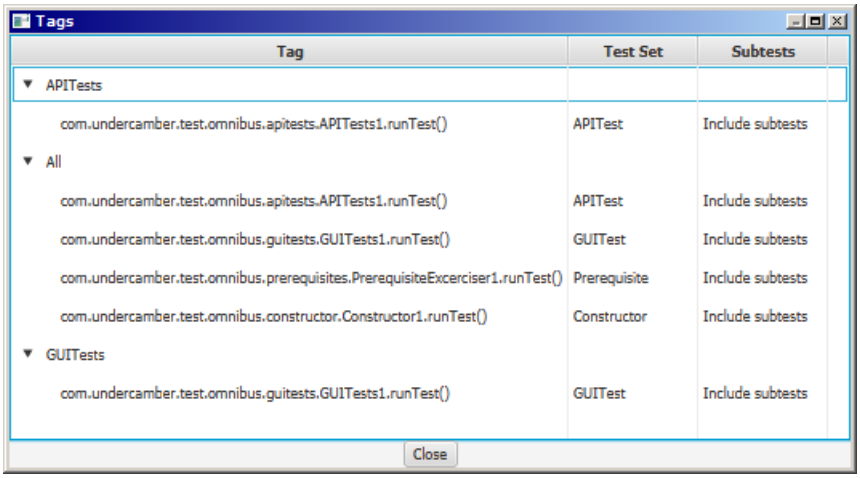

Figure 48: Tags List

The Tags window has a pop-up menu that allows the user to locate a test in the main results window:

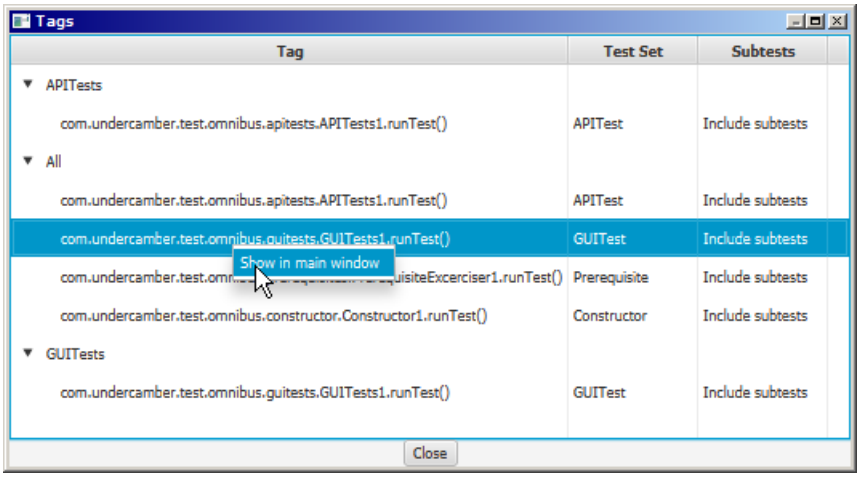

Figure 49: Tags Popup Menu

Skipping the Results Window

The results window can be skipped by any of several methods:

- Specify the `-resultWindow` option on the command line, as described below in “Command Line Arguments”.
- Call `Configurator.setShowResultsWindow(…)` attribute in the configurator, as described below in “Show or Hide the Results Screen”.
- Set the `UNDERCAMBER_SHOW_RESULTS_WINDOW` environment variable, as described below in “Environment Variables”.

If the results screen flag is specified multiple ways, Undercamber uses this search precedence:

1. The `-resultWindow` option on the command line, then
2. The call to `Configurator.setShowResultsWindow(…)`, then
3. The `UNDERCAMBER_SHOW_RESULTS_WINDOW` environment variable, then
4. The default is to show the results screen.

The Configurator

The Minimum

A very basic Undercamber configuration class might look like this:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.createTestSet();

      testSetBuilder.setTestSetName( "AdvancedTests" );
      testSetBuilder.setClass( com.mydomain.advanced.Tests.class );
      testSetBuilder.createTestSet();
   }
}
```

The configuration must specify the name of the test suite. Undercamber uses this name to create a directory for persistence.

This configuration illustrates the minimum to specify two test sets:

- The name of each test set must be specified:
  - The test set name is used for persistence. The persistence for each test set is stored in a separate file, and the file name is based on the name of the test set.
  - The names of the test sets must be unique.
- The class name is the name of the top-level test in the test set. This must be specified:
  - The specified class must implement the `com.undercamber.TestUnit` interface.
  - The specified class must have a no-argument constructor.
- Each test set will have a unique execution environment during the second pass.

Note: Do not name a test set `UndercamberMain`; this is a reserved name.

Output Directory

The configuration can optionally specify an output directory:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );
      configurator.setResultsRootDirectoryName( "D:\\TestRoot" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.createTestSet();

      testSetBuilder.setTestSetName( "AdvancedTests" );
      testSetBuilder.setClass( com.mydomain.advanced.Tests.class );
      testSetBuilder.createTestSet();
   }
}
```

See “Results Directory”, above, for a complete description of the output directory.

String Expansion in the Configuration

The above configuration has an obvious problem: The specified path is specific to Windows, but the tests will probably need to run on Unix-like operating systems, as well. Also, not every Windows machine will have a D: drive, and not every developer will have access to the `D:\TestRoot` directory.

To overcome this, most strings in the configuration are expanded using environment variables:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );
      configurator.setResultsRootDirectoryName( "${WIDGET_TEST_RESULTS_DIRECTORY}" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.createTestSet();

      testSetBuilder.setTestSetName( "AdvancedTests" );
      testSetBuilder.setClass( com.mydomain.advanced.Tests.class );
      testSetBuilder.createTestSet();
   }
}
```

With this capability, a developer can customize his/her machine for testing by setting the `WIDGET_TEST_RESULTS_DIRECTORY` to an appropriate location.

In the Undercamber configurator, the `%{…}` notation is used for all platforms, including Windows. The `%`, `{`, and `}` characters can be escaped using the `\` character.

Show or Hide the Results Screen

To show or hide the result screen, call the `Configurator.setShowResultsWindow(…)` method:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );
      configurator.setShowResultsWindow( false );

      testSetBuilder = configurator.getEmptyTestSetBuilder();
      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.createTestSet();
   }
}
```

This flag can be overridden, as described above in “Skipping the Results Window”.

Different JVM Versions

To specify the JVM to be used by a test set in the second pass, call one of the `TestSetBuilder.setJVMDirectoryName(…)` methods:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();
      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.setJVMDirectoryName( "${JAVA_HOME}" );
      testSetBuilder.createTestSet();
   }
}
```

On Windows systems, Undercamber will look for the `java.exe` file in the `bin` subdirectory below the specified directory. On other platforms, Undercamber will look for the `java` file in the `bin` subdirectory. For example:

- If the JVM directory is set to `C:\Program Files\Java\jdk1.8.0` on Windows, then Undercamber will look for `C:\Program Files\Java\jdk1.8.0\bin\java.exe`.
- If the JVM directory is set to `/usr/lib/jvm/jdk1.8.0` on Linux, then Undercamber will look for `/usr/lib/jvm/jdk1.8.0/bin/java`.

If the JVM directory is not specified, Undercamber will simply call `java`, without specifying any directory.

A test developer using different JVM versions should be aware of potential issues, as describe above in “Different JVMs” under “The Sequence of Operations”.

Java Parameters

In some test systems, different tests need to be run in different environments, so Undercamber runs different test sets in separate processes with separate JVMs. Each such process can be given a different Java configuration. In the Undercamber configuration, there are two different types of parameters that can be passed to the JVM: simple parameters and path-like parameters.

Use `TestSetBuilder.appendJavaParameter(…)` or `TestSetBuilder.prependJavaParameter(…)` to create simple parameters:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();
      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.appendJavaParameter( "-Xmx2048m" );
      testSetBuilder.createTestSet();
   }
}
```

Use `TestSetBuilder.appendJavaParameterPair(…)` or `TestSetBuilder.prependJavaParameterPair(…)` to add a name/value pair:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();
      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.appendJavaParameterPair( "--limit-modules", "myModule" );
      testSetBuilder.createTestSet();
   }
}
```

This is equivalent to:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();
      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.appendJavaParameter( "--limit-modules" );
      testSetBuilder.appendJavaParameter( "myModule" );
      testSetBuilder.createTestSet();
   }
}
```

com.undercamber.Path

Undercamber includes a convenience class for building path-like strings. The `com.undercamber.Path` class can be used to create platform-specific paths using platform-independent code. This code snippet illustrates how to use the com.undercamber.Path class to build a platform-dependent path string and use it in Undercamber:

```
com.undercamber.Path path = new com.undercamber.Path();
path.addEntries( "${WIDGET_ROOT}/Source",
                 "${WIDGET_ROOT}/Test",
                 "${WIDGET_ROOT}/Libraries/Undercamber.jar" );
testSetBuilder.appendJavaParameterPair( "-cp", path );
```

In addition to string expansion as illustrated above, the `Path` class will also by default verify that the entries point to a valid location on disk. This behavior can be altered by using an overloaded version of `Path.addEntries(…)`.

Note: If a classpath is specified for a test set, the Undercamber library must be included in the classpath.

Because each test set is run in a separate process, different test sets can have different configurations:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;
      com.undercamber.Path           path;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();
      testSetBuilder.setTestSetName( "FreeVersion" );
      testSetBuilder.setClass( com.myproject.test.FreeVersionTest.class );
      path = new Path( "${WIDGET_ROOT}/Source",
                       "${WIDGET_ROOT}/FreeVersion",
                       "${WIDGET_ROOT}/Test",
                       "${WIDGET_ROOT}/Libraries/Undercamber.jar" );
      testSetBuilder.appendJavaParameterPair( "-classpath", path );
      testSetBuilder.createTestSet();

      testSetBuilder = configurator.getEmptyTestSetBuilder();
      testSetBuilder.setTestSetName( "PaidVersion" );
      testSetBuilder.setClass( com.myproject.test.PaidVersionTest.class );
      path = new Path( "${WIDGET_ROOT}/Source",
                       "${WIDGET_ROOT}/PaidVersion",
                       "${WIDGET_ROOT}/Test",
                       "${WIDGET_ROOT}/Libraries/Undercamber.jar" );
      testSetBuilder.appendJavaParameterPair( "-classpath", path );
      testSetBuilder.createTestSet();
   }
}
```

Test Parameters in the Configurator

Test parameters are introduced in “Test Parameters”, above, and described in more detail in this section.

Use `TestSetBuilder.appendTestParameter(…)` or `TestSetBuilder.prependTestParameter(…)` to create parameters:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "FreeVersion" );
      testSetBuilder.setClass( com.myproject.test.FreeVersionTest.class );
      testSetBuilder.appendTestParameter( "-skipDeprecatedFeatures" );
      testSetBuilder.createTestSet();
   }
}
```

Use TestSetBuilder.appendTestParameterPair(…) and TestSetBuilder.prependTestParameterPair(…) to add a name/value pair to the test parameters:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "FreeVersion" );
      testSetBuilder.setClass( com.myproject.test.FreeVersionTest.class );
      testSetBuilder.appendTestParameterPair( "-testLevel", "maximum" );
      testSetBuilder.createTestSet();
   }
}
```

This is equivalent to:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "FreeVersion" );
      testSetBuilder.setClass( com.myproject.test.FreeVersionTest.class );
      testSetBuilder.appendTestParameter( "-testLevel" );
      testSetBuilder.appendTestParameter( "maximum" );
      testSetBuilder.createTestSet();
   }
}
```

To specify a platform-specific path-like parameter using platform-independent code, use the `com.undercamber.Path` class:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;
      com.undercamber.Path           path;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "FreeVersion" );
      testSetBuilder.setClass( com.myproject.test.FreeVersionTest.class );
      path = new Path( "${TEST_ROOT}/Keystrokes.dat",
                       "${TEST_ROOT}/MouseGestures.dat" );
      testSetBuilder.appendJavaParameterPair( "inputFiles", path );
      testSetBuilder.createTestSet();
   }
}
```

Setting Environment Variables

To specify the environment variables available to the UUT in the second pass, use `TestSetBuilder.addEnvironmentVariable(…)`:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "FreeVersion" );
      testSetBuilder.setClass( com.myproject.test.FreeVersionTest.class );
      testSetBuilder.addEnvironmentVariable( "PROJECT_ROOT", "${WIDGETS_ROOT}/source" );
      testSetBuilder.addEnvironmentVariable( "DATA_ROOT", "${WIDGETS_ROOT}/inputData" );
      testSetBuilder.createTestSet();
   }
}
```

In the above example, the environment variable strings use string expansion. These references are populated using the environment variables that Undercamber inherits from the operating system; not from the environment variables specified by the `TestSetBuilder`.

To specify path-like variables in a platform-independent fashion, use `com.undercamber.Path`:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;
      com.undercamber.Path           path;

      configurator.setSuiteName( "ExampleTestSuite" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "FreeVersion" );
      testSetBuilder.setClass( com.myproject.test.FreeVersionTest.class );
      testSetBuilder.addEnvironmentVariable( "PROJECT_ROOT", "${WIDGETS_ROOT}/source" );
      testSetBuilder.addEnvironmentVariable( "DATA_ROOT", "${WIDGETS_ROOT}/inputData" );
      path = new Path( "${WIDGETS_ROOT}/bin",
                       "${WIDGETS_ROOT}/Libraries" );
      testSetBuilder.addEnvironmentVariable( "-path", path );
      testSetBuilder.createTestSet();
   }
}
```

The environment variables specified by `TestSetBuilder.addEnvironmentVariable(…)` are applied only to the second pass. During the first pass, Undercamber uses the environment inherited from the operating system.

Important Notes on Environment Variables

- If a test set does not specify environment variables, then that test set will use the environment variables inherited from the operating system.
- If a test set specifies environment variables, then that test set will have only the specified environment variables, and no environment variables will be inherited from the operating system.
- In the configurator, the `%{…}` notation embedded in parameters is replace by environment variables, as described above in “String Expansion in the Configuration”. All such string expansion is done using the environment variables that Undercamber inherits from the operating system. The string expansion does not use the environment variables specified in `TestSetBuilder`.
- The environment variables specified by the configurator are not applied to the first pass. Instead, the first pass runs using the environment variables inherited from the operating system. The environment variables specified in the calls to addEnvironmentVariable(…) section are applied only during the second pass.

Thread Pool Size

To specify the thread pool size for the first pass (the discovery pass), use `Configurator.setPass1ThreadCount(…)`:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );
      configurator.setPass1ThreadCount( 8 );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.createTestSet();
  }
}
```

To specify different thread pool sizes for different test sets, use `TestSetBuilder.setPass2ThreadCount(…)`:

```
public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "ExampleTestSuite" );
      configurator.setPass1ThreadCount( 8 );

      testSetBuilder = configurator.getEmptyTestSetBuilder();

      testSetBuilder.setTestSetName( "BasicTests" );
      testSetBuilder.setClass( com.mydomain.basic.Tests.class );
      testSetBuilder.setPass2ThreadCount( 4 );
      testSetBuilder.createTestSet();
  }
}
```

See “Specifying Thread Count”, above, for more information on the thread pool size.

Command Line Arguments

The syntax for Undercamber is

```
java com.undercamber.Undercamber [flag [parameter]...]...
```

The only required flag is `-config`. All other flags are optional.

The number of required parameters for each flag depends on the flag. The supported flags are:

Flag	Parameter #1	Parameter #2	Parameter #3	Parameter #4	Description
-config	The name of the configuration class.	-	-	-	The name of the configuration class. This is required. The configuration class must implement com.undercamber.ConfigurationCallback. The configuration class must have a no-argument constructor.
-a	-	-	-	-	Run all tests. If this option is specified, the selection window will not be shown. This might be useful for an automated regression test.
-g	-	-	-	-	Run without the selection window. Run the tests selected in the last GUI run of Undercamber. This useful for repeatedly running an ad-hoc group of tests.
-rootDirectory	Output root directory name	-	-	-	The directory in which Undercamber will write its outputs. See “Output Root Directory”, above, for more information.
-subdirectory	Output subdirectory name	-	-	-	The name of the subdirectory in which Undercamber will write its outputs. If this is not specified, Undercamber will create a subdirectory name from the current date and time. See “Output Subdirectory”, above, for more information.
-threadCount	Thread pool size	-	-	-	The size of the thread pool. This overrides the `threadCount` attribute in the configurator. If the thread pool size is not specified, Undercamber will use the number of processor cores available to the JVM. See “Specifying Thread Count”, above, for more information.
-p	Parameter value	-	-	-	A test parameter to be passed to the tests. In Undercamber, test parameters specified on the command line are first in the list of parameters available to the tests, and parameters specified in the configurationor are last. See “Test Parameters”, above, for more information.
-pp	Flag	Parameter value	-	-	A parameter pair to be passed to the tests. This is the same as passing the flag and value separately (but consecutively and in order) using the –p option twice. See “Test Parameters”, above, for more information.

Flag	Parameter #1	Parameter #2	Parameter #3	Parameter #4	Description
-config	The name of the configuration class.	-	-	-	The name of the configuration class. This is required. The configuration class must implement com.undercamber.ConfigurationCallback. The configuration class must have a no-argument constructor.
-tag1	Comma-separated list of tag names	-	-	-	This command line argument is used to specify that all of the tests with the specified tags, and their prerequisites, will be run without showing the selection window. See "Tags", above, for more information.
-tag2	Comma-separated list of test set names	Comma-separated list of tag names	-	-	Run all tests with the specified tags. Limit the search for tags to the specified test sets. All of the tests with the specified tag, and their prerequisites, will be run without showing the selection window. See "Tags", above, for more information.
-test1	Class name	Method name	-	-	This specifies which test to run. Works only with tests that have no 'argument', as described above in "Test Uniqueness". When this option is used, the specified test, all of its subtests, and all of their prerequisites will be automatically run, without showing the selection window.
-test2	Class name	Method name	Test argument	-	This specifies which test to run. Works only with tests that include an 'argument', as described above in "Test Uniqueness". When this option is used, the specified test, all of its subtests, and all of their prerequisites will be automatically run without showing the selection window.
-test3	Comma-separated list of test set names	Class name	Method name	-	This specifies which test to run. Limit the search for tests to the specified test sets. Works only with tests that have no 'argument', as described above in "Test Uniqueness". When this option is used, the specified test, all of its subtests, and all of their prerequisites will be automatically run, without showing the selection window.
-test4	Comma-separated list of test set names	Class name	Method name	Test argument	This specifies which test to run. Limit the search for tests to the specified test sets. Works only with tests that include an 'argument', as described above in "Test Uniqueness". When this option is used, the specified test, all of its subtests, and all of their prerequisites will be automatically run without showing the selection window.
-set	Test set name	-	-	-	This command line argument specifies that all of the tests in the specified test set, and their prerequisites, should be run.
-resultWindow	`true` or `false`	-	-	-	Flag indicating whether the results screen should be displayed.
-forcePrerequisites	-	-	-	-	Run all prerequisites, including previously satisfied prerequisites. See "Previously Satisfied Conditional Prerequisites", above, for more information.
-fp					Same as `-forcePrerequisites`

The command line can specify combinations of `-tag1`, `-tag2`, `-test1`, `-test2`, `-test3`, `-test4`, `-set`, and `-a` parameters. In this case, Undercamber will run the union of the specified tests, in the order specified by the calls to `TestManager.addSubtest(…)`.

Examples

- To run using the `com.mypackage.MyConfiguration` class:

```
java com.undercamber.Undercamber -config com.mypackage.MyConfiguration
```

- To use the `com.mypackage.MyConfiguration` configurator class and run all tests without displaying the selection window:

```
java com.undercamber.Undercamber -config com.mypackage.MyConfiguration
```

- To run the `myTest` method in class `com.mydomain.Tests`, its subtests, and their prerequisites, without displaying the selection window:

```
java com.undercamber.Undercamber -config com.mypackage.MyConfiguration -test1 com.mydomain.Tests myTest
```

- To run the `myTest` and `yourTest` methods in class `com.mydomain.Tests`, their subtests, and their prerequisites, without displaying the selection window:

```
java com.undercamber.Undercamber -config com.mypackage.MyConfiguration -test1 com.mydomain.Tests myTest -test1 com.mydomain.Tests yourTest
```

- To run the `myTest` method in class `com.mydomain.Tests`, its subtests, all tests tagged `Advanced`, and all of their prerequisites, without displaying the selection window:

```
java com.undercamber.Undercamber -config com.mypackage.MyConfiguration -test1 com.mydomain.Tests myTest -tag Advanced
```

- To pass the test parameters `-resources` and `~/TestDirectory` (as a pair) to all tests and suppress the results window:

```
java com.undercamber.Undercamber -config com.mypackage.MyConfiguration -pp -resources ~/TestDirectory -resultScreen false
```

- To place the output in the `D:\Test` directory, in the Undercamber-supplied default subdirectory:

```
java com.undercamber.Undercamber -config com.mypackage.MyConfiguration -rootDirectory D:\Test
```

- To place the output in the `~/Test` directory, in a subdirectory based on the current date (this works only on Unix-like operating systems):

```
java com.undercamber.Undercamber -config com.mypackage.MyConfiguration -rootDirectory ~/Test -subdirectory $(date '+%Y-%m-%d')
```

- To place the output in the `D:\Test` directory, in a subdirectory based on the current date and time (this hack works only on Windows):

```
for /F "usebackq tokens=1,2 delims==" %%i in (`wmic os get LocalDateTime /VALUE 2^>NUL`) do if '.%%i.'=='.LocalDateTime.' set LDT=%%j
set LDT=%LDT:~0,4%-%LDT:~4,2%-%LDT:~6,2%-%LDT:~8,2%-%LDT:~10,2%-%LDT:~12,6%
java com.undercamber.Undercamber -config com.mypackage.MyConfiguration -rootDirectory D:\Test -subdirectory %LDT%
```

Environment Variables

Undercamber uses these environment variables:

Environment Variable	Description
`UNDERCAMBER_THREAD_COUNT`	Specifies the number of threads in Undercamber's thread pool. This is not required. See "Specifying Thread Count", under "Concurrent Execution", above, for more information about the thread count.
`UNDERCAMBER_ROOT`	Specifies the root directory for Undercamber's output. This is required if the output root directory is not specified elsewhere, as described in "Results Directory", above.
`UNDERCAMBER_SHOW_RESULTS_WINDOW`	Set this to either "`true`" or "`false`". Indicates whether Undercamber should display the results screen. This is not required. The default is "`true`". This is described above in "Skipping the Results Window".

A More Complete Example

This section does not introduce any new features or concepts. Instead, it is simply a more complete, more realistic illustration of how to use Undercamber, with a (slightly) more realistic UUT (Unit Under Test).

This example contains these classes:

`MyCalculator`	This is the UUT. It is a crude 'calculator' with just addition and division. It records its operations to a text file.
`MyTopLevelTest`	This is the top level test in the Undercamber hierarchy of tests. It is an executive, and does not perform any verifications itself.
`ConfigurationCallback`	This is the configurator.
`AdditionChecker`	This is a few tests to verify the addition capability of the calculator
`DivisionChecker`	This is a few tests to verify the division capability of the calculator

The UUT

This is the UUT, a crude recording calculator:

```
package com.mydomain.test;

import java.io.*;

public class MyCalculator
{
   java.io.PrintStream _printStream;

   MyCalculator( File outputFile )
      throws IOException
   {
      _printStream = new PrintStream( outputFile );
   }

   public double add( double addend1,
                      double addend2 )
      throws IOException
   {
      double sum;

      sum = addend1 + addend2;

      _printStream.println( Double.toString(addend1) + " + " + addend2 + " = " + sum );

      return sum;
   }

   public double divide( double numerator,
                         double denominator )
   {
      double dividend;

      if ( denominator == 0.0 )
      {
         throw new RuntimeException( "Divide by zero" );
      }

      dividend = numerator / denominator;

      _printStream.println( Double.toString(numerator) + " / " + denominator + " = " + dividend );

      return dividend;
   }

   public void close()
      throws IOException
   {
      _printStream.close();
   }
}
```

Top Level Test

Below is the top-level Undercamber test, `TopLevelTest.java`. This is just an executive that invokes other tests, and does not perform any verification itself. It illustrates:

- How to spread tests across multiple classes (using Java's ordinary lambda expressions).
- How to use the Undercamber-supplied working directory (See "User Directory", above).

This is the source code:

```
package com.mydomain.test;

import com.undercamber.*;

public class MyTopLevelTest
   implements TestUnit
{
   public void runTest( TestManager testManager )
      throws Throwable
   {
      boolean      verify;
      java.io.File localDirectory;

      verify = testManager.initialize();

      if ( verify )
      {
         localDirectory = testManager.getUserWorkingDirectory();
      }
      else
      {
         localDirectory = null;
      }

      testManager.addSubtest( tm -> new AdditionChecker(tm,
                                                        localDirectory) );

      testManager.addSubtest( tm -> new DivisionChecker(tm,
                                                        localDirectory) );
   }
}
```

Addition Checker

`AdditionChecker.java` tests the addition function in the UUT. It illustrates:

- How to pass a unique 'argument' to `TestManager.initialize(…)` (See "Test Uniqueness", above).
- A strategy to use the work directory created by Undercamber (See "User Directory", above).
- How to tag a group of tests (See "Tags", above).

This is the source code:

```
package com.mydomain.test;
import com.undercamber.*;
import java.io.*;

public class AdditionChecker
{
   AdditionChecker( TestManager  testManager,
                    java.io.File parentDirectory )
      throws Throwable
   {
      boolean      verify;
      java.io.File localDirectory;
      MyCalculator calculator;

      verify = testManager.initialize( new Tag("Addition") );

      if ( verify )
      {
         localDirectory = new File( parentDirectory, "DivisionChecker" );

         localDirectory.mkdirs();

         calculator = new MyCalculator( new java.io.File(localDirectory,"CalculatorOutput.txt") );
      }
      else
      {
         calculator = null;
      }

      testManager.addSubtest( tm -> checkAddition(tm,calculator,2,2,4) );
      testManager.addSubtest( tm -> checkAddition(tm,calculator,3,6,9) );
   }

   private void checkAddition( TestManager  testManager,
                               MyCalculator calculator,
                               double       addend1,
                               double       addend2,
                               double       expectedResult )
      throws Throwable
   {
      boolean verify;
      double  result;

      verify = testManager.initialize( Double.toString(addend1) + "," + addend2 + "," + expectedResult );

      if ( verify )
      {
         result = calculator.add( addend1, addend2 );

         if ( result != expectedResult )
         {
            testManager.addException( new Exception("Incorrect result.  Expected = "+expectedResult+".  Found = "+result) );
         }
      }
   }
}
```

Division Checker

`DivisionChecker.java` tests the division function in the UUT. It illustrates:

- How to pass an 'argument' to `TestManager.initialize(…)` (See "Test Uniqueness", above).
- A strategy to use the work directory created by Undercamber (See "User Directory", above).
- How to set up a negative test in Undercamber (See "Negative Tests", above).
- How to tag a group of tests (See "Tags", above).

This is the source code:

```
package com.mydomain.test;

public class DivisionChecker
{
   DivisionChecker( com.undercamber.TestManager testManager,
                    java.io.File                parentDirectory )
      throws Throwable
   {
      boolean      verify;
      java.io.File localDirectory;
      MyCalculator calculator;

      verify = testManager.initialize( new com.undercamber.Tag("Division") );

      if ( verify )
      {
         localDirectory = new java.io.File( parentDirectory, "DivisionChecker" );
         localDirectory.mkdirs();
         calculator = new MyCalculator( new java.io.File(localDirectory,"CalculatorOutput.txt") );
      }
      else
      {
         calculator = null;
      }

      testManager.addSubtest( tm -> checkDivision(tm,calculator,2,2,1) );
      testManager.addSubtest( tm -> checkDivision(tm,calculator,18,6,3) );
      testManager.addSubtest( tm -> checkDivideByZero(tm,calculator) );
   }

   private void checkDivision( com.undercamber.TestManager testManager,
                               MyCalculator                calculator,
                               double                      numerator,
                               double                      denominator,
                               double                      expectedResult )
      throws Throwable
   {
      boolean verify;
      double  result;

      verify = testManager.initialize( Double.toString(numerator) + "," + denominator + "," + expectedResult );

      if ( verify )
      {
         result = calculator.divide( numerator, denominator );
         if ( result != expectedResult )
         {
            testManager.addException( new Exception("Incorrect result.  Expected = "+expectedResult+".  Found = "+result) );
         }
      }
   }

   private void checkDivideByZero( com.undercamber.TestManager testManager,
                                   MyCalculator                calculator )
      throws Throwable
   {
      boolean verify;

      verify = testManager.initialize();

      if ( verify )
      {
         try
         {
            calculator.divide( 1, 0 );
            testManager.addException( new Exception("Did not get expected divide-by-zero exception") );
         }
         catch ( RuntimeException expected )
         {
         }
      }
   }
}
```

The Configurator

This configuration class will work with the example:

```
package com.mydomain.test;

public class ConfigurationCallback
   implements com.undercamber.ConfigurationCallback
{
   final public void configure( com.undercamber.Configurator configurator )
      throws Throwable
   {
      com.undercamber.TestSetBuilder testSetBuilder;

      configurator.setSuiteName( "CalculatorTest" );

      testSetBuilder = configurator.getEmptyTestSetBuilder();
      testSetBuilder.setTestSetName( "CalculatorTest" );
      testSetBuilder.setClass( MyTopLevelTest.class );
      testSetBuilder.createTestSet();
   }
}
```

Running the Example

Setup

1. Make sure Java 8 or later is installed and in your path.
2. Make sure the Undercamber JAR file is in your classpath.
3. Enter the code above for the Java classes. Make sure they are in your classpath, in a subdirectory that is appropriate for the Java package.

Run

- To run this example, use this command line:

```
java com.undercamber.Undercamber –config com.mydomain.test.ConfigurationCallback
```

- To run the tagged addition tests without displaying the selection window and without the results window, use this command line:

```
java com.undercamber.Undercamber –config com.mydomain.test.ConfigurationCallback –tag Addition –resultScreen false
```

- To run just the first division check without showing the selection window, use this command line:

```
java com.undercamber.Undercamber –config com.mydomain.test.ConfigurationCallback –test2 com.mydomain.test.DivisionChecker checkDivision 2.0,2.0,1.0
```

Legal Notices

"Undercamber" and the airfoil logo are trademarks of Rygaard Technologies, LLC.

Rygaard Technologies, LLC reserves the right to make changes to this document and its contained information without notice. This document does not represent any commitment on the part of Rygaard Technologies, LLC.

Rygaard Technologies, LLC makes no warranty of any kind regarding the information in this document. No statement or representation in this document may be deemed a warranty by, or lead to liability of, Rygaard Technologies, LLC. Rygaard Technologies, LLC shall not be held liable for any damages whatsoever resulting from the information in this document.

The Undercamber software is licensed under this agreement:

Copyright 2018 Rygaard Technologies, LLC

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
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. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS 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 COPYRIGHT HOLDER OR 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.