



# *AgitarOne Test Generation User Guide*

Version 6.0

May 2013

---



41 Sharpe Drive  
Cranston, RI 02920 USA

[www.agitar.com](http://www.agitar.com)

Copyright © 2013 Agitar® Technologies, Inc. All rights reserved.

**Ownership of Materials.** This manual, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The contents of this manual are furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Agitar. Agitar assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.

This manual is protected by copyright and distributed under licenses restricting its use, copying, translation, distribution, and decompilation. Except as permitted by such licenses, no part of this manual may be reproduced in any form by any means without prior written authorization of Agitar. Except as expressly provided herein, Agitar grants no express or implied rights to anyone under any patents, copyrights, trademarks, trade names, or trade secret information with respect to the contents of the manual.

The overall software distribution from Agitar includes software licensed from third parties, covered by copyrights as described in their license agreements.

**Ownership of Trademarks.** The trademarks, service marks, product names, company names or logos and other marks displayed in the manual are the property of Agitar Technologies, Inc. or other third parties. Any use of trademarks, service marks, product names, company names or logos, and other marks, including the reproduction, modification, distribution, or republication of same without the prior written permission of the owner is strictly prohibited.

Agitar and Agitator are registered trademarks of Agitar Technologies; AgitarOne, Software Agitation, JUnit Factory, and AutoMock are trademarks of Agitar Technologies, Inc. Other trademarks, service marks, trade names, and company logos referenced are the property of their respective owners.

**Disclaimers.** This manual is provided “as is” and all express or implied conditions, representations, and warranties, including any implied warranty of merchantability, fitness for a particular purpose, or non-infringement, are disclaimed, except to the extent that such disclaimers are held to be legally invalid. Further Agitar does not warranty, guarantee, or make any representations regarding the use, or the results of the use, of the written material in terms of correctness, accuracy, reliability, or otherwise.

# Table of Contents

Preface .....	vii
Audience .....	vii
Typographic Conventions .....	vii
What’s in this Guide? .....	viii
1 Introducing AgitarOne Test Generation .....	1
The Test Generation Process .....	1
Influencing Test Generation .....	2
Using Generated Tests in Continuous Integration .....	3
Changes That Characterization Tests Detect .....	3
2 Generating and Running AgitarOne JUnit Tests .....	5
Generating Tests .....	6
Specifying the Location of Generated Tests .....	7
Generating Missing Tests .....	8
Specifying the Location of Native Code for Test Generation Projects .....	9
Test Generation and Java Security .....	10
About Tests Generated by AgitarOne .....	11
Agitation Artifacts and Generated Tests .....	12
Running Generated Tests .....	12
Rerunning Tests .....	13
Understanding Code Coverage Markers .....	14
Setting Coverage Display Options .....	14
Which Test Classes Cover Which Source Code? .....	17
Saving Regression Suites .....	17
The Super-Runner Ant Task .....	18
Monitoring Progress Over Time .....	18
Troubleshooting Test Generation .....	18
Test Run Times Out With “Too Many Threads” Message .....	18
3 Reviewing the Generated Tests .....	21
Basic Structure of a JUnit Test .....	21
A Look at the Generated Tests .....	22

Learning About Your Code from Tests. . . . .	23
Names of Test Methods . . . . .	24
Why Is My Test Method Named <code>test*WithAggressiveMocks()</code> ? . . . . .	25
Generated Test Values . . . . .	26
Missing Code Coverage . . . . .	26
About Mock Objects in Generated Tests . . . . .	27
Reading Generated Tests That Use Mockingbird . . . . .	28
Evaluating Generated Tests. . . . .	30
What Gets Mocked in a Generated Test . . . . .	31
Items That Are Never Mocked . . . . .	31
Advantages of Testing With Real Objects. . . . .	32
4 Handling Test Failures . . . . .	33
Diagnosing Test Failures. . . . .	33
Platform Dependencies and Cross-Platform Differences . . . . .	34
Tests That Depend On Run Order . . . . .	35
Running Your Own Tests with the Super-Runner. . . . .	35
Understanding <code>TestException</code> Messages. . . . .	35
Unexpected Method Called . . . . .	36
Recorded Value Not Used . . . . .	36
When You Will See “Value Not Used” Messages . . . . .	36
<code>TestException</code> Example: Recorded Value Not Used . . . . .	37
Handling Test Failures From a Dashboard Report . . . . .	39
Importing Test Failures from a Dashboard . . . . .	40
Importing Test Failures from a Developer Dashboard . . . . .	42
Re-Generating Failing Tests . . . . .	43
Deleting Invalid Failing Tests. . . . .	44
5 Tuning AgitarOne Test Generation . . . . .	47
Getting Suggestions on How to Improve Tests. . . . .	47
Understanding Test Helpers . . . . .	48
Test Helper General Requirements . . . . .	50
Improving Test Generation with Test Data Helpers . . . . .	50
About Test Data Helpers . . . . .	51
How AgitarOne Test Generation Selects Test Data Helpers . . . . .	51
Writing Test Data Helpers . . . . .	52

Test Data Helper Examples .....	54
Global Test Data Helpers.....	54
Scoped Test Data Helpers .....	54
Improving Assertions in Generated Tests.....	55
Assertion Helper Methods .....	56
When Assertions Fail.....	57
Observation Helpers .....	59
Specifying Complex Setup with Setup Helpers .....	60
Using Concrete Mocks in Test Generation.....	62
Preventing Assertions and Automatic Mocking.....	63
Setup Helper Example .....	63
Using the MockRunner Library .....	64
6 Intercepting Method Calls During Test .....	67
Method Interception APIs.....	67
MethodInterceptor Interface.....	68
Working with the MethodInterceptor Interface.....	68
InterceptorWithDefaultValue Class .....	69
InterceptorWithFactory Class.....	69
Interceptor Class .....	70
The Process of Programming Method Interception Using the MethodInterceptor Interface...	70
Method Interceptor Examples.....	71





# Preface

The *AgitarOne Test Generation User Guide* describes the test generation client and how to use it to generate tests for your Java code.

This preface contains the following information:

- [Audience](#)
- [Typographic Conventions](#)
- [What's in this Guide?](#)

---

## Audience

Agitar's tools enable software development organizations to test and manage Java applications throughout the development lifecycle. This includes the following types of users:

- *Developers*—Who build and test Java classes.
- *QA/Test Engineers*—Who validate that new versions of the software still comply with the original specification.
- *Development Managers*—Who want to discover defects early in the lifecycle.

The Agitar documentation assumes that readers are familiar with the software development process, Java syntax, and the Eclipse platform.

---

## Typographic Conventions

Agitar manuals use the following typographic conventions:

<b>bold text</b>	File names, XML elements, user interface controls, and language keywords.
<b><i>bold italic text</i></b>	Variable elements; for example, parameters in code syntax.
<i>italic text</i>	New terminology; also emphasized words and book titles.

plain text	Names of keyboard keys.
monospace text	Examples, such as Java or XML code; or text you type exactly as it appears.

Descriptions of procedures also use the following conventions:

<b>File&gt;Import</b>	A menu path to follow; in this example, from the <b>File</b> menu, select <b>Import</b> .
Ctrl+C	Press both keys at the same time. Names of keyboard keys appear in plain text.
ESC S C	Press and release each key in succession.

---

## What's in this Guide?

This guide contains the following chapters:

- [Chapter 1, “Introducing AgitarOne Test Generation,”](#) describes the basics of test generation and how it can help your development team implement a developer testing program.
- [Chapter 2, “Generating and Running AgitarOne JUnit Tests,”](#) shows you how to generate JUnit tests for your Java code.
- [Chapter 3, “Reviewing the Generated Tests,”](#) shows examples of tests generated by AgitarOne test generation and how you can use the generated tests to learn more about your code.
- [Chapter 4, “Handling Test Failures,”](#) explains how to understand test failures.
- [Chapter 5, “Tuning AgitarOne Test Generation,”](#) describes the kinds of helper classes you can create to improve generation of both test data and assertions.
- [Chapter 6, “Intercepting Method Calls During Test,”](#) describes method interceptors and the details of programming alternate behavior for problem classes.



# 1

## Introducing AgitarOne Test Generation

AgitarOne test generation is a server-based automated software test generation service that works with a client installed in your IDE. The AgitarOne server generates JUnit tests—the industry standard for developer testing for Java code.

When a test generation client sends Java classes to an AgitarOne server, the server analyzes the code and returns a set of tests that validate the current behavior of that code. These are *characterization tests*, because they describe what your code does, not necessarily what it is supposed to do.

You can use these tests as regression suites to detect behavior changes to your code over time. You can also use the generated tests to learn more about your code and to identify unexpected behavior.

AgitarOne test generation takes advantage of Agitar’s Mockingbird libraries to provide mock implementations of external services or complex objects required to test individual code units. The test generation client includes the Agitar test runner, which runs existing JUnit tests and those generated by AgitarOne test generation, reporting failures like the standard JUnit runner while providing additional information including the coverage achieved by the tests.

This chapter contains the following topics:

- [The Test Generation Process](#)
- [Influencing Test Generation](#)
- [Using Generated Tests in Continuous Integration](#)
- [Changes That Characterization Tests Detect](#)

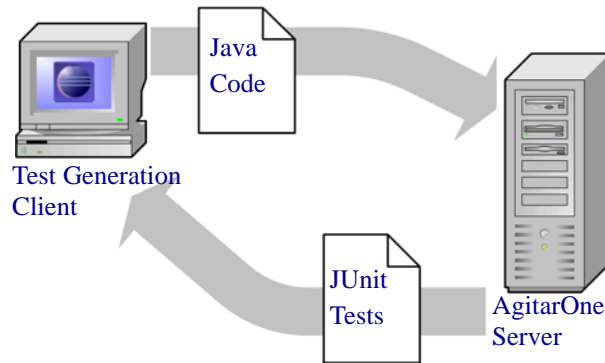
---

### The Test Generation Process

To request tests for one or more units of code, a test generation client sends Java classes and associated configuration files to an AgitarOne server (installed on Windows or Linux). The server performs both static and dynamic analysis on the code and sends a test class back to the client for each Java class.

AgitarOne test generation is especially effective for applications built with POJOs (Plain Old Java Objects) or J2EE using servlets, JDBC, or EJBs.

The following figure shows the client/server interaction.



The generated tests include test methods that validate both normal and exception outcomes, including outcomes detected but not explicitly declared in the code. These test methods use a variety of test values, making sure to test for boundary conditions identified during the code analysis. When necessary, the generated tests use mock objects in place of external services. However, test generation does not return tests for interfaces or for abstract classes with no concrete classes in the project.

The test-generation client includes an enhanced version of the Eclipse JUnit runner, called the Agitar super-runner, that recognizes all of the constructs used in the generated tests, including Mockingbird calls to define and use mock objects. In addition, the Agitar super-runner monitors coverage as it runs JUnit tests and reports coverage information alongside the source code in the Java editor.

---

## Influencing Test Generation

The tests produced by the AgitarOne server validate existing behavior of the code, whether that was the intended behavior or not. If you review the generated tests, you can learn more about how your code actually works. For example, reviewing the tests can help you:

- identify unexpected outcomes
- discover boundary conditions and side effects
- increase test coverage

In the course of reviewing the tests, you might find cases where test generation could produce better tests if it had test objects in a particular state. Or you might find places where you could add assertions based on information you have about the application's intended behavior.

You can add helper classes to provide additional information that would enable test generation to create more effective tests for your code. For examples, see [“Tuning AgitarOne Test Generation” on page 47](#).

---

## Using Generated Tests in Continuous Integration

Continuous integration is the practice where developers check in their code changes frequently and incorporate those changes into automated software builds so that they can receive rapid feedback about the results of those changes. Ideally, each code change is accompanied by one or more unit tests that validate the code's behavior. Agitar has identified the following benefits of practicing continuous integration:

- Measurable quality improvements, tracked by reports from the Management Dashboard
- High level of confidence for team members, product managers, and executives
- Ability to confidently add functionality until late in the cycle
- Better code design
- Learning environment for team members
- Higher morale of the whole team

When you have verified that tests produced by test generation pass and reflect the intended behavior of your code, you can check those tests into your version control system. Agitar test generation includes a custom Ant task (the super-runner task) that mirrors the functionality of the standard `junit` Ant task and runs the generated JUnit tests, including those that include Mockingbird calls. For more information, see [“Saving Regression Suites” on page 17](#).

---

## Changes That Characterization Tests Detect

AgitarOne generates what are called *characterization* tests. Such tests describe the current behavior of your code, not necessarily what the code *should* do.

Such tests are valuable both in describing current code behavior and in detecting changes as you enhance your project over time. The following kinds of changes are detected by running your AgitarOne-generated tests:

- return value changes

- field value changes
- new runtime exceptions
- changes involving **assertInvoked()** tests
  - ◆ failure to call a method expected by **assertInvoked()**
  - ◆ changes to the parameters passed to a method expected by **assertInvoked()**

To detect changes in return values and field values, AgitarOne must be able to make a local variable for these values. Such items include Java primitives, wrapper classes for primitives (**String**, **Integer**, etc.), or any object created or returned during the setup sequence. Items in which AgitarOne cannot detect changes include values initialized or set in classes called by the class under test.

# 2

## Generating and Running AgitarOne JUnit Tests

AgitarOne is a server-based automated software test generation service that works with your IDE. When you use the test generation client plug-in, it sends your classes to an AgitarOne server that generates tests for your code and sends those tests back to your IDE. The AgitarOne server can be local to your company or on the Internet.

When the AgitarOne server receives your Java classes, it performs both static and dynamic analysis of the code to determine its behavior. Then it generates a set of JUnit tests and returns them to your IDE. These are characterization tests, as they confirm the current behavior of your code. As you change the code over time, the generated tests serve as a regression-detection suite. You can quickly see when code changes cause test failures, indicating that previous behavior has changed. You can review these failures to see whether code changes have the intended side-effects and you can fix the code if they don't.

You can review the generated tests to see what they test. You can also provide hints to AgitarOne that guide the generation of test data and assertions in future tests. When you have tests that accurately reflect the way you want to unit-test your code, you can save those tests in your version control system so that they do not get overwritten if you regenerate tests for your code.

This chapter contains the following topics:

- [Generating Tests](#)
- [Running Generated Tests](#)
- [Rerunning Tests](#)
- [Understanding Code Coverage Markers](#)
- [Saving Regression Suites](#)
- [Troubleshooting Test Generation](#)

## Generating Tests

If you have never written a JUnit test, consider writing a few before using the AgitarOne server to generate tests for you; this will give you a better idea of what AgitarOne test generation can do for you. If you are already familiar with JUnit, then feel free to jump right in.

### ABOUT PLATFORM DEPENDENCE AND GENERATED TESTS

If the AgitarOne client and the AgitarOne server are on different platforms, you might sometimes get a test back from the server that fails when run. This is because tests that access platform-specific items and run reliably on the server might not get the same results when run on the client.

If you experience this problem, you can exclude the method from test or you can revise the method such that it uses cross-platform approaches.

To generate JUnit tests for your project:

1. If you don't already have the test generation client, follow the instructions in [Installing and Configuring the AgitarOne Client Software](#) in *AgitarOne Installation and Configuration Guide* and restart your IDE.

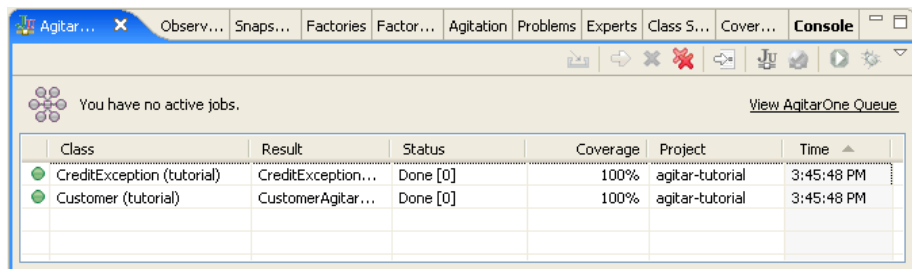


2. In the **Package Explorer** view, select one or more classes or packages, then click the **Generate Tests** toolbar button.

If your project doesn't already have a source folder named **agitar/test**, the test generation client creates one for you. If you have exported Ant build files, you will need to re-export them. Generating tests also adds the Agitar libraries defined by the classpath variables **AGITAR\_TEST\_LIB** and **AGITAR MOCK\_OBJECTS** or **AGITAR MOCK\_OBJECTS5** to your project.

**TIP:** You can change the name of the folder that contains the generated tests. For instructions, see the next section, [“Specifying the Location of Generated Tests.”](#)

3. The **AgitarOne Server** view opens and shows you the progress of test generation:



To see all the jobs on the server, click **View AgitarOne Queue**.

### HOW THE AGITARONE SERVER QUEUE WORKS

Each developer using the server has a dedicated queue. AgitarOne takes jobs from the queues in a round-robin fashion, so the amount of time until a test returns varies with the number of active users and the number of worker threads in the server.

When test generation is complete, the generated tests appear in the source folder specified in the previous step. From this view, you can do the following things:

- ◆ Click an entry in the **Class** column to go to the source code of the class under test.
- ◆ Click an entry in the **Result** column to go to the source code of the generated test class.
- ◆ Click the entry in the **Status** column to see the diagnostic report and the user log.

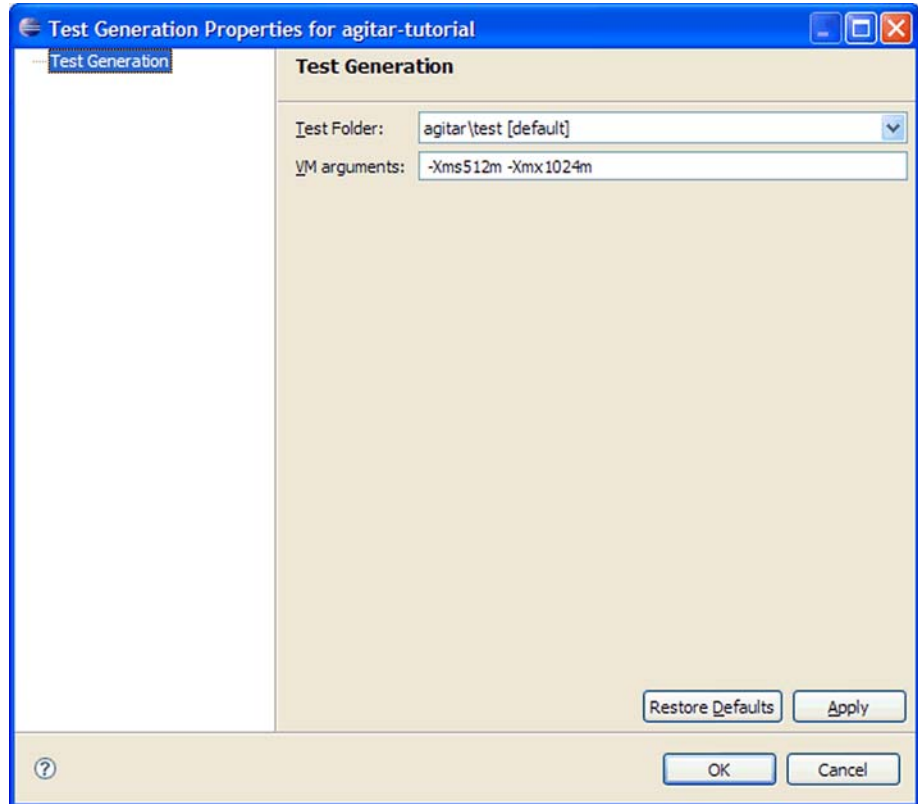
**NOTE:** The AgitarOne server does not generate tests for interfaces or for abstract classes with no concrete subclasses in the project. If you want to test abstract classes, you must either provide a concrete subclass or write a test helper that generates objects of the type you want to test. For more about test helpers information, see [“Improving Test Generation with Test Data Helpers” on page 50](#). If your interface contains an inner class for which you want a generated test, you must move the class out of the interface.

## Specifying the Location of Generated Tests

By default, the test generation client puts generated tests in the source folder **agitar\test**. You can designate any source folder as the location for generated tests received from the AgitarOne server. The source folder you specify must already be defined in your project.

To specify a folder for generated tests:

1. Select **Agitar>Test Generation Properties**:



2. From the **Test Folder** drop-down list, select a source folder.

**TIP:** If the folder you want to use is not on the list, cancel this dialog box and create a new source folder in your project with the name you want. If the folder exists, make sure it is a source folder.

3. Click **OK** to save your settings.

## Generating Missing Tests

Sometimes, AgitarOne is unable to generate a test class for one or more classes. When you examine the diagnostic report for such a class, you see messages suggesting refactoring or a setup helper as a way to help AgitarOne generate the tests that you want.



To generate tests only for those classes that do not yet have them:

1. Make the code changes necessary to help AgitarOne generate the tests.

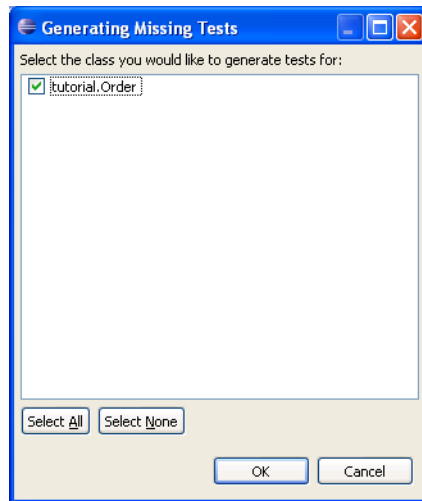
Consider the suggested solutions in your diagnostic reports when making these changes. Refactor code that depends upon external state or write a helper to set up such state before the test case runs. Save your changes.

2. Select the package or project for which you want to generate missing tests.
3. Right-click and select **Agitar>Generate Missing Tests**.



You can also use the drop-down menu on the **Generate Tests** toolbar button.

4. The **Generating Missing Tests** dialog box appears. All classes which currently lack tests are selected.



**NOTE:** AgitarOne considers only tests that it generates when looking for classes that do not have tests. It does not consider your hand-written tests.

5. In the **Generating Missing Tests** dialog box, select those classes for which you want to generate tests and click **OK**.

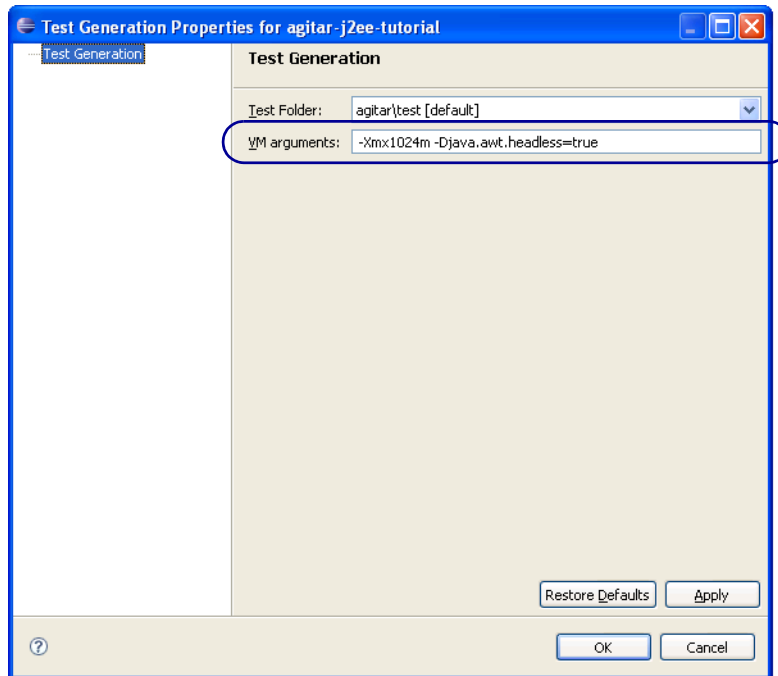
## Specifying the Location of Native Code for Test Generation Projects

If your application uses JNI to access native code for some of its operations, specify the location of the native code by setting a JVM argument for the project.

To specify the native code location for agitation and for test generation:

1. In the **Package Explorer** view, select your project.

2. Right-click and select **Agitar>Test Generation Properties**:



3. Add the native path to the **VM args** field.

Enter this argument as `-Dagitar.native.path=` and set the value relative to your workspace root. For example, if your workspace contained the `MyProject` directory, you might enter:

```
-Dagitar.native.path=MyProject/libs/native
```

4. Click **OK**.

## Test Generation and Java Security

To protect the AgitarOne server's host computer and your own computer during test runs, generated tests are restricted in what they can access. Generated tests are subject to the default permissions for agitation, documented in [“About the Default Security Settings”](#). Tests are further restricted in that they can read only the project directory, the classpath, and the **temp** directory.

The same restrictions apply regardless of how you run the tests: in Eclipse or using Ant tasks.

If you change the security settings for agitation, those changes do not affect your generated tests at runtime.

## About Tests Generated by AgitarOne

JUnit tests generated by the AgitarOne server validate the current behavior of your code. The server takes the following steps to generate tests:

- analyzes the code
- determines the number of branches that must be traversed for thorough coverage
- identifies normal and exception outcomes that can occur,
- generates test values for normal, exception cases and for boundary conditions

For each project class, AgitarOne generates a test class with as many test methods as necessary to meet the requirements discovered during the analysis of the code. Each test method tests one target method's normal or exception cases, as reflected in the name of the test method.

Generated tests follow basic JUnit conventions. Specifically:

- To name the test class, AgitarOne appends **AgitarTest** to the original class name.
- The test class extends **com.agitar.lib.junit.AgitarTestCase**, a subclass of **junit.framework.TestCase**. **AgitarTestCase** provides some helper methods to make the generated tests easier to read.
- Each generated test class contains a **getTargetClass()** method that returns the project class being tested.
- To name each test method, AgitarOne prepends **test** to the original method name.
- Each test method executes some of the code under test, using appropriate test values, followed by one or more **assert** statements to validate identified behavior.

[“Reviewing the Generated Tests” on page 21](#) shows some examples of generated tests.

**NOTE:** If you want to generate tests for a class that takes longer than 500ms to initialize, consider initializing the class in a **SetupHelper.setUpTestGeneration()** method. Otherwise, test generation might time out and fail. Alternatively, you can set the timeout property higher in the [Agitation Properties: Timeouts](#) dialog box. However, changing this setting can result in slower test generation.

## Agitation Artifacts and Generated Tests

If you also agitate your code, some configuration set during agitation is respected by test generation. Test generation uses the following agitation artifacts: factories, concrete mocks, setup methods, init methods, and excluded or unexcluded methods.

Test generation ignores the following agitation artifacts: security settings, VM arguments (use the [Test Generation Properties Dialog Box](#) instead), assertions, observations, and partitions.

---

## Running Generated Tests

Agitar provides an enhanced version of the Eclipse JUnit runner to run the generated JUnit tests. In addition, AgitarOne includes an Ant task that you can use in automated builds to run generated JUnit tests; for more information, see [“The Super-Runner Ant Task” on page 18](#).

To run JUnit tests from your Eclipse client using the Agitar test runner:

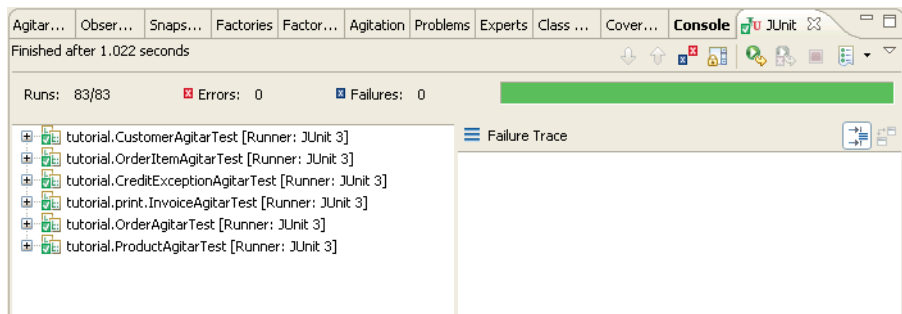
1. Select one or more test classes (or a package or folder) in the **Package Explorer** view or the **Outline** view.
2. Right-click and select **Run As>Agitar JUnit Test**.

If this option isn’t available, make sure that the folder containing the test classes is included in the project’s source path.

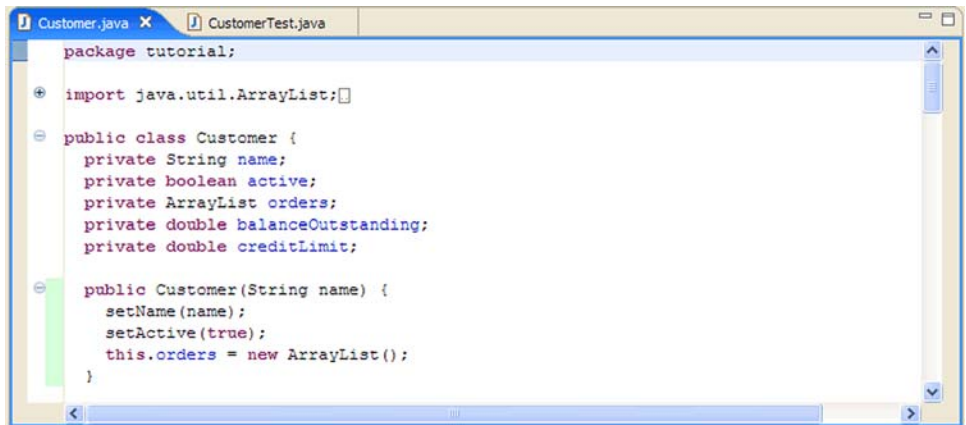


**TIP:** When you have just generated tests, you can also select one or more tests in the **AgitarOne Server** view and click the **Run Selected Tests** toolbar button.

If the **JUnit** view is not open, then it will open and show you the status of the running tests:



When you run JUnit tests using the Agitar test runner, you can see coverage statistics in the Java editor. Green indicates covered lines; red indicates lines that were missed.



The **Coverage Summary** view shows even more coverage information:

Name	Coverage (%)
Product	91
Product(String, String, double)	100
getCode()	100
getName()	100
getPrice()	100
setName(String)	100
setPrice(double)	100
toString()	100
validateCode(String)	71
validateName(String)	100
validatePrice(double)	67

To run your JUnit tests on the AgitarOne server and generate a dashboard report on the results, see [How to Use a Developer Dashboard Report](#).

---

## Rerunning Tests



To rerun any JUnit test you have already run, select the test and click the **Rerun Last Test** toolbar button in the JUnit view.

---

## Understanding Code Coverage Markers

When you run JUnit tests using the Agitar test runner, it automatically adds coverage information to the source code display. Green bars indicate code that was covered by one or more test methods and red bars indicate code that was not.

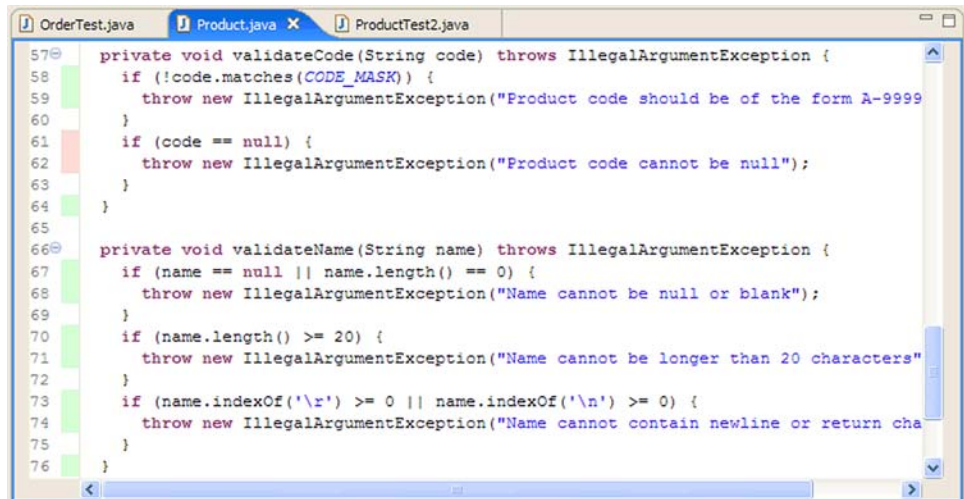
If you pause the mouse pointer over the coverage indicator, you can see a tooltip with more details about coverage for that line of code.

If you ran a group of test classes together, the coverage statistics include combined coverage from all of those tests. If you ran a series of test classes individually, the coverage data shows only information about the last test you ran.

You can change how coverage data shows in the Java editor; for instructions, see [“Setting Coverage Display Options.”](#)

## Setting Coverage Display Options

By default, AgitarOne shows coverage information in the Java editor in the left column. Green indicates code that has been exercised, and red indicates code that was not covered. For example:

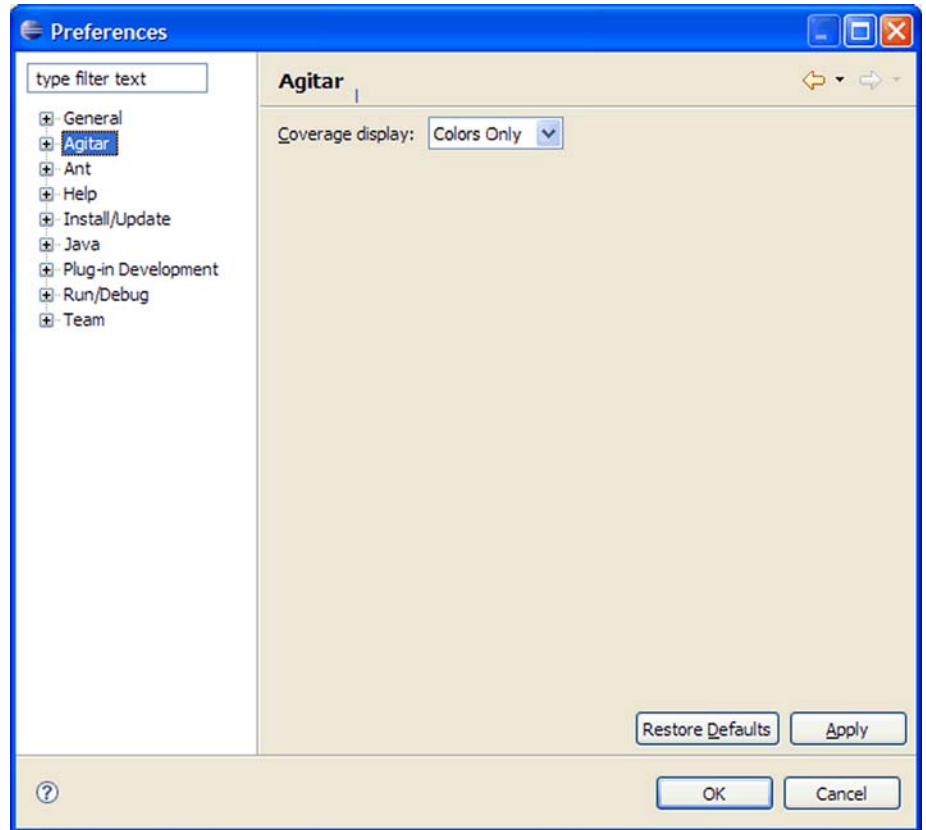


You can modify this display, to show coverage counters along with the colors. You can also change the default colors for covered and missed code.

To customize the coverage display:

1. Select **Window>Preferences**.

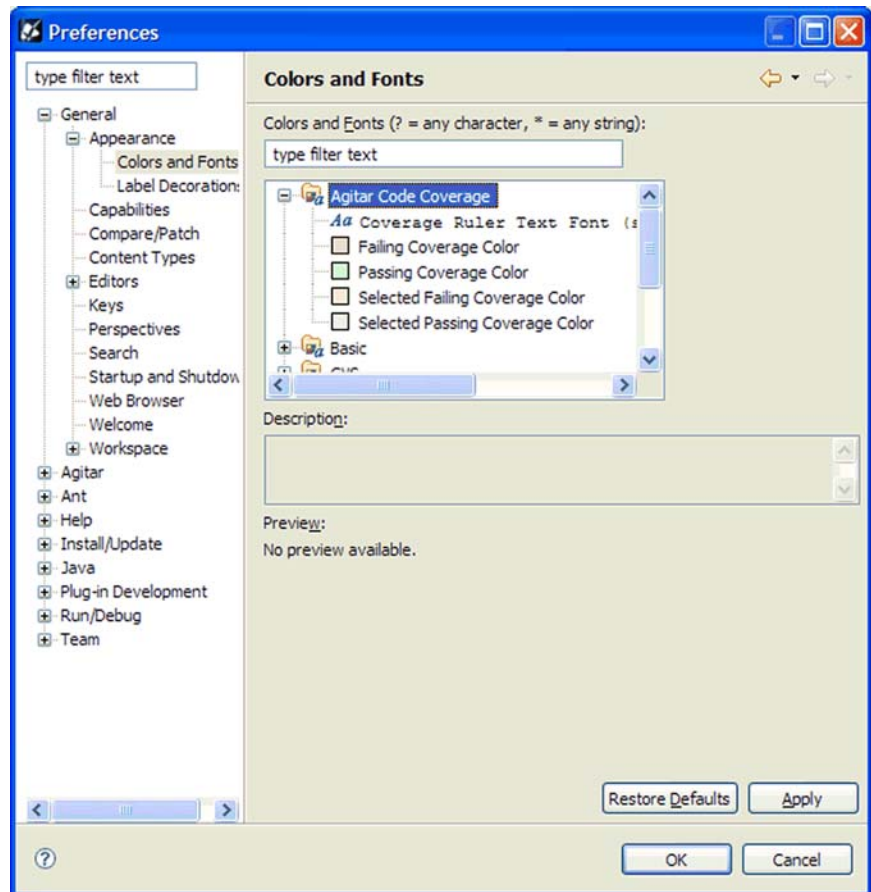
2. Select **Agitar**.



3. From the **Coverage display** drop-down list, select one of the following options:
  - ◆ **Full**—Both color coding and line execution counts.
  - ◆ **Colors Only**—Coverage status in color only, with no line execution counts. This is the default.

When you have **Colors Only** selected, you can hover your mouse over the coverage marker to see coverage percentages in a tooltip.
  - ◆ **Off**—No coverage information.
4. Click **Apply** to continue.

5. To change the coverage colors or the font used for displaying the coverage counters:
  - a. In the **Window Preferences** dialog box, expand **General**, expand **Appearance**, then select **Colors and Fonts** and expand **Agitar Code Coverage**:



- b. Select **Coverage Ruler Text Font**, then click either **Use System Font** or **Change**, and then make any changes you want to the font settings.
    - c. To change the coverage display colors, select either **Failing Coverage Color** or **Passing Coverage Color**, then click the button with the current color to select a new color.
6. Click **OK** to save your settings.

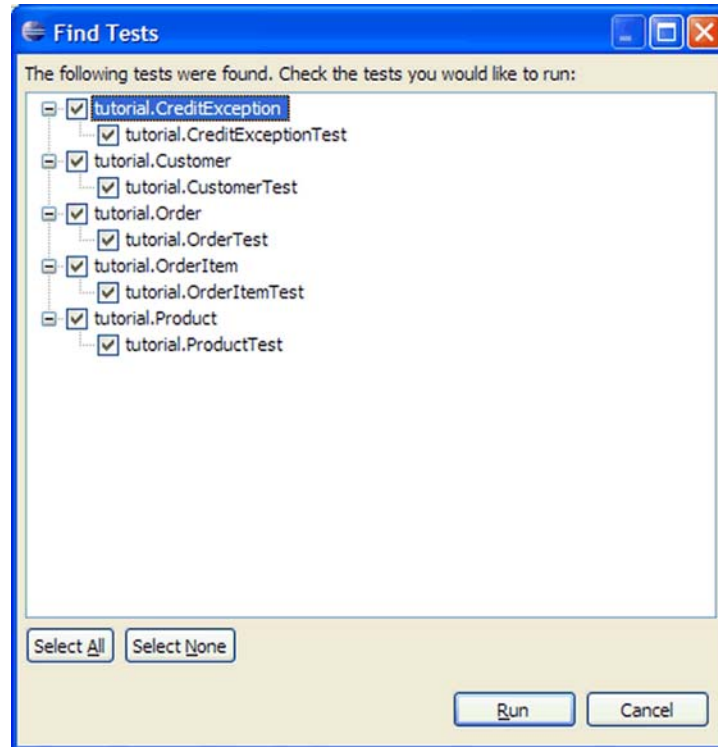


## Which Test Classes Cover Which Source Code?

To find out which tests exercise which parts of the source code:

1. In the **Package Explorer** view, select one or more project classes or packages.
2. Right-click and select **Agitar>Find Tests for Selection**.

The **Find Tests** dialog box shows you all test classes that apply to the selection:



3. To run one or more of these tests, select the check box next to the test, then click **Run**.

---

## Saving Regression Suites

When you are satisfied with a set of generated tests, check them into your version control system. From there, you can incorporate them into your regular build process so that you can identify code changes that violate the behavior described by the tests. If you want to save these tests in your project, copy them to another folder before re-generating tests. Otherwise, AgitarOne overwrites existing tests with the new ones.

As you run these tests as part of your build process, you can use information from failing tests to identify either regressions to expected behavior or changes in your code. If a test has found a true regression, you can fix the problem and rerun the tests. When tests fail because of code changes, you can regenerate the tests to account for the new behavior and incorporate the new tests into your build process instead of the failing ones.

## The Super-Runner Ant Task

AgitarOne includes Ant support for running generated tests as part of a continuous integration process. This enhanced JUnit runner instruments your code and captures coverage information that allows you to see how much of the code was covered by the test run. The **super-runner** task works just like the standard **junit** task and recognizes all of the constructs in AgitarOne-generated tests, including Mockingbird calls.

If you already use Ant with the standard **junit** task, you can keep your settings and replace the name of the task with **super-runner**. You also need to include the definition of this task in your build script. If you are new to using JUnit tests for developer testing, you can add the **super-runner** task to your build script. For more information see [Super-Runner Task](#).

## Monitoring Progress Over Time

You can also use the Management Dashboard to generate progress reports after each build. You can use these reports to see coverage, test failures, and individual developer status.

You can generate a Management Dashboard using Eclipse or Ant. For more details see [Management Dashboard and Build Integration](#).

---

## Troubleshooting Test Generation

The following topics provide information to help troubleshoot problems you might experience during test generation:

- [Diagnosing Test Failures](#)
- [Test Run Times Out With “Too Many Threads” Message](#)

## Test Run Times Out With “Too Many Threads” Message

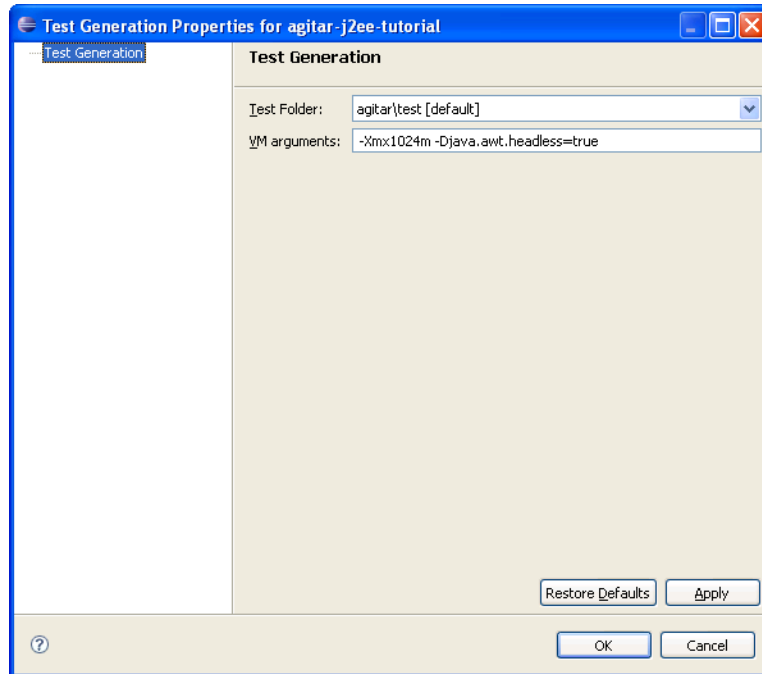
By default, AgitarOne respects the threading model of your code when generating unit tests. Occasionally, however, threads can depend on one another and result in a race condition when the tests are run.

In this situation, your JUnit test run hangs and times out. Examine the **UserLog** tab of the **Diagnostic Report** and you will see a message like the following:

Multiple threads were detected making calls into the target code. Some threads will be killed in order to prevent interference.

If you experience this problem, you can configure A1 to run your threads one at a time (for running your JUnits only):

1. In Eclipse, select your project and then select **Agitar>Test Generation Properties**.



2. In the VM arguments field of the Test Generation Properties dialog, add the following argument:  
`-DThreadController=on`
3. Re-generate your JUnit tests.

When you run the new tests, the multithreading issue should be resolved and the problem test should run normally.



# 3

## Reviewing the Generated Tests

When you get tests from the AgitarOne server, you can learn about your code by looking at what was generated. As you look through the test methods, you can see how the server analyzed your code and what tests cover each part of the code. If you see test values that don't make sense, you can use them as indicators that something in your code might not be quite right.

Because the generated tests represent exactly what the code does, they will continue to reinforce unexpected behavior unless you notice the discrepancies and address them in the code. Without human intervention, the most you can expect from the generated tests is a thorough regression suite, one that can detect changes from the current behavior.

- [Basic Structure of a JUnit Test](#)
- [A Look at the Generated Tests](#)
- [Learning About Your Code from Tests](#)
- [About Mock Objects in Generated Tests](#)
- [Evaluating Generated Tests](#)

---

### Basic Structure of a JUnit Test

JUnit tests have a standard structure, and contain the following parts:

- **setUp()**—A method that constructs test objects to be used by the test methods, places them in the proper state for testing, and initializes any global services needed during the test.
- **testMethod()**—Each test method has the following sections:
  - ◆ some setup specifically for the test call
  - ◆ a test call
  - ◆ one or more assertions to verify that the test call had the intended side effects
- **tearDown()**—A method that releases any objects created for testing and closes connections to any external resources.
- Test data that appropriately exercises particular conditions of the code, including values for expected outcomes and for boundary conditions.

JUnit tests generated by AgitarOne test generation follow this basic structure.

---

## A Look at the Generated Tests

Each test method of a generated test class focuses on one path through the code or one specific outcome. The names of the test methods give a broad indication of what they test; for example `testConstructorThrowsIllegalArgumentException()`. The test values cover a variety of conditions.

Consider the **Product** class in the tutorial project. The generated test class **ProductAgitarTest** has a number of test methods. First, several methods test the constructor, using different combinations of test values for product codes, names, and prices. For example:

```
public void testConstructor() throws Throwable {
    Product product = new Product("J-3088-67-M",
        "testProductName", -0.0010);
    assertEquals("product.getCode()", "J-3088-67-M",
        product.getCode());
    assertEquals("product.getPrice()", -0.0010,
        product.getPrice(), 1.0E-6);
    assertEquals("product.getName()", "testProductName",
        product.getName());
}
```

Because the constructor of **Product** throws **IllegalArgumentException**, the generated test class includes test methods that explicitly create the conditions that cause this exception to be thrown. Each test method corresponds to one of the ways that the exception can be thrown. For example:

```
public void testConstructorThrowsIllegalArgumentException()
    throws Throwable {
    try {
        new Product("J-3088-67-M", "", 100.0);
        fail("Expected IllegalArgumentException to be thrown");
    } catch (IllegalArgumentException ex) {
        assertEquals("ex.getMessage()", "Name cannot be null
            or blank", ex.getMessage());
        assertThrownBy(Product.class, ex);
    }
}
```

Finally, the test class contains test methods that exercise each of the getter and setter methods in the **Product** class. The next section, [“Learning About Your Code from Tests,”](#) shows examples of circumstances where generated tests can reveal unexpected behavior or other problems in your code.

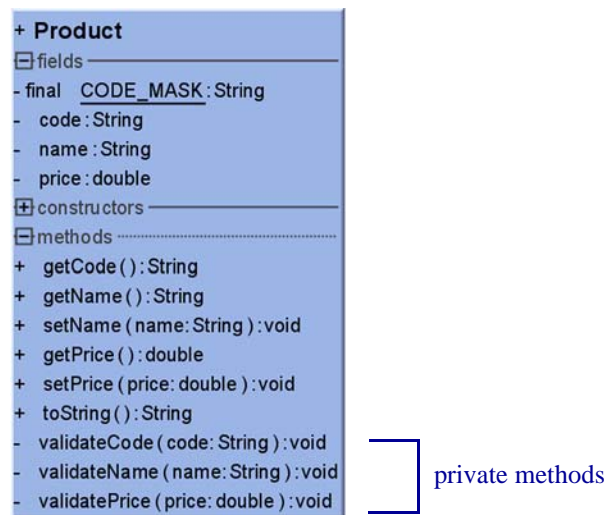
---

## Learning About Your Code from Tests

What can reviewing the generated tests tell you about your code? Because the tests describe existing behavior, you can look through the generated test methods to discover unexpected exceptions, test values that don't match your design specifications, and unintentional code behavior. In addition, you can examine the code coverage for each class under test to see what parts of the code were not exercised by any of the generated tests.

The **Product** class in the **agitar-tutorial** project contains some built-in bugs. By looking at the test methods generated for this class, you can see how the tests reveal these problems.

The following figure shows the structure of the **Product** class:



This class has three fields, with the following constraints on their values:

- **code**—Unique identifier of a product; a **String** whose format is defined by the regular expression specified by **CODE\_MASK**. Well-formed **code** values look like **A-9999-99-A**, where **A** represents an upper-case letter and **9** represents a digit.
- **name**—A non-null **String** up to 20 characters long.
- **price**—A **double** whose value can be between **0.0** and **1000.0**.

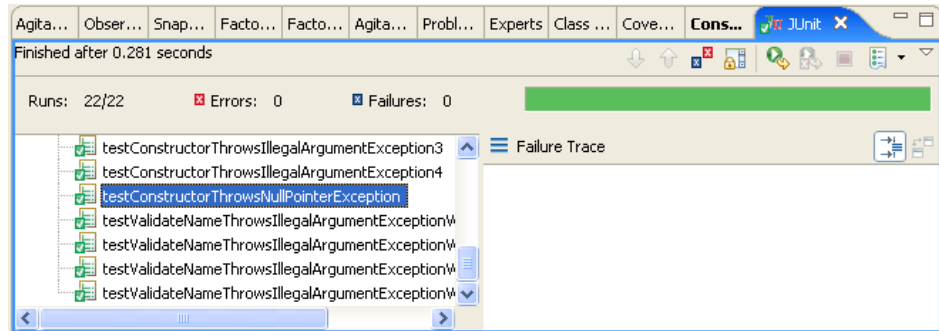
Notice that this class has private methods that validate the format of each of these fields.

The following sections show examples of what you can learn from the generated tests:

- [Names of Test Methods](#)
- [Generated Test Values](#)
- [Missing Code Coverage](#)

## Names of Test Methods

Because the name of each test method describes what that method will test, you can use the names of the generated tests to quickly scan for anomalous behavior.



In the case of the **Product** class, you can see that one of the tests for the constructor identifies a **NullPointerException** outcome. Because this is not one of the expected outcomes for the code, it is worth some investigation to see how this exception happened.

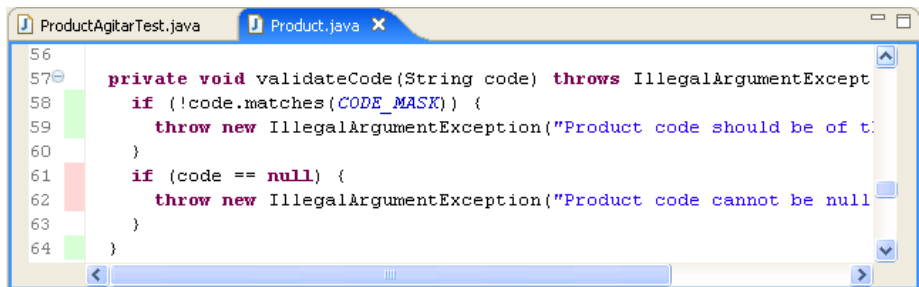
The test method looks something like this:

```
public void testConstructorThrowsNullPointerException() throws Throwable {
    try {
        new Product(null, "testProductName", -0.0010);
        // should throw: java.lang.NullPointerException;
        fail("Expected NullPointerException to be thrown");
    }
    catch(NullPointerException ex) {
        assertNull("ex.getMessage()", ex.getMessage());
        assertThrownBy(Product.class, ex);
    }
}
```

**NOTE:** Some of the generated test values might be different in your test method.



You can see from the test method that the exception happens because the product code can be **null**. Shouldn't the code test for this condition? A look at the source for the **validateCode()** method shows no coverage on the line that tests for **null** values:



```
56
57 private void validateCode(String code) throws IllegalArgumentException {
58     if (!code.matches(CODE_MASK)) {
59         throw new IllegalArgumentException("Product code should be of t
60     }
61     if (code == null) {
62         throw new IllegalArgumentException("Product code cannot be null
63     }
64 }
```

It turns out that the test for **code == null** comes after the test to see whether **code** matches the defined mask. Reversing the order of the **if** blocks fixes the problem.

When you fix the code and rerun the generated test, you can see that the test method **testConstructorThrowsNullPointerException()** now fails, as you would expect. Now that you have fixed the problem, you can regenerate the test.

To re-generate tests for a class:

1. In the **Package Explorer** view, select the class.
2. Click the **Generate Tests** toolbar button.



The AgitarOne server places the new test for your class in the **agitar/test** folder.

3. In the **Product** example, look at the generated test class to see that it no longer contains a test method called **testConstructorThrowsNullPointerException()**.

## Why Is My Test Method Named **test\*WithAggressiveMocks()**?

AgitarOne uses another naming convention when generating test methods: the label **WithAggressiveMocks**. When a test method name ends with this label, it indicates that AgitarOne was unable to get full coverage using real objects and is progressively mocking more and more of the application. In tests with this label, some part of the class under test is being mocked. The mock might include only one method, or it might include all of the class except the specific method being tested. Such a test provides good regression coverage.

However, if you want the test to use real objects, you can provide a test helper that creates class instances in the necessary state to create the tests. See [“Tuning AgitarOne Test Generation” on page 47](#) for information about writing test helpers. For more about assessing the generated tests, see [“Evaluating Generated Tests” on page 30](#).

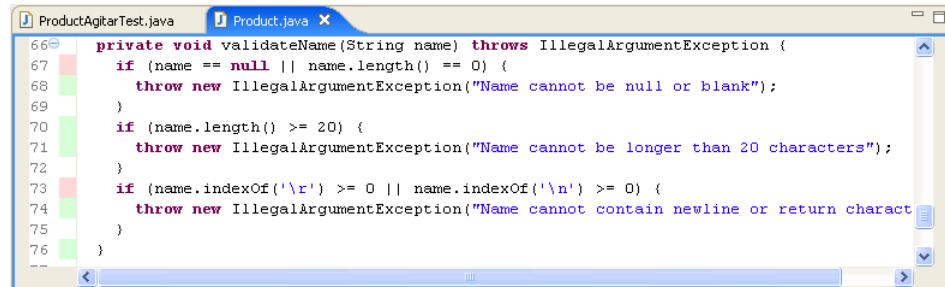
## Generated Test Values

A review of the test data in generated test values can provide useful information about both typical values and boundary conditions the AgitarOne server discovered as it analyzed your code. For clarity, generated strings that test boundary conditions include the string length being tested in the string value. One of the first tests generated for the **Product** class looks like this:

```
public void testConstructor1() throws Throwable {
    Product product = new Product("B-1234-12-B",
        "19CharactersXXXXXXX", 0.0);
    assertEquals("product.getCode()", "B-1234-12-B",
        product.getCode());
    assertEquals("product.getPrice()", 0.0,
        product.getPrice(), 1.0E-6);
    assertEquals("product.getName()",
        "19CharactersXXXXXXX", product.getName());
}
```

Part of the definition of **Product** is that the **name** field can contain up to 20 characters. However, this test method uses a string that is only 19 characters long. Why not test 20-character strings? Is there something in the code that limits the value to 19?

The answer is in the source code for the **validateName()** method:

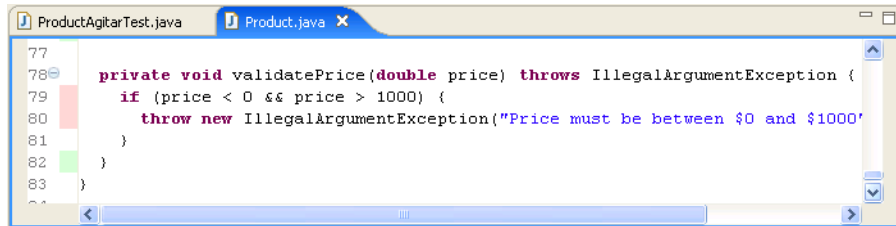


The **if** statement should only test for values greater than 20, not greater than or equal to 20. When you fix the code and rerun the tests, other methods in the test class, **testConstructorThrowsIllegalArgumentException()** and **testValidateNameThrowsIllegalArgumentException()**, catch the code change. Regenerating the test gives a new test method with **name** strings up to 20 characters long.

## Missing Code Coverage

In addition to reviewing the generated test code, you can discover possible problems in your code by examining the code coverage information in the Java editor. Blocks marked red were not exercised by any of the generated test methods. Exploring these blocks can help you determine what is happening in your code.

For example, a look at the coverage on the **Product** class after running the generated tests shows the following uncovered code:

A screenshot of an IDE window showing two tabs: 'ProductAgitarTest.java' and 'Product.java'. The 'Product.java' tab is active, displaying a Java method 'private void validatePrice(double price) throws IllegalArgumentException {'. The method contains an 'if' statement: 'if (price < 0 && price > 1000) {'. Inside the 'if' block, there is a 'throw new IllegalArgumentException("Price must be between \$0 and \$1000');'. The code is highlighted with a red background, indicating it is uncovered. The line numbers 77, 78, 79, 80, 81, 82, and 83 are visible on the left side of the editor.

Looking at the **if** statement reveals a logic error. The value of **price** can never be both less than 0 and greater than 1000. The bug here is that **&&** should be **||** instead. When you correct the error and regenerate the tests, you will have full coverage for this method.

---

## About Mock Objects in Generated Tests

When necessary, the generated tests create mock implementations of required classes or external services, to better isolate the part of the code that is being tested. The AgitarOne server uses Mockingbird, a library similar to EasyMock (<http://www.easymock.org/>) and JMock (<http://www.jmock.org/>), to mock interfaces, and abstract and concrete classes. Mockingbird makes it possible to test complex code that these other mock libraries cannot test. Mockingbird enhances mocking ability (by bytecode instrumentation of all non-system classes) to allow mocking of final classes and final methods, plus the following:

- *Auto-catching of exceptions in class initialization*—by default, all exceptions from **clinit** (except those in the system class) are caught and **clinit** returns normally. Initialization of the class under test can throw exceptions to let you know that the class is bad. Mockingbird has a method, **clinitShouldThrow()**, that allows a **clinit** to throw exceptions.
- *Mocking of static methods*—record/playback can be done on static methods of all user classes, but only for static method calls of system classes called from user classes.
- *Mocking of objects created inside method of user code*—this is done by “dubbing,” using the following steps:
  - ◆ Get a new object of the type to be mocked and record the methods that needed to be mocked.
  - ◆ During playback, copy the recording to the object inside the method under test immediately after the object is constructed.

## Reading Generated Tests That Use Mockingbird

When a test fails because of code changes and it is not immediately clear from the assertion message why it failed, you might need to read the test method to understand what it does. While a large number of AgitarOne tests are easy to read and understand, several of our generated tests use the Mockingbird API. Reading a generated JUnit test that uses Mockingbird code can be challenging unless you understand what some common Mockingbird APIs are doing.

There are just 5 API methods that you need to know to better understand what Mockingbird is used for:

- **Mockingbird.enterRecordingMode()** puts the API in recording mode. Once in this mode, the API can be used to set return values or exception expectations for specific methods.
- **Mockingbird.enterTestMode()** marks the end of the recording mode and begins playing back any recordings. Generally, this method is immediately followed by a call to method being tested (the test call), followed by assertions to validate that the test call behaved as expected.
- **Mockingbird.getProxyObject()** creates a mock instance of the class passed as the parameter to the method. For example:  

```
List list = Mockingbird.getProxyObject(List.class)
```
- **Mockingbird.setReturnValue()** sets a method to return a specific value. For example:  

```
Mockingbird.setReturnValue(connection.prepareStatement(  
    "testString"), preparedStatement)
```

  
causes a call to **prepareStatement()** with the parameter **testString** return the **preparedStatement** object.
- **Mockingbird.setException()** sets a method to throw an exception. For example:  

```
Mockingbird.setException(statement.executeUpdate(sqlStatement),  
    new SQLException("cannot update"))
```

  
causes the **executeUpdate()** method on **statement** to throw a **SQLException** with the message “cannot update”.

For example, consider the following method from the J2EE example project’s **DatabaseHelper** class:

```
void insert(String sqlStatement, Object[] parameters) throws SQLException {  
    PreparedStatement statement =  
connection.prepareStatement(sqlStatement);  
    for (int i = 0; i < parameters.length; i++) {  
        statement.setObject(i + 1, parameters[i]);  
    }  
    statement.executeUpdate(sqlStatement);  
}
```

The **insert()** method accepts a SQL statement and an array of parameters, and updates the database accordingly. When you generate tests for the **DatabaseHelper**, you might get a test method like the following:

```
public void testInsert() throws Throwable {
    // Test Preparation
    Connection connection = (Connection)
        Mockingbird.getProxyObject(Connection.class);
    PreparedStatement preparedStatement = (PreparedStatement)
        Mockingbird.getProxyObject(PreparedStatement.class);
    DatabaseHelper databaseHelper = new DatabaseHelper(connection);
    Object[] objects = new Object[0];
    Mockingbird.enterRecordingMode();
    Mockingbird.setReturnValue(connection.prepareStatement("testString"),
        preparedStatement);
    Mockingbird.setReturnValue(connection.prepareStatement("testString"),
        preparedStatement);
    Mockingbird.enterTestMode(DatabaseHelper.class);

    // Test call
    databaseHelper.insert("testString", objects);

    // Assertions
    assertInvoked(preparedStatement, "executeUpdate", new Object[]
        {"testString"});
    assertInvoked(connection, "prepareStatement", new Object[]
        {"testString"});
}
```

In this test, the preparation section includes the following Mockingbird operations:

- **getProxyObject(connection.class)** creates a mock **Connection** object to be used in creating the **DatabaseHelper** instance.  
During unit testing, you do not generally want to connect and write to the database. This proxy object helps mock those operations for testing purposes.
- **getProxyObject(PreparedStatement.class)** creates a mock **PreparedStatement** object to be used in mocking the **Connection.prepareStatement()** call.

The test method then creates a **DatabaseHelper** instance using the mock **Connection** and prepares an empty array which is needed as a parameter for the **insert()** method call.

Next, **testInsert()** uses Mockingbird's recording mode to set some specific return values:

- **enterRecordingMode()** marks the portion of the test code where return values are specified for the test.
- **setReturnValue(connection.prepareStatement("testString"), preparedStatement)** sets up the test so the mock **PreparedStatement** is returned when **insert()** calls **connection.prepareStatement()**.

- **setReturnValue(connection.prepareStatement("testString"), preparedStatement)** sets up the test so zero (0) is returned when **insert()** calls **preparedStatement.executeUpdate()**. Depending on the underlying implementation, this might be an interesting test. The **insert()** method code does not check for the return value. If zero indicates an error state for an update, the test could indicate a problem with the code.

The **enterTestMode()** call marks the end of the preparation and the beginning of the actual test. The test calls **DatabaseHelper.insert()** with the empty array created earlier and a fake SQL statement **testString**.

Finally, the test makes the following assertions about the test call results:

- **assertInvoked(preparedStatement, "executeUpdate", new Object[] {"testString"})** asserts that **preparedStatement.executeUpdate()** is called during the test with our fake SQL statement **testString**, which was passed to the **DatabaseHelper**'s constructor.
- **assertInvoked(connection, "prepareStatement", new Object[] {"testString"})** asserts that **connection.prepareStatement()** is called during the test, also using our fake SQL statement.

You will also notice that the generated tests typically have many assertions, including standard JUnit assertions like **assertEquals()** and **assertSame()**, as well as any **AssertionHelper** assertions you have provided. The **assertInvoked()** assertions help to guard the behavior of calls made on mock object instances.

In some tests you will notice variants of the **Mockingbird.setReturnValue()** and **setException()** calls using a **String** to specify the method name. This is generally used when the method is private or not accessible directly. For example:

```
Mockingbird.setReturnValue(false, log, "debug(Ljava/lang/
Object;)V", null, 1)
```

causes the **log.debug()** call to return 1.

For more details about the Mockingbird APIs, refer to the javadoc reference documentation in the product online help.

---

## Evaluating Generated Tests

AgitarOne attempts to cover all of your code when generating tests. Depending on how testable your classes are, AgitarOne uses no, a few, or many mock objects to enable testing. The primary goal of test generation is to get as much code coverage as possible, relying on the developer to understand whether tests using many mocked objects are sufficient for testing purposes.

This topic guides you in evaluating the generated tests and determining whether you need to provide test helpers to improve them. It contains the following sections:

- [What Gets Mocked in a Generated Test](#)

- [Advantages of Testing With Real Objects](#)

## What Gets Mocked in a Generated Test

The following situations are difficult or impossible for AgitarOne to test without mock objects:

- **Extreme complexity**  
The more conditions that have to be true to reach certain parts of the code, the harder it is to get all those conditions right.
- **Extreme indirection**  
The farther from the class under test a piece of data originates, the harder it is for AgitarOne to create that data in the correct state for testing.
- **Unavailable classes**  
Classes that won't load or cannot be constructed block test generation. If your class under test depends on a class that has a static initializer that fails, or if it depends on static state, AgitarOne won't be able to create a real instance for testing.

AgitarOne considers the “unit” in unit testing to be the Java class. When generating tests for one of your classes, AgitarOne tries to respect the class boundary and to use real instances of the class under test, and of the classes that class uses, whenever possible.

When analyzing your code for test generation, AgitarOne tries first to create real instances of the objects your class uses. If that is not possible in all cases, AgitarOne tries to use mocks only for the objects with least proximity to your class under test. If your class uses an object indirectly, AgitarOne will mock that object if it is needed for test generation. If your class directly calls on another class, AgitarOne will try not to mock that class, but will mock it if AgitarOne cannot create a valid instance for test generation.

In the most extreme cases, AgitarOne will mock other instances of the class under test and even the instance under test though never the method actually being tested. As the expert on your code, you must determine when a given test is fine using mock objects and when you need to improve the generated tests. In some cases, writing a short test helper will provide valid objects for many generated tests. In other cases, you might decide that a hand-crafted JUnit test will provide a better test.

## Items That Are Never Mocked

AgitarOne never mocks the following in your code:

- impossible conditions (for example: `a == 3 && a == 5`)
- native methods
- reflection
- static final constants (unless they use a method call or are set in a static initializer)

## Advantages of Testing With Real Objects

Although tests using mocked objects are valid characterization tests for the item being tested, they do not always test the interactions between classes in your project as desired. When assessing a test that uses aggressive mocks, consider the following advantages of testing with real objects:

- Tests that use real objects will better test the interactions between classes.
- Tests that use a real instance of the class under test will more fully and realistically test class code other than the method being tested (which is never mocked).
- Tests that use real instances of other classes in the project will more fully and realistically test the interactions between classes in the project. They are more likely to fail when the behavior of objects on which they depend has changed.
- Tests that use mock objects aggressively are more sensitive to changes in your project implementation. They are more likely to fail because of trivial changes in your code.

For information about how to read a test that uses the Mockingbird APIs, see [“Reading Generated Tests That Use Mockingbird” on page 28](#).



# 4

## Handling Test Failures

Tests that fail are good tests; they tell you something important about your code. The point, however, is to understand and fix the test failures. This chapter provides information to help you understand test failures in your project.

This chapter contains the following topics:

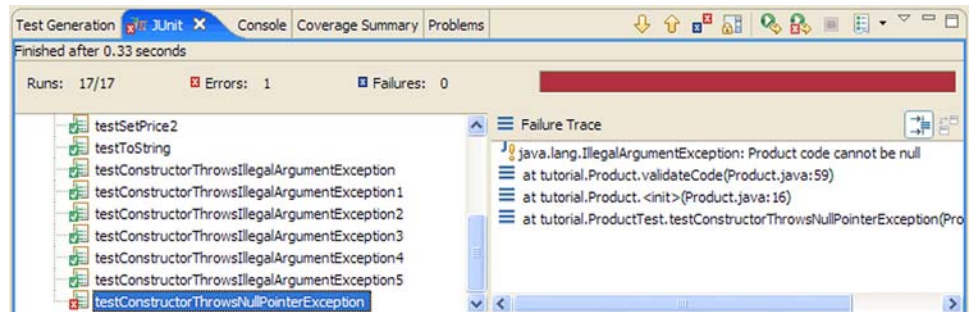
- [Diagnosing Test Failures](#)
- [Understanding TestException Messages](#)
- [Handling Test Failures From a Dashboard Report](#)
- [Re-Generating Failing Tests](#)
- [Deleting Invalid Failing Tests](#)

---

### Diagnosing Test Failures

If any tests fail, the progress bar will be red. Each failed test has a blue X icon. To see a list of only your failed tests, select **Show Failures Only** from the view menu.

When you select a failed test, the **Failure Trace** shows what happened to lead up to the error. To track down how the error happened, you can double-click any line in the trace to go to that location in the code.



In a Management Dashboard report, you can see a stacktrace for an error by clicking the **Result** link in the **Failures Details** report.

Failing tests are good tests; they indicate regressions and other problems in your code. However, if you have test failures immediately after generating tests, the problem is most likely with the test and not with your code. If your newly generated tests fail, consider possible [platform differences](#) as the root cause of the failure.

If your newly generated tests fail on their initial run, you should either delete or re-generate them. See [“Re-Generating Failing Tests” on page 43](#) or [“Deleting Invalid Failing Tests” on page 44](#) for instructions.

When you choose to run non-Agitar test cases on the AgitarOne server, the following can also cause initial test failures when the tests pass in your local environment:

- [dependency on run order](#)
- [unexpected assert statement results](#)

If your test fails with a **TestException**, it means that AgitarOne encountered something unexpected when running a test that uses Mockingbird to achieve test coverage. For more about such mock objects, see [“About Mock Objects in Generated Tests” on page 27](#). For more about **TestException** results, see [“Understanding TestException Messages” on page 35](#).

## Platform Dependencies and Cross-Platform Differences

If your AgitarOne client and AgitarOne server are on different platforms, you might sometimes get a test back from the server that fails when run. This is because tests that access platform-specific items and run reliably on the server might not get the same results when run on the client. (For example Windows and Linux systems use different line separators.) If you experience this problem, you can exclude your method from test or you can revise the method so that it uses cross-platform approaches.

This issue can arise in both generated and hand-written tests. The AgitarOne server attempts to generate platform-independent tests, but on some occasions a platform-specific item will slip through. If you then modify the generated test so that it passes on your client, that test will fail again if you run it on the server to generate a developer dashboard.

To diagnose such failures, look for the following kinds of things in your stacktrace:

- assertion failures when the assertion contains platform-specific values  
You will see a message similar to: Expected <.../> but was <...\>.
- hard-coded system paths in the tests
- platform-specific conditions that control different paths through your code

## Tests That Depend On Run Order

If your hand-written JUnit tests depend on running in a specific order, you can see unexpected test failures when you run your tests on the server as part of a developer dashboard request. Tests generated by the AgitarOne server do not depend on run order.

## Running Your Own Tests with the Super-Runner

The **super-runner** always runs with the **-ea** option to enable assert statements if you have them in your code. If you run your own JUnit classes with the Agitar test runner, be sure that they can pass with this option enabled.

---

## Understanding TestException Messages

When examining the stacktrace for a failure, pay particular attention to the exception or error message. You might need to drill down in the stack and look for the “caused by” entry. If the cause is a **TestException**, it means that a test with mock objects behaved unexpectedly, most likely because of a change in the class being tested.

The **TestException** messages are designed to help you diagnose the problem when a test that uses the Mockingbird API fails. Your message will fall into one of the following categories:

- [Unexpected Method Called](#)
- [Recorded Value Not Used](#)

For more information about Mockingbird and how to understand generated test code that uses this API, see [“Reading Generated Tests That Use Mockingbird” on page 28](#).

## Unexpected Method Called

When Mockingbird finds a method call in the method under test for which it has no return value in the mock object, it constructs the **TestException** with an “unexpected method called” message. You will see the following message:

TestException Message	Likely Cause
Unexpected method <i>yourMethod</i> called on mock object	The method under test calls <i>yourMethod</i> now, but did not when the test was generated. Either the method call has been added or its parameter list has changed.

This message might indicate a bug in your code and you should investigate. However, if you have recently changed the method being tested, it is possible that the **TestException** indicates an intentional change. If this is the case, you should consider deleting or re-generating the test.

## Recorded Value Not Used

Mockingbird throws a **TestException** with a “value not used” message when it reaches the end of a test and has not consumed all the recordings for methods that return **void**. You might see the following messages:

TestException Message	Likely Cause
setNormalReturnForVoid recorded for <i>yourMethod</i> is not used	The method under test called <i>yourMethod</i> when the test was generated, but it no longer does or it calls the method fewer times.
setException recorded for <i>yourMethod</i> is not used	
setReturnValue recorded for <i>yourMethod</i> is not used	

Any of these messages might indicate a bug and you should investigate. However, if you have recently changed the method being tested by removing a method call from your class or by changing parameter values, it is possible that the **TestException** indicates an intentional change. If this is the case, you should consider deleting or re-generating the test.

## When You Will See “Value Not Used” Messages

A **TestException** with a “value not used” message is only thrown if the unused recording in the mock object is for a **void** method. Recordings for methods that return a value do not trigger the exception.

This is because, if the test passes with unused calls to methods that return a value, Mockingbird assumes that the logic of the method being tested either does not use the return values or that they were correct. If a method is void, we cannot make that assumption. Consider a call to **FileInputStream.close()** for example. This void method performs a critical function. You need to know if a call to this method has been removed by mistake. However, a call to the **read()** or **getFD()** methods of the same object return values that must be handled by the method being tested. A failure to get these return values, because the call has been mistakenly removed, is very likely to cause the test to fail anyway.

## TestException Example: Recorded Value Not Used

---

**To Technical Reviewers:** *New example to come from Mark.*

---

To see how you can analyze a failed test to determine the meaning of the underlying **TestException** message, consider the following simple class, **TargetClass**:

```
public class TargetClass {
    public boolean readData(OtherClass otherClass) {
        if(otherClass.update("abc") >4 )
            return true;
        else
            return false;
    }
}
```

The **readData()** method calls **OtherClass.update()**, which looks like this:

```
public class OtherClass {
    int x;

    OtherClass() {
        x = 5;
    }

    public int update(String str) {
        return str.length() + x;
    }
}
```

AgitarOne generates the following test for **TargetClass**:

```
public void testReadData() throws Throwable {
    TargetClass targetClass = new TargetClass();
    OtherClass otherClass = (OtherClass)
        Mockingbird.getProxyObject(OtherClass.class);
    Mockingbird.enterRecordingMode();
```

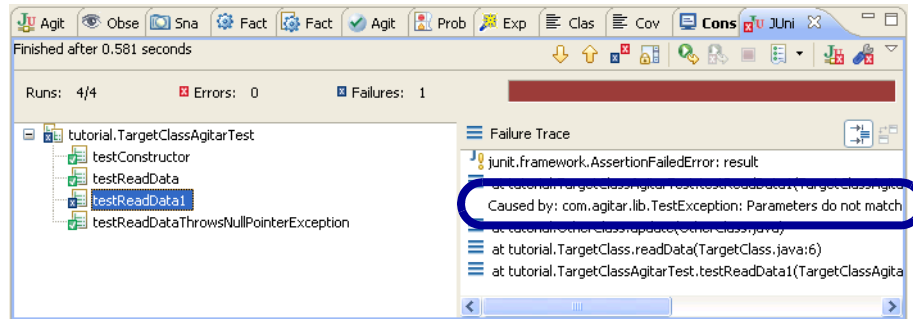
```

Mockingbird.setReturnValue(otherClass.update("abc"), 1);
Mockingbird.enterTestMode(TargetClass.class);
boolean result = targetClass.readData(otherClass);
assertFalse("result", result);
}

```

Notice that the generated test mocks the **OtherClass** instance (the **getProxyObject()** call) and sets a return value “abc” for the **OtherClass.update()** call. What happens when you change the parameter value of that method call in **TargetClass**?

Adding “de” to the parameter of the **update()** call, so that it passes “abcde,” results in a test failure:



The **TestReadData()** method failed. Notice the details in the **Failure Trace** on the right. The message at the top indicates an assertion failure and the next line identifies the test method that failed. You need to read to the third line to see:

```

Caused by: com.agitar.lib.TestException: Parameters do not match
Mockingbird recording of method:
tutorial.OtherClass.update(java.lang.String)

```

The remaining lines in the trace indicate the exact method call in the **OtherClass** that failed.

## Handling Test Failures From a Dashboard Report

If your team uses the Management Dashboard to track project status, you sometimes will receive reports with test failures. The **Summary** tab shows the number of failures, and if you follow the link in the **Current** column, you can see the details of those failures:

Failures Details				
Test Failures (19 in 7 groups )				
Failure Message		Test Method	Status	Target Classes
10 exceptions originating from the same location: Order.java:22	<a href="#">[Show Stacktrace]</a>	tutorial. OrderAgitarTest. testCreateForTestingThrowsIllegalArgumentExpection()	Failed	--
4 exceptions originating from the same location: Order.java:32	<a href="#">[Show Stacktrace]</a>	tutorial. web. CannedDataAccessAgitarTest. testSaveOrder()	Failed	CannedDataAccess
java.lang.NullPointerException	<a href="#">[Show Stacktrace]</a>	tutorial. OrderAgitarTest. testAddItemThrowsIllegalArgumentExpection()	Failed	Order
java.lang.NullPointerException	<a href="#">[Show Stacktrace]</a>	tutorial. OrderAgitarTest. testConstructor()	Failed	Order
java.lang.NullPointerException	<a href="#">[Show Stacktrace]</a>	tutorial. OrderAgitarTest. testGetItemsArray1()	Failed	Order
java.lang.NullPointerException	<a href="#">[Show Stacktrace]</a>	tutorial. OrderAgitarTest. testToString()	Failed	Order
junit.framework.AssertionFailedError : result	<a href="#">[Show Stacktrace]</a>	tutorial. web. ShowCartActionAgitarTest. testRequiresLogin()	Failed	ShowCartAction

Depending on the size of your project, you might have to scroll down to see the **Failures Details** report. The **Failures Details** report groups the test failures in groups according to their likely root cause.

When more than one test fails because of a single underlying change, AgitarOne analyzes the failure stack traces and applies heuristics to group related failures together. The largest group of failures appears at the top of the report. You can use this arrangement to help you focus your efforts when you are working to resolve the test failures in your project.

To resolve these test failures, you will need to work with them in Eclipse. Once you have the failures in the IDE, you can run the failing tests, re-generate them, or delete them depending upon your analysis. You select and import the test failures differently depending upon the type of dashboard report you are using:

- [“Importing Test Failures from a Dashboard” on page 40](#) explains how to download the results from the AgitarOne server.
- [“Importing Test Failures from a Developer Dashboard” on page 42](#) explains how to download developer dashboard reports that are in your **AgitarOne Server** view.

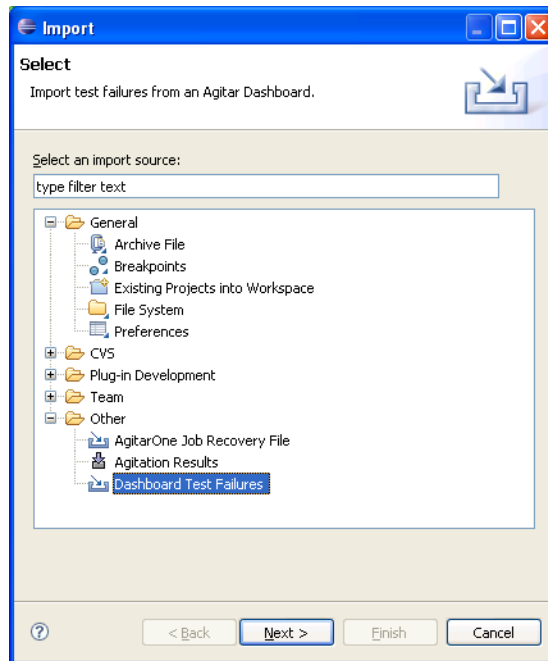
## Importing Test Failures from a Dashboard

You can use this procedure to import the test failure results from any dashboard if the dashboard job is still on the AgitarOne server. If your developer dashboard results are still in the **AgitarOne Server** view in Eclipse, you can use the steps in [“Importing Test Failures from a Developer Dashboard” on page 42](#) to get these results faster.

To import the list of failures from a dashboard into Eclipse:

1. In Eclipse select **File>Import**.

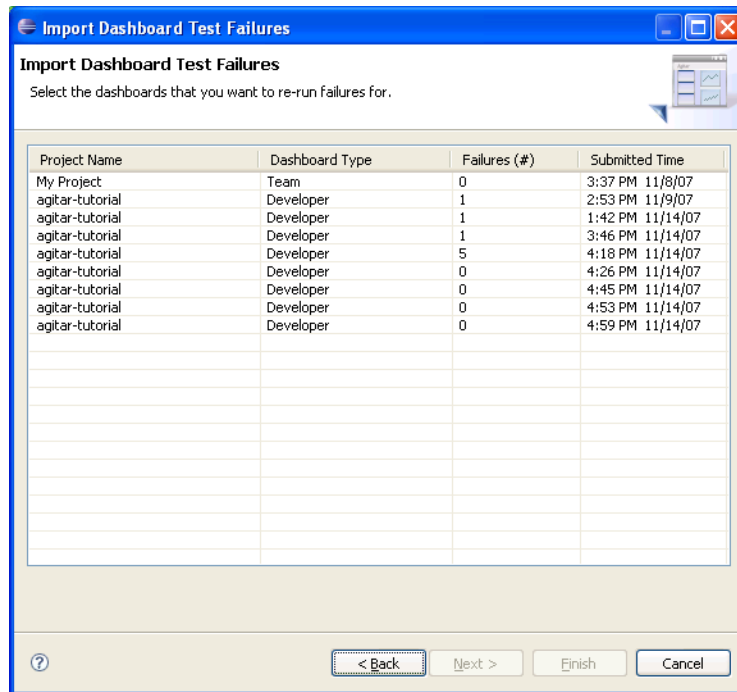
The **Import** dialog box appears.



2. Expand **Other** and select **Dashboard Test Failures**.
3. Click **Next**.

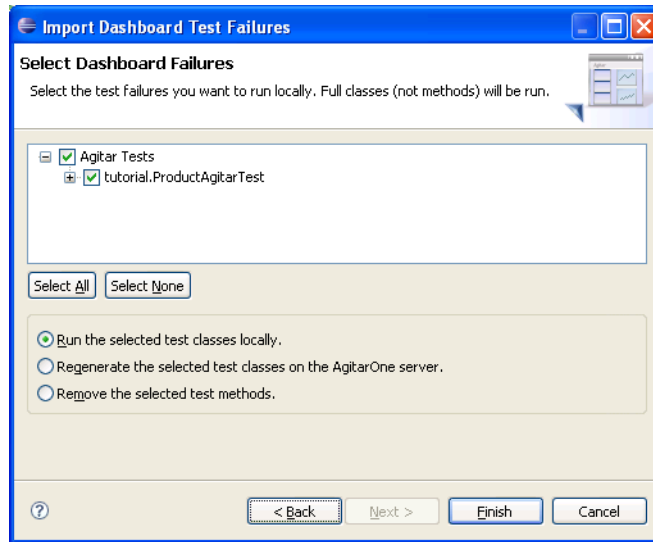


The **Import Dashboard Test Failures** dialog box appears.



4. Select the dashboard with the test failures you want to import and click **Next**.

The **Select Dashboard Failures** dialog box appears.



5. Select the tests with which you want to work.
  - ◆ If you want to run the failed tests in Eclipse, select the desired test classes.
  - ◆ If you want to re-generate the failed tests, select the desired test classes.
  - ◆ If you want to delete the failing tests, select the desired test methods, not the test classes.
6. Select the desired operation from the radio buttons.
7. Click **Finish**.

AgitarOne downloads the list of failed tests into Eclipse and performs the operation you selected. If you chose to run the tests, the JUnit view comes forward and you see the results of the failed test runs. If you chose to re-generate the tests, the **AgitarOne Server** view comes forward and you see the test-generation request being sent to the server.

## Importing Test Failures from a Developer Dashboard

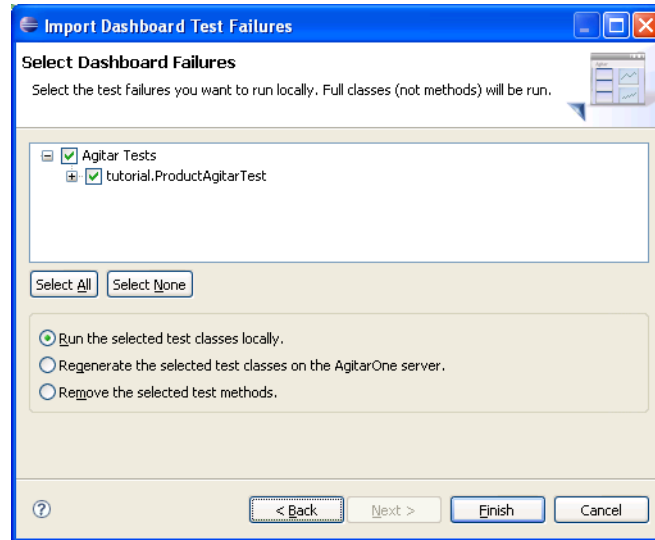
To use this shortcut procedure for the developer dashboard, you must have the developer dashboard job still visible in the **AgitarOne Server** view. If you do not, follow the instructions in [Importing Test Failures from a Dashboard](#).

To import the test failures from a developer dashboard:

1. In Eclipse, in the **AgitarOne Server** view, select the developer dashboard with the failures that you want to download.

2. In the context menu for the **AgitarOne Server** view, select **Import Dashboard Test Failures**.

The **Select Dashboard Failures** dialog box appears.



3. Select the tests with which you want to work.
  - ◆ If you want to run the failed tests in Eclipse, select the desired test classes.
  - ◆ If you want to re-generate the failed tests, select the desired test classes.
  - ◆ If you want to delete the failing tests, select the desired test methods, not the test classes.
4. Select the desired operation from the radio buttons.
5. Click **Finish**.

AgitarOne downloads the list of failed tests into Eclipse and performs the operation you selected. If you chose to run the tests, the JUnit view comes forward and you see the results of the failed test runs. If you chose to re-generate the tests, the **AgitarOne Server** view comes forward and you see the test-generation request being sent to the server.

---

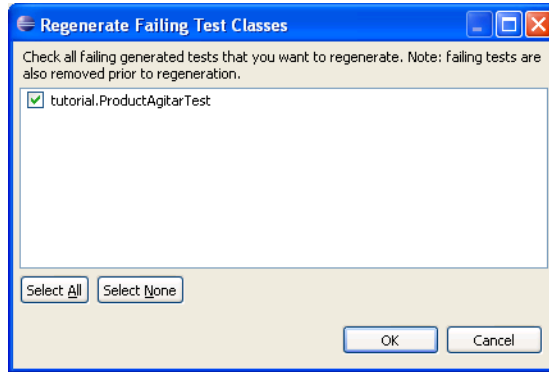
## Re-Generating Failing Tests

If you have failing tests that you need to re-generate, use the following steps:



1. In the JUnit view, click the **Regenerate Failing Test Classes** toolbar button.

The **Regenerate Failing Test Classes** dialog box appears, with all failing tests selected.



2. In the **Regenerate Failing Test Classes** dialog box, select the test classes that you want to re-generate.
3. Click **OK**.

The **AgitarOne Server** view comes forward and displays messages about the progress of your request on the server.

---

## Deleting Invalid Failing Tests

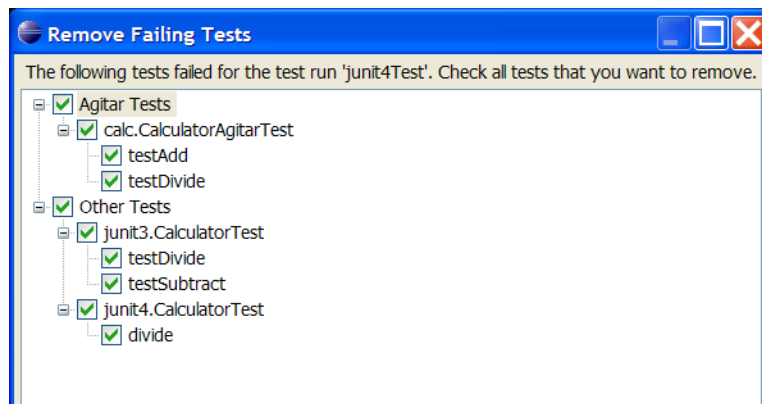
You can use these steps to remove tests that fail with compilation errors as well as those with assertion failures. You must run the tests first.

If you are certain that a generated test is invalid, you can delete it easily from the JUnit view:



1. In the JUnit view, click the **Remove Failing Tests** toolbar button.

The **Remove Failing Tests** dialog box appears, with all failing tests selected.



2. In the **Remove Failing Tests** dialog box, select the test methods that you want to delete.  
By default only failing tests which are generated by AgitarOne are selected.

**TIP:** If you delete any test in error, you can restore them by using Eclipse's **Restore from Local History** feature.

3. Click **OK**.

The methods are deleted from your generated test classes.

**NOTE:** If a test class has only failing test methods and you delete them, the class is also deleted.



# 5

## Tuning AgitarOne Test Generation

As you review tests produced by AgitarOne test generation, you might notice places where the tests would be more effective with more focused assertions or more targeted test data. You can use your understanding of the application to improve the generated tests by providing helper classes (also called test helpers) that codify your domain knowledge. Human intervention helps to produce better tests.

This chapter contains the following sections:

- [Getting Suggestions on How to Improve Tests](#)
- [Understanding Test Helpers](#)
- [Improving Test Generation with Test Data Helpers](#)
- [Improving Assertions in Generated Tests](#)
- [Specifying Complex Setup with Setup Helpers](#)

---

### Getting Suggestions on How to Improve Tests

Whenever you agitate or generate JUnit tests for a class, AgitarOne delivers a diagnostic report as part of the results downloaded to Eclipse. This report provides a list of the most important problems AgitarOne had in analyzing your class, along with suggestions to solve these problems. To see the report, click the **Status** column of the desired class in the **AgitarOne Server** view.

When you agitate or generate tests for many classes, you receive a separate diagnostic report for each class. Analyzing each report and deciding which test helper to write first can be cumbersome. Additionally, a single helper might be crucial for many classes in your project. To see the most important improvement tasks across your set of classes, you can request a summary report about any classes that have results in the **AgitarOne Server** view.

The summary report collects the 20 most important and most frequent issues in the set of classes you select and presents them in priority order. You can use the report to identify the most important things you can do, such as write a particular test helper or set up a method interceptor, to improve the results for your project.

To get a summary diagnostic report:

1. Agitate or generate tests for the desired classes as you normally would.

2. In the **AgitarOne Server** view, select the classes for which you want a summary.  
To get a summary report for all of the classes in the view, do not select any.



3. Click the **Summarize Reports** toolbar button.  
The summary report opens in your editor.



---

## Understanding Test Helpers

AgitarOne does its best to analyze your code and generate effective tests. In some cases, it is difficult to construct complex objects or to place objects in specific states required to follow individual paths through the code.

Automatic mocks are AgitarOne's way of reaching code that it cannot test directly. AgitarOne uses the **Mockingbird** API to mock as much as necessary to reach the majority of your code, including mocking the class under test in extreme situations. This might not be what you want, because overly mocked tests are not always as meaningful as tests that use real project code.

While helpful to detect code changes in the class under test, heavily mocked tests won't catch regressions caused by changes in other classes. It is also easy to break these tests by refactoring your code. When you notice this kind of mock object use in your generated tests, check the class's diagnostic report for suggestions about the kinds of test helpers that will improve your tests.



To assist AgitarOne to use real objects in the generated tests, you can provide helper classes that produce useful objects which contribute to better generated tests. You can also write helpers that produce more realistic test values than AgitarOne can, or to make generated tests more readable.

AgitarOne includes the following test helpers to improve your generated tests:

- **TestHelper** implementations contain helper methods for several purposes:
  - ◆ Test data helper methods can provide objects in particular states and objects that AgitarOne cannot construct without help. For more about test data helpers, see [“Improving Test Generation with Test Data Helpers” on page 50](#).
  - ◆ Assertion helper methods can tell AgitarOne about the intended behavior of your code. You can write these methods to include some of your knowledge about the application domain in the generated tests. For more about assertion helpers, see [“Improving Assertions in Generated Tests” on page 55](#).
  - ◆ Observation helper methods ask AgitarOne to observe certain fields in your classes and make an assertion about their state or behavior. For more about observation helpers, see [“Improving Assertions in Generated Tests” on page 55](#).
- **SetupHelper** implementations contain project-wide and method-specific helper options:
  - ◆ Setup sequences that set up the external environment for test generation. For more about setup helpers see [“Specifying Complex Setup with Setup Helpers” on page 60](#).
  - ◆ Method interceptors for methods on which the class being tested depends, but which AgitarOne cannot successfully exercise  
Method interceptors allow you to effectively mock a single method. They allow you to improve the generated tests in cases where the object under test depends upon another, problematic class. Unlike concrete mocks, which do not help when running your tests using the super-runner, the interceptor behavior is valid both during test generation and when you run your tests. For more about intercepting individual methods, see [“Intercepting Method Calls During Test” on page 67](#).
  - ◆ Mock objects to enable AgitarOne to successfully generate tests for your code. Concrete mock objects for which you write the code are the helpers of last resort because they are the most labor-intensive, requiring you to maintain them in the code base. They can also limit the usefulness of the resulting tests. For more about writing concrete mocks, see [“Specifying Complex Setup with Setup Helpers” on page 60](#).

## Test Helper General Requirements

For all test helpers, keep in mind the following requirements and restrictions on their behavior:

- Test helpers must not make method calls on concrete mock objects, whether directly or indirectly.  
Some AgitarOne experts, including the Spring and Hibernate experts, implement concrete mocks. If you have applied experts to your project, disable them before generating tests when using helpers.
- Test helpers must not call agitation factories, whether directly or indirectly.
- Test helpers should return either immutable objects or new but identical objects.
- Test helpers must not call Mockingbird APIs directly.

Failure to observe these requirements might result in your helper being ignored during test generation.

**NOTE:** You cannot write a test helper for a class in the default package.

---

## Improving Test Generation with Test Data Helpers

*Test data helpers* are methods that produce objects in particular states that are useful for testing or objects that are difficult to construct. When you provide the right test data helpers, AgitarOne can generate tests that use real instances of your classes instead of using mock objects. Test data helpers can be global—applicable everywhere in a project—or scoped to a particular class.

This section contains the following topics:

- [About Test Data Helpers](#)
- [How AgitarOne Test Generation Selects Test Data Helpers](#)
- [Writing Test Data Helpers](#)
- [Test Data Helper Examples](#)

## About Test Data Helpers

Test data helpers are based on the **ObjectMother** pattern created by ThoughtWorks®. To create a test data helper, you implement the interface **com.agitar.lib.TestHelper**. Test data helpers are methods whose name starts with **create** and that return an object of the type you want to use. If the method is non-static, then AgitarOne assumes that the corresponding helper class has a default constructor. Test data helpers require the variable **AGITAR\_TEST\_LIB** in the build path of the project.

**TIP:** Before trying to use a test helper, use AgitarOne to agitate and generate tests for your helper code.

If you use test data helpers during test generation, the AgitarOne server issues warnings under the following circumstances:

- If a helper method throws an exception, then the log will contain a warning so that you can investigate the cause of the exception and correct the problem.
- If a helper class contains static mutable fields, then the log will contain a warning telling you not to use mutable static fields in a helper.

As a general rule, do not declare any fields in your test helper classes. Test data helpers should construct new objects to return within their method body. Do not call mock object methods in test data helpers.

## How AgitarOne Test Generation Selects Test Data Helpers

You can save test data helpers in any source folder in a project. When AgitarOne needs an object of a particular type for a generated test, it uses any test data helpers that are available. To decide which test data helpers to use, AgitarOne uses the following criteria:

- Only use a data helper if it increases coverage or enables more outcomes.  
If your helper class includes multiple **createX()** methods, AgitarOne uses as many of them as help generate unique tests.
- Match available helpers in the following order, based on their scope:
  - ◆ Use helpers scoped to a particular class first.
  - ◆ Use global helpers in the same package as the source class.
  - ◆ Use global helpers in other packages.
  - ◆ Use the agitation auto-factory.
  - ◆ Use helpers that are scope to another class, if useful.
- Use global helper methods in the order they appear in the class, from top to bottom; the first helper found becomes the default.
- If a particular helper does not produce useful objects, try the next available helper.
- Choose data values used in helpers based on the helper's scope.
  - ◆ Use values from helpers scoped to a particular class first.

- ◆ Use canonical default values.
- ◆ Use values from global helpers in other packages.

AgitarOne might optimize tests which use various data types such as **String**, **Class**, Java primitives, and primitive wrappers by placing the desired value directly into the generated code rather than by tracking the sequence of calls to the helpers that generated them. In part, this is to improve the readability of the generated test code, where a long string of calls might confuse the main flow of control.

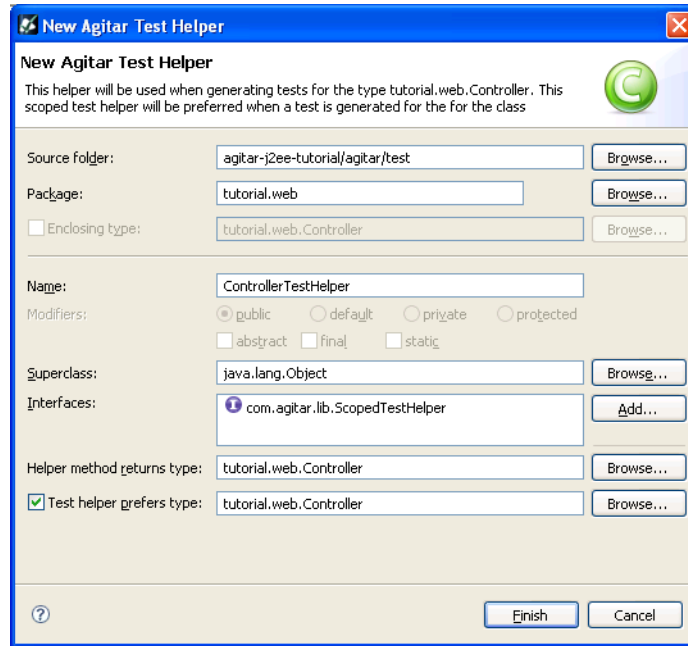
However, this means that while some uses of test helpers are easily detectable in your generated tests by looking for references to helper methods, other uses are harder to discover except by scanning the tests. For example, if you use a test helper to construct a difficult-to-construct **String** (for example, to cover a **String.matches()** call in the code under test), the **String** returned by the helper appears directly in the tests. You need to scan the tests to see your helper-generated **String** being used.

## Writing Test Data Helpers

To generate a test helper method stub that returns a specific type:

1. Select that type in the Java editor within Eclipse.
2. Right-click the type and choose **Agitar>Create Test Helper** (or you can choose **File>New>Other**, and under the **Agitar** category, select **Agitar Test Helper**).

The test helper wizard appears.



The new test helper wizard tries to fill in both the type to create and the preferred type for the test helper automatically from your selection in the editor. By default, the generated test helper class will be a **ScopedTestHelper**, which means you can set it to be preferred for a particular class during test generation.

To make a globally applicable test helper, uncheck the **Test helper prefers type** checkbox. The generated test helper method will be called `createXXX`, where `XXX` is the type that the test helper returns. The method is stubbed out, so you will need to fill in the actual method body to create a useful test helper.

The wizard automatically adds the variable **AGITAR\_TEST\_LIB** to the build path for you.

**NOTE:** While you are writing your test helper, you might want to re-generate tests for your class several times to see how the helper is affecting the generated tests. If test generation is taking very long every time, you can shorten that cycle time by turning off tests that use aggressive mocking. Instead of simply clicking the **Generate Tests** toolbar button, use the drop-down menu beside it and select **Generate tests without aggressive mocks**.

## Test Data Helper Examples

Most of the time, the test data helpers you write will be global to a project. The AgitarOne server will use the helper methods to create test objects each time an object of that type is required. In some cases, however, you might want to write test data helpers that apply only to a specific class. The following examples show both global and scoped test data helpers.

### Global Test Data Helpers

In the **agitar-tutorial** project, if the total cost of an order is more than \$500.00, the customer gets a 10% discount on that order. The following helper class creates **Order** objects, one that will test the code that applies a discount to an order, and one that does not qualify for a discount:

```
package tutorial;
import com.agitar.lib.TestHelper;
public class OrderHelper implements TestHelper {
    public static Order createOrderWithDiscount() {
        Order hasDiscount = new Order();
        hasDiscount.addItem(new Product("B-1224-42-C",
            "Widget", 125), 5);
        return hasDiscount;
    }
    public static Order createOrderWithoutDiscount() {
        Order noDiscount = new Order();
        noDiscount.addItem(new Product("B-1224-42-S",
            "Small Widget", 99), 5);
        return noDiscount;
    }
}
```

Once you regenerate the test for **Order** after adding the helper, you will notice that calls to your newly created test helpers are embedded within the generated tests.

In some cases, AgitarOne optimizes the test code by inlining the values provided by the test helper. If it appears that the helper is not being used, search for the specific strings that your helper provides. For example, when using the helper method **createOrderWithDiscount()** above, you might not see a call to the method in the generated test. Instead, you might see an order with the product code **B-1224-42-S**.

### Scoped Test Data Helpers

To create a scoped test data helper, you implement the method **isPreferredFor()** from the **ScopedTestHelper** interface to specify which class this test data helper applies to.

**NOTE:** AgitarOne prefers a scoped test data helper for the class to which it is scoped, but might also use it for other classes where useful.

The following helper class creates **Order** objects to be preferred when testing both the **Customer** and **User** classes:

```
package tutorial;

import com.agitar.lib.TestHelper;

public class CustomerTestHelper implements ScopedTestHelper {
    public boolean isPreferredFor(Class clazz) {
        return Customer.class == clazz || User.class == clazz;
    }

    public static Order createOrderWithLargeAmountDue() {
        Order expensiveOrder = new Order();
        expensiveOrder.addItem(new Product("A-1298-99-E",
            "Golden Widget", 1000), 1000);
        return expensiveOrder;
    }
}
```

The generated test class for **Customer** includes the following test method, which uses the scoped test data helper:

```
public void testAddOrderThrowsCreditException() throws Throwable {
    Order order = CustomerTestHelper.createOrderWithLargeAmountDue();
    try {
        customer.addOrder(order);
        fail("Expected CreditException to be thrown");
    } catch (CreditException ex) {
        assertFalse("customer.isActive()", customer.isActive());
        assertEquals("ex.getMessage()", "Credit limit exceeded",
            ex.getMessage());
        assertThrownBy(Customer.class, ex);
        assertEquals("customer.getOrders().length", 0,
            customer.getOrders().length);
        assertEquals("customer.getBalanceOutstanding()",
            0.0, customer.getBalanceOutstanding(), 1.0E-6);
        assertEquals("order.getOrderDate()",
            order.getOrderDate(), 60000L);
    }
}
```

---

## Improving Assertions in Generated Tests

In addition to writing data helper methods, you can write assertion helper and observation helper methods to help AgitarOne generate better assertions. These helpers let you provide AgitarOne with some of your knowledge about your application, its requirements, and its implementation.

Assertion helper methods tell AgitarOne about how your code is supposed to behave. You provide an invariant that is always true of a particular type and that AgitarOne can use to generate assertions.

Observation helper methods provide AgitarOne with a value to use in generating assertions.

**NOTE:** Both assertion helpers and observation helpers work only when your tests use real objects. If your tests are heavily mocked, you will not see much benefit from such helpers until you write some test data helpers and generate tests that use real objects.

Do not call any mock object methods from your assertion helper. If you do, the invariant will most likely call into a mocked method for which no recording exists, generating test failures.

## Assertion Helper Methods

Invariants express things that should always be true for a class no matter what operations are performed on an instance of that class. They define things that are (or should be) true about member variables and relationships between them across all method calls, for normal outcomes. For example, in a retail application, a product price would never be negative. You could write an assertion helper to tell AgitarOne that the **price** field of your **Product** class is always zero or more.

Only write assertion helper methods for conditions that are always true for the target type. If a condition is only true sometimes, be sure to incorporate the appropriate conditions in the helper method.

Generated tests will include your helper assertions. An exception thrown from a helper method will cause a test failure. This is useful, because it will signal to you when your code is behaving according to specification. You can write assertion helpers before you finish your implementation, knowing that the assertions will fail until your code is correct.

Do not use mocked objects in your assertions. If you do, the assertion will most likely call into a mocked method for which no recording exists, generating test failures.

To write an assertion helper, create a class that implements the **com.agitar.lib.TestHelper** interface and add a **public static void** method whose name starts with the string **assertInvariant**. The method should take one parameter, which is the class that assertion is testing. Each **assertInvariant()** method should include an assertion. For example, the following helper class defines assertions about the behavior of **Customer** objects in the **agitar-tutorial** project:

```
package tutorial;

import junit.framework.Assert;
import com.agitar.lib.TestHelper;

public class CustomerTestHelper implements TestHelper {
```



```

    public static void assertInvariantWithinCreditLimit(Customer customer)
    {
        Assert.assertTrue("Customer must stay under credit limit",
            customer.getBalanceOutstanding() <=
                customer.getCreditLimit());
    }

    public static void assertInvariantNameProperlyFormatted(Customer
        customer) {
        String customerName = customer.getName();
        Assert.assertNotNull("Customer name cannot be null",
            customerName);
        Assert.assertFalse("Customer name cannot be empty",
            "".equals(customerName));
        boolean nameHasNewLine = customerName.indexOf('\r') >= 0 ||
            customerName.indexOf('\n') >= 0;
        Assert.assertFalse("Customer name cannot have newline characters",
            nameHasNewLine);
    }
}

```

During test generation, AgitarOne applies your assertions to all objects available within the test method scope of the specified type, including the return value and the **this** object for the test call, as well as any parameters. For example the **CustomerAgitarTest** class includes the following test method:

```

public void testAddOrder() throws Throwable {
    Customer customer = new Customer("testCustomerName");
    Order order = new Order();
    customer.addOrder(order);
    assertEquals("customer.getOrders().length", 1,
        customer.getOrders().length);
    assertSame("customer.getOrders()[0]", order,
        customer.getOrders()[0]);
    assertEquals("customer.getBalanceOutstanding()", 0.0,
        customer.getBalanceOutstanding(), 1.0E-6);
    CustomerTestHelper.assertInvariantWithinCreditLimit(customer);
    CustomerTestHelper.assertInvariantNameProperlyFormatted(customer);
}

```

## When Assertions Fail

When you write assertion helper methods, AgitarOne generates an assertion for each object that matches the specified type, whether it will pass or not. This lets you generate failing tests on purpose. When AgitarOne detects that an assertion will fail, it adds a comment above the assertion, like this:

```
// TODO: invariant assertion fails
```

If your class under test uses another class, and that class has assertion helpers with assertions that fail, AgitarOne includes those failing assertions in your generated tests. These failing tests will help you identify class invariants that are violated somewhere in your class under test.

Consider the following example assertion helper:

```
package tutorial;

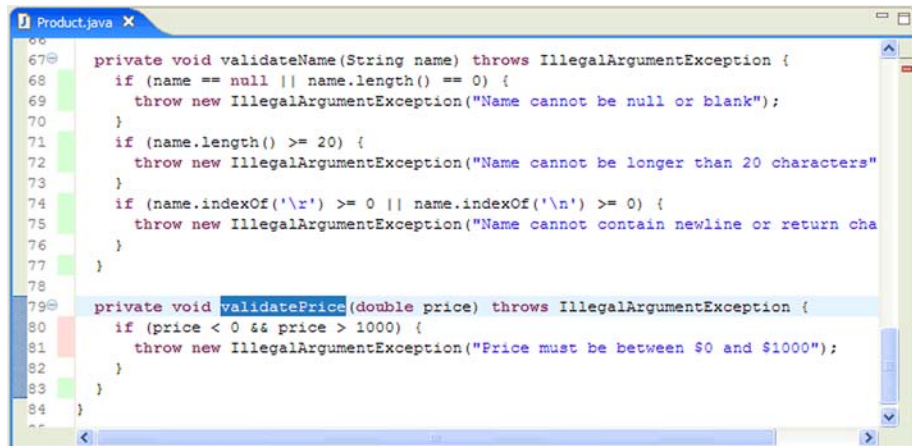
import junit.framework.Assert;
import com.agitar.lib.TestHelper;

public class ProductTestHelper implements TestHelper {
    public static void assertInvariantPriceWithinRange(Product product) {
        Assert.assertTrue("Product price must be between 0 and 1000",
            product.getPrice() >= 0.0 && product.getPrice() <= 1000);
    }
}
```

Using this helper, the generated **ProductAgitarTest** class includes the following test method:

```
public void testConstructor() throws Throwable {
    Product product = new Product("G-9429-94-I", " ", 0.0);
    assertEquals("product.getCode()", "G-9429-94-I", product.getCode());
    assertEquals("product.getPrice()", 0.0, product.getPrice(), 1.0E-6);
    assertEquals("product.getName()", " ", product.getName());
    // TODO: invariant assertion fails
    ProductTestHelper.assertInvariantPriceWithinRange(product);
}
```

When you run this test, it does indeed fail. This is because the **Product** class in the **agitar-tutorial** project has some built-in bugs, and this test has caught one of them. The failure trace reveals that the failed test method is **testConstructor()**. In the source code, you can see that the constructor for **Product** calls **validatePrice()**. Notice that this method has missing coverage on the test for valid values of **price**:



Looking at the **if** statement reveals a logic error. The value of **price** can never be both less than 0 and greater than 1000. The bug here is that **&&** should be **||** instead. After fixing this error, you can regenerate the tests to take the correction into account.

**NOTE:** If you worked through the tutorial lesson entitled “[Improving Code with AgitarOne](#)” on page 23, you fixed this bug, and so the test does not fail here.

## Observation Helpers

Observation helpers describe relationships among objects that you want AgitarOne to watch during the analysis phase of test generation. To define an observation helper method, write a class that implements the `com.agitar.lib.TestHelper` interface and add a **public static** (non-void) method whose name starts with the string `observe`. Observation helpers can return any type, including non-primitives.

AgitarOne uses observation helpers to generate meaningful assertions for the generated tests. During test generation, AgitarOne:

1. Determines what class fields it should write assertions about.
2. Locates any relevant observation helpers and calls those helper methods with each significant value.
3. Adds an assertion for each result.

**TIP:** Do not write your own assertions in an observation helper method. The AgitarOne server will not generate assertions for failing observation helpers.

Observation helpers are especially useful when the state of an object is not transparent, but it is important to determine how the object was transformed by other objects. For example, in an application built on the model-view-controller pattern, it is not always easy to inspect the state of the model. Instead, you can write an observation helper method that provides a view into the model to produce more transparent assertions.

The following example shows an observation helper method that provides information about the number of items in an **Order** object:

```
package tutorial;

import com.agitar.lib.TestHelper;

public class OrderTestHelper implements TestHelper {
    public static String observeOrderItems(Order order) {
        StringBuffer orderBuffer = new StringBuffer();
        OrderItem[] items = order.getItems();
        for (int i = 0; i < items.length; i++) {
            if (i > 0) {
                orderBuffer.append(", ");
            }
            orderBuffer.append(items[i].getQuantity() + " of " +
                items[i].getProduct().getName());
        }
        return orderBuffer.toString();
    }
}
```

The generated test for **Order** uses this helper:

```
public void testConstructor() throws Throwable {
    public void testAddItem() throws Throwable {
        Order order = new Order();
        order.addItem(new Product("Q-8763-87-Z", "testOrderName",
            100.0), 100);
        assertEquals("order.getItems().length", 1,
            order.getItems().length);
        assertEquals("OrderTestHelper.observeOrderItems(order)",
            "100 of testOrderName",
            OrderTestHelper.observeOrderItems(order));
    }
}
```

And the generated test for **Customer** also uses it:

```
public void testAddOrder() throws Throwable {
    Order order = new Order();
    customer.addOrder(order);
    assertEquals("customer.getOrders().length", 1,
        customer.getOrders().length);
    assertSame("customer.getOrders()[0]", order,
        customer.getOrders()[0]);
    assertEquals("customer.getBalanceOutstanding()", 0.0,
        customer.getBalanceOutstanding(), 1.0E-6);
    assertEquals("OrderTestHelper.observeOrderItems(order)", "",
        OrderTestHelper.observeOrderItems(order));
    CustomerTestHelper.assertInvariantWithinCreditLimit(customer);
    CustomerTestHelper.assertInvariantNameProperlyFormatted(customer);
}
```

---

## Specifying Complex Setup with Setup Helpers

Setup helpers are a type of test helper that you can use for the following purposes:

- to provide setup sequences that set up the external environment for test generation
- to provide method interceptors for methods on which the class being tested depends, but which AgitarOne cannot successfully exercise
- to provide mock objects to enable the AgitarOne server to successfully generate tests for your code

Setup helpers are globally scoped; therefore, all setup helpers in the project are invoked each time you generate tests. As an example, you can use a setup helper to set up properties for loggers, or to associate concrete mocks that you have written yourself with specific classes in your project.

If you create concrete mocks using AgitarOne, they will automatically be used during test generation. You do not need to specify them by calling **registerConcreteMock()**.

**NOTE:** While you can use a setup helper to associate a mock object with a class in your project, do not call any mock object methods from the helper itself.

The **SetupHelper** interface extends **TestHelper** and includes the following methods:

<b>setUpConcreteMocks()</b>	Use this method to map concrete mocks. When you use this method, only include calls to <b>ConcreteMocks.createDefaultConcreteMock(Class/String class)</b> or <b>ConcreteMocks.registerConcreteMock(Class/String originalClass, Class/String concreteMock)</b> . For more information, see <a href="#">“Using Concrete Mocks in Test Generation.”</a>
<b>setUpTestGeneration()</b>	The AgitarOne server invokes this method once at the beginning of test generation for any class. Use this method to set up any environment state that will facilitate test generation; for example, setting up properties. This method has security privileges, so you can use it to set up most kinds of environment state.
<b>setUpTestCase()</b>	The AgitarOne server does not call this method during test generation; instead it is automatically included in the <b>setUp()</b> method of the generated test. For example, if you set up logging properties for test generation, then set up the same logger in the <b>setUpTestCase()</b> method so that tests can be run similarly. This method is restricted to regular Java user privileges.
<b>tearDownTestGeneration()</b>	Use this method to reverse any setup done by <b>setUpTestGeneration()</b> .
<b>tearDownTestCase()</b>	Use this method to reverse the setup done by <b>setUpTestCase()</b> . It will be embedded in the <b>tearDown()</b> method of the generated test case.

The setup and teardown methods in **SetupHelper** are called separately, rather than being called by the same object as a pair of methods. As such, when using these methods you should only set up static states. Instance states should not occur in these helper classes.

## Method Interceptors

If you have a class that is not testable because it is calling into a problematic class, you can specify a method interceptor to provide alternative behavior for that problem class. Method interceptors are a special kind of setup helper. For more information about them, see [“Intercepting Method Calls During Test”](#) on page 67.

## Java Security and Test Helper Methods

AgitarOne runs using a standard Java security manager and policy to protect your host systems. Test helpers use the same security settings as the rest of your project code. Among other restrictions, test helpers:

- can only resolve localhost or 127.0.0.1
- can read system properties
- can only read files from the project directory, classpath, or Java home directory (specified by the system property **java.home**)
- can only read and write files into the **temp** directory pointed to by the system property **java.io.tmpdir**

You can find the details of the default security settings that establish these restrictions in [“About the Default Security Settings.”](#)

The **SetupHelper** methods **setUpTestGeneration()** and **tearDownTestGeneration()** are exempted from the security policy and do not have these restrictions.

## Using Concrete Mocks in Test Generation

To supply concrete mocks for the AgitarOne server to use in test generation, use the following methods of the class **ConcreteMocks**. Because **createDefaultConcreteMock** creates a concrete mock with auto-generated values, and **registerConcreteMock** works with user-designed mock classes, you should not try to register a class for which you have created a default concrete mock.

---

**registerConcreteMock(originalClass, mockClass)**  
**registerConcreteMock(originalClassName, mockClassName)**

Use this method to merge the named mock class implementation into the original class. For example, if you call this method with **Foo** and **MockFoo**, any future references to **Foo** will actually refer to a hybrid class where any implementation in **MockFoo** overrides the original implementation in **Foo**.

**NOTE:** If you create concrete mocks using AgitarOne, they will automatically be used during test generation. You do not need to specify them by calling **registerConcreteMock()**.

---

**createDefaultConcreteMock(originalClass)**

Use this method to mock every method in a class to return auto-generated values.

---

**NOTE:** Because the **ConcreteMock** classes have been moved to a different jar file, **ConcreteMocks** created in Agitator 3.0 will not compile correctly. To finish the compilation, press **Ctrl+1 (Quick Fix)** to add the new **AGITAR\_TEST\_LIB** variable.

When you use a **ConcreteMock** in a generated test, you can expect that **Mockingbird** code will be inserted into the tests to reproduce the behavior of the mocked class. For example, if **MockFoo** has a method **getValue()** which always returns “2”, the generated test will contain calls such as:

```
"Mockingbird.setReturnValue(mockFoo.getValue(), 2)"
```

**TIP:** Remember, you create the mock object in your test helper. Do not call any mock-object methods in the test helper code.

## Preventing Assertions and Automatic Mocking

This API can be used to turn off assertion creation for field names or to turn off mocking of objects of a particular type. Below is an excerpt of a SetupHelper class that uses this API.

```
public void setUpTestGeneration() throws Exception {
    // do not mock java.util.List types
    AgitarTestUtility.getAgitarPreferences().neverMockClass("java.util.List");
    //do not create assertions for field names CVSID across the project
    AgitarTestUtility.getAgitarPreferences().neverAssertOnField("CVSID");
    //do not create assertions for the field name lastUsedDate in class
    com.mycompany.utils.DateManager
    AgitarTestUtility.getAgitarPreferences().neverAssertOnField("com.mycompany.
    utils.DateManager", "lastUsedDate");
}
```

## Setup Helper Example

The following example shows a setup helper illustrating the setup for the JNDI initial lookup:

```
package helpers;

import com.agitar.lib.ConcreteMocks;
import tutorial.hibernate.HibernateDataAccess;
import tutorial.web.DataAccessException;
import tutorial.web.ShoppingCart;

import com.agitar.lib.SetupHelper;

public class TutorialTestHelper implements SetupHelper

    /* setup helpers behave as normal test helpers as well - so you can
       embed createXXX methods in setupHelpers if you want */
```

```

public static String createProductCode() {
    return "A-1234-12-B";
}

public void setUpConcreteMocks() {
    ConcreteMocks
        .createDefaultConcreteMock("com.acme.DifficultToInitClass");
}

public void setUpTestGeneration() {
    /* initializes Hibernate - so that Hibernate doesn't time out during
    test generation */

    try {
        new ShoppingCart(new HibernateDataAccess());
    } catch (DataAccessException e) {}

    /* binds the given double to the JNDI with the name
    java:comp/env/MaxDebit */

    AgitarTestUtility.bindJndi("java:comp/env/MaxDebit",
        new Double(1000.0));
}

public void setUpTestCase() {}

public void tearDownTestCase() {}

public void tearDownTestGeneration() {}
}

```

## Using the MockRunner Library

AgitarOne bundles the MockRunner JAR, which provides useful mock objects for J2EE interfaces including JDBC and JMS. This library complements the **org.agitar.mock JAR** which also provides mock objects. You can create test helpers with these mock object implementations.

For documentation about the MockRunner library, refer to the [MockRunner's website](#).

To use MockRunner, add the JAR and its dependencies as a user library. To add an **AGITAR MOCKRUNNER** user library to the project's classpath, perform the following steps:

1. Select your project, right-click, and select **Build Path>Configure Build Path**.  
The project properties dialog box is displayed.
2. Select the **Libraries** tab and click **Add Library**.  
The **Add Library** dialog box is displayed.
3. Select **User Library** and click **Next**.



4. Click **User Libraries**.

The **Filtered Preferences>User Libraries** dialog box is displayed.

5. If the **AGITAR MOCKRUNNER** user library already exists, select it and click **Finish**.  
Confirm your selections on the remaining dialog boxes.

6. If the **AGITAR MOCKRUNNER** user library does not yet exist, create it:

a. Click **New**.

The **New User Library** dialog box is displayed.

b. For **User library name** enter **AGITAR MOCKRUNNER**.

c. Select **AGITAR MOCKRUNNER** and click **Add JARS**.

d. Browse to the **plugins/com.agitar.eclipse.api\_version\_number/lib/ext** directory in your Eclipse installation and add all the JARs in that folder except **org.agitar.mock.jar** and **org.agitar.mock5.jar**.

**NOTE:** Omit the **org.agitar.mock.jar** and **org.agitar.mock5.jar** JARs.

e. Click **OK** to close the **Preferences (filtered)** dialog.

f. Make sure **AGITAR MOCKRUNNER** is checked and click **Finish** to close the **Add Library** dialog.

g. Click **OK** to close the **Properties for project** dialog box.

**NOTE:** When you upgrade your client plug-ins (see [AgitarOne Client Plug-In Updates](#)), *DO NOT* remove the earlier plug-ins from the Eclipse directory unless you also update your project to point to the updated Mockrunner jars.



# 6

## Intercepting Method Calls During Test

Method interceptors allow you to substitute behavior during test for a method's original behavior. You specify a method interceptor in a setup helper.

Method interceptors help AgitarOne both during test generation and during agitation. Unlike concrete mocks, which do not help when running your tests using the super-runner, the interceptor behavior is valid both during test generation and when you run your tests.

The sections that follow introduce the APIs that participate in method interception.

- [Method Interception APIs](#)
- [The Process of Programming Method Interception Using the MethodInterceptor Interface](#)
- [Method Interceptor Examples](#)

---

### Method Interception APIs

All interceptor APIs reside in the **com.agitar.lib.interceptor** package. AgitarOne includes the following APIs for programming method interception:

API	Purpose
<a href="#">MethodInterceptor Interface</a>	Defines the interface for you to identify the method to be intercepted and its alternative behavior during test.
<a href="#">InterceptorWithDefaultValue Class</a>	A <b>MethodInterceptor</b> implementation that replaces the intercepted method's behavior with a return type defined by <b>com.agitar.lib.mockingbird.DefaultValueMap</b> .
<a href="#">InterceptorWithFactory Class</a>	A <b>MethodInterceptor</b> implementation that replaces the intercepted method's behavior with that of an agitation factory.
<a href="#">Interceptor Class</a>	A singleton that tracks all interceptor assignments.

## MethodInterceptor Interface

The **MethodInterceptor** interface defines the following methods for your implementation:

<b>getDeclaringClass()</b>	Implement this method to return the name of the class whose method you are intercepting.
<b>getReturnType()</b>	Implement this method to return the return type of the method you are intercepting.
<b>getSignature()</b>	Implement this method to return the signature of the method you are intercepting.
<b>invoke(Object obj, Object[] args)</b>	Implement this method to provide the replacement behavior for the method you are intercepting.

## Working with the MethodInterceptor Interface

When working with the **MethodInterceptor** interface, you write your own class that implements the interface, then instantiate the class in your project's setup helper. For an example of how to work with this interface, see [“Simple Method Interception Example” on page 71](#).

The following special cases pertain when working with this interface.

### Intercepting a void method

The **invoke()** method must return **null**.

The **getReturnType()** method must return **“void”**.

### Intercepting a method that returns a Java primitive

The **invoke()** method must return the fully qualified Java wrapper class. For example, if the method returns **int** the **invoke()** method returns **java.lang.Integer**.

The **getReturnType()** method must return the primitive name as a string. For example, if the method returns **int** the **getReturnType ()** method returns **“int”**.

### Additional restrictions

You cannot intercept any system class methods, constructors, or static initializers.

## InterceptorWithDefaultValue Class

The **InterceptorWithDefaultValue** is a singleton implementation of the **MethodInterceptor** interface that replaces the intercepted method's behavior with a default return type as defined by **com.agitar.lib.mockingbird.DefaultValueMap**. This class includes the following methods:

<b>addInterceptor(Class type), addInterceptor(Method method)</b>	When you pass a <b>Method</b> , the method is intercepted and the default value defined in <b>DefaultValueMap</b> is returned. When you pass a <b>Class</b> , all class methods are intercepted.
<b>getDeclaringClass(), getReturnType(), getSignature()</b>	Returns the identifying values for the method being intercepted.
<b>invoke(Object obj, Object[] args)</b>	Returns a default value by calling <b>DefaultValueMap.returnDefaultValue()</b> .

AgitarOne applies the **InterceptorWithDefaultValue** class when the class under test uses another class, and that second class's static initializer throws an exception. AgitarOne uses this interceptor to provide default return values for all of the second class's methods, so that test generation can continue. If, upon examination, the generated tests are sufficient for your purposes, you can keep them. If they are not, you can fix the `<clinit>` method, and possibly write test helpers, to enable AgitarOne to generate more specific tests using actual class instances.

For an example of how the application of the **InterceptorWithDefaultValue** class appears in your user log, see [InterceptorWithDefaultValue Example](#).

## InterceptorWithFactory Class

If you have a factory class that returns the appropriate values for the method you want to mock, or if one of Agitar's library factories returns the appropriate values, you can use that factory class to perform method interception. The **InterceptorWithFactory** class is a singleton implementation of the **MethodInterceptor** interface and includes the following methods:

<b>addInterceptor(Method method, String factoryName), addInterceptor(Method method, Factory factory)</b>	When you pass a <b>String</b> for the fully qualified factory name, the interceptor performs a lookup and assigns a factory that you have configured for agitation. When you pass a <b>Factory</b> class, the interceptor uses that factory for the substitute behavior.
<b>getDeclaringClass(), getReturnType(), getSignature()</b>	Return the identifying values for the method being intercepted.

---

<b>invoke(Object obj, Object[] args)</b>	Returns a value defined by the factory you specified.
--	---

---

When working with the **InterceptorWithFactory** class, you call a method in your project's setup helper, setting the factory class and the method to be intercepted. For an example of how to work with this interface, see [InterceptorWithFactory Example](#).

For information about writing and specifying a factory class, see [Custom Factories and the Factory API](#).

## Interceptor Class

The **Interceptor** class is a singleton that tracks all interceptor assignments. It includes the following methods for your use:

---

<b>add(MethodInterceptor interceptor)</b>	Registers and keeps track of the <b>MethodInterceptor</b> passed as the argument.
<b>clear()</b>	Unregisters all method interceptors in the virtual machine.

---

Other **Interceptor** class methods are for the internal operations of the interceptor. You should not call them directly in your code.

The **Interceptor** class also defines a **CALL\_ORIGINAL** data member. You can use this data member to control method interception depending upon the parameters passed to the method being intercepted. For information about using this data member, see [Method Interception When Certain Parameters Are Passed \(Example\)](#).

---

## The Process of Programming Method Interception Using the MethodInterceptor Interface

When using your own method interception implementation to improve your generated tests, perform the following steps:

1. Create a **SetupHelper** class for your project.

Remember, you are not intercepting the class under test. Rather, you are intercepting a method on a class upon which the class under test depends. One obvious case is when the class under test calls a method that is not yet implemented in the other class.

2. Implement the **MethodInterceptor** interface.

You can define a separate class for the project or define an inner class in the **SetupHelper**, whatever works for your project. Whichever technique you choose, remember to:

- ◆ Import the **com.agitar.lib.interceptor** interfaces and classes that you are using.

- ◆ Implement the interface methods.
  - ◆ Never attempt to intercept a system class method, a constructor, or the static initialization (*<clinit>*) of a class.
3. In the **SetupHelper.setUpTestCase()** method, call **Interceptor.add()** and pass in the **MethodInterceptor** implementation.
  4. In the **SetupHelper.setUpTestGeneration()** method, make the same method call.

For an example of this programming problem, see [Simple Method Interception Example](#).

If you are using either the **InterceptorWithDefaultValue** or **InterceptorWithFactory** implementation, simply import the appropriate class in the **SetupHelper** and call the **addInterceptor()** method.

---

## Method Interceptor Examples

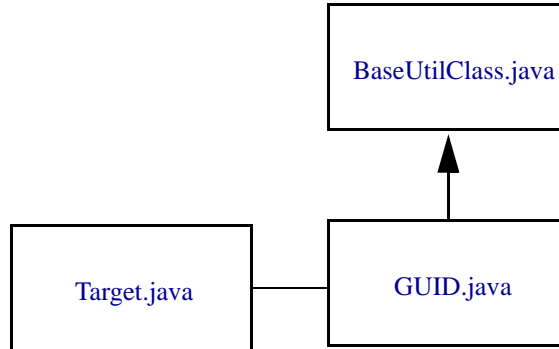
The following examples illustrate how to work with the method interceptor APIs:

- [Simple Method Interception Example](#) illustrates writing your own implementation of the **MethodInterceptor** API and registering the interception in the setup helper.
- [Method Interception When Certain Parameters Are Passed \(Example\)](#) illustrates intercepting only method calls using certain parameters while allowing other tests to call the original implementation.
- [getSignature\(\) Example With A Parameter List](#) shows how to construct the return string of the **getSignature()** method when the intercepted method has different kinds of parameters.
- [InterceptorWithDefaultValue Example](#) illustrates how the automatic application of the **InterceptorWithDefaultValue** class appears in the diagnostic log.
- [InterceptorWithFactory Example](#) illustrates how to use the **InterceptorWithFactory** class to intercept a method and use values from an existing factory class during test.

### SIMPLE METHOD INTERCEPTION EXAMPLE

This example illustrates the kind of testing situation where you would need to use a method interceptor and the process you use to implement one. It is important to remember that you do not intercept anything in the class under test. Instead, you intercept a method that is called, sometimes indirectly through several intervening classes, by that class under test.

This example uses the following simple set of classes:



The **Target** class is the class under test. Its source code follows:

```
package org;

public class Target {
    private GUID gui = new GUID();
    public String getClassGUID() {
        return gui.getGUID(getClass());
    }
}
```

When you generate tests for **Target**, the test for **getClassGUID()** is aggressively mocked:

```
public void testGetClassGUIDWithAggressiveMocks() throws Throwable
{
    Target target = (Target)
        Mockingbird.getProxyObject(Target.class, true);
    GUID gUID = new GUID();
    setPrivateField(target, "gui", gUID);
    Mockingbird.enterRecordingMode();
    Mockingbird.setReturnValue(false, gUID, "getGUID",
        "(java.lang.Class)java.lang.String", "", 1);
    Mockingbird.enterTestMode(Target.class);
    String result = target.getClassGUID();
    assertEquals("result", "", result);
}
```



Clearly, this test has limited utility. To discover why AgitarOne cannot solve this puzzle and use a real **Target** instance in this case, you need to examine the **GUID** and **BaseUtilClass** classes. **Target.getClassGUID()** calls **GUID.getGUID()**. When you examine the **GUID** class, you can see that this simply calls into the parent class's method:

```
package org;

public class GUID extends BaseUtilClass {

}
```

Investigating further, you find that **BaseUtilClass**'s implementation is private:

```
package org;

import java.net.InetAddress;
import java.net.UnknownHostException;

public abstract class BaseUtilClass {
    public String getGUID(Class class1) {
        return lookupGUID();
    }

    private String lookupGUID() {
        try {
            InetAddress host = InetAddress.getByName("middle.of.nowhere");
            return host.getHostName() + ":"
                + host.getCanonicalHostName().hashCode();
        } catch (UnknownHostException e) {
            throw new UnsupportedOperationException();
        }
    }
}
```

Not only is **BaseUtilClass.lookupGUID()** private, it is also using a network lookup to find the new **GUID**. This is not an ideal situation for creating unit tests. Any network problem could cause the JUnit test to fail for reasons that are irrelevant to the code under test, which should be isolated during the test. Method interceptors were created for this purpose.

To get a better test for the **Target.getClassGUID()** method, you can intercept the **BaseUtilClass.lookupGUID()** method and return a **String** of the proper format. The setup helper for this method interception looks like the following:

```
public class TargetSetupHelper implements SetupHelper {

    public void setUpTestCase() throws Exception {
        Interceptor.add(new GuidInterceptor());
    }

    public void setUpTestGeneration() throws Exception {
        setUpTestCase();
    }
}
```

```

private static class GuidInterceptor implements
    com.agitar.lib.interceptor.MethodInterceptor {
    public String getDeclaringClass() {
        return "org.BaseUtilClass";
    }

    public String getReturnType() {
        return "java.lang.String";
    }

    public String getSignature() {
        return "lookupGUID()";
    }
    public Object invoke(Object obj, Object[] args)
        throws Throwable {
        return "test.host.name: 123456";
    }
}
...
}

```

In this setup helper, notice the following items:

- **Interceptor.add()** call in the **setUpTestCase()** method
- the same call, by indirection, in the **setUpTestGeneration()** method to ensure that AgitarOne uses your helper during test generation
- implementation of the **MethodInterceptor** class, with replacement behavior in the **invoke()** method

When you re-generate **Target**'s tests using the helper, you get the following test:

```

public class TargetAgitarTest extends AgitarTestCase {

    public Class getTargetClass() {
        return Target.class;
    }
    protected void setUp() throws Exception {
        new TargetSetupHelper().setUpTestCase();
    }
    ...
    public void testGetClassGUID() throws Throwable {
        String result = new Target().getClassGUID();
        assertEquals("result", "test.host.name: 123456", result);
    }
}

```

Note the following changes in the generated test class:

- The test class's **setUp()** method calls **AgitarSetupHelper.setUpTestCase()**. Every test class in the project will also call this method as part of its setup.
- The method **testGetClassGUID()** now exists, and in its body the string `"test.host.name"` indicates that the test receives the GUID from the method interceptor.

This example has shown how to replace indeterminate objects on which your code under test depends with a single legitimate value. You can, of course, provide as complex an **invoke()** method as is necessary to properly test your code. For an example of how to use the original behavior of such an object in some cases and intercept that behavior in other cases, see [“Method Interception When Certain Parameters Are Passed \(Example\)” on page 75](#).

### METHOD INTERCEPTION WHEN CERTAIN PARAMETERS ARE PASSED (EXAMPLE)

Sometimes you will encounter a situation where a method behaves fine under test when some valid parameters are passed in, but not when other parameters are used. When you have this problem, you can use the `Interceptor.CALL_ORIGINAL` data member to control the method's behavior during test.

For example, consider the following method. In this case, the class under test is calling this method, and `AgitarOne` is unable to test it when the **number** parameter is null.

```
package sample;

public class TargetOfInterception {

    public double compute(Double number) {
        return PI * number.doubleValue();
    }
    private final static double PI = 3.14159;
}
```

In this case, `PI times null` is causing an exception during test. To intercept the `compute` method only when **number** is null:

- Write the setup helper for your project as usual (see [“Simple Method Interception Example” on page 71](#) for details).
- Write a **MethodInterceptor** implementation also as usual (see [“Simple Method Interception Example” on page 71](#) for details)—but see the following details for the **invoke()** method.

- In the **MethodInterceptor.invoke()** method, define different behavior depending on the value of the **number** parameter, as follows:

```
// Invoke method of Method Interceptor
public Object invoke(Object thisObject, Object[] parameters) {
    if (parameters[0] == null) {
        return new Double(0);
    } else {
        return Interceptor.CALL_ORIGINAL;
    }
}
```

In this **invoke()** method, the behavior during test is changed, or intercepted, only if the parameter is null. If it is not null, the return of **Interceptor.CALL\_ORIGINAL** allows the compute method's original behavior to be used.

### GETSIGNATURE() EXAMPLE WITH A PARAMETER LIST

The **getSignature()** method must return a string with the intercepted method's name and a comma-separated list of the parameter types (if any). If any parameters are Java classes, use the fully-qualified class name. For example, a **getSignature()** method for the following method:

```
public int compute(int number, boolean bool);
```

would return the following string:

```
"compute(int, boolean)"
```

While the **getSignature()** method for the following:

```
public int compute(Integer, Double);
```

would return the following string:

```
"compute(java.lang.Integer, java.lang.Double)"
```

### INTERCEPTORWITHDEFAULTVALUE EXAMPLE

For an introduction to the **InterceptorWithDefaultValue** class, see [InterceptorWithDefaultValue Class](#).

**NOTE:** AgitarOne applies this mechanism internally as described below. You do not implement a default value interceptor in your own code.

AgitarOne applies this interceptor when a class used by the class under test throws an exception from its **<clinit>** method. Using the interceptor allows test generation to continue.

For example, the following class depends on the **Bean** class:

```
public class MyClass {
    Bean bean;
    public Bean getBean() {
        return bean;
    }

    public void setBean(Bean bean) {
        this.bean = bean;
    }
}
```

But the **Bean** class's static initialization results in an exception as follows:

```
public class Bean {
    static {
        if (true)
            throw new IllegalArgumentException("Bean exception");
    }
    ...
}
```

Without the interceptor, test generation on **MyClass** would fail to cover much because AgitarOne would never be able to create a **Bean** instance. But with the interceptor, AgitarOne proceeds to generate the tests with default values provided by the interceptor.

If AgitarOne applies this interceptor to one of your classes, you will see messages similar to the following in your user log:

```
[STATUS] [16:40:41:337] Start generating test for class example.MyClass
[INFO] [16:40:41:418] Interceptor added for method: example.Bean.getX()
[WARNING] [16:40:41:420] Exception suppressed and intercepted in static
initializer of example/Bean: java.lang.IllegalArgumentException: Bean
exception
```

As the diagnostic report for **MyClass** indicates, there was an exception in the static initializer of the **Bean** class. To eliminate the interceptor and generate tests with actual **Bean** instances, you would write a test helper to create a **Bean** in the correct state.

## INTERCEPTORWITHFACTORY EXAMPLE

For an introduction to the **InterceptorWithFactory** class, see [InterceptorWithFactory Class](#).

In this example, the **NetworkClient** class reads from a database and gets a list of names. Because the database is not available when the tests are generated or run, AgitarOne cannot test this class on its own. The **getDatabaseNames()** method always throws an exception. Because we want to test the rest of the application, we need to mock out the **getDatabaseNames()** method.

Here is the **NetworkClient** class:

```
package example;
import java.util.List;
public class NetworkClient {
    public String getNameWithM() {
        List<String> nameList = getDatabaseNames("M");
        return nameList.get(0);
    }
    // Method that throws exception and we want to intercept
    public List<String> getDatabaseNames(String namePrefix) {
        // read from database here
        throw new RuntimeException("Network problem occurred");
    }
}
```

The application has a factory class, **ListFactory**, that provides realistic return values for the **getDatabaseNames()** method. To use that factory for method interception, we need to specify it in the setup helper. Here is the code from **NetworkClientSetupHelper**:

```
public void setUpTestCase() throws Exception {
    Method someMethod =
        Class.forName("example.NetworkClient").getDeclaredMethod(
            "getDatabaseNames",
            new Class[] { java.lang.String.class });
    InterceptorWithFactory.addInterceptor(someMethod,
        new ListFactory());
}
```

method 76

## A

**add** method 70

**addInterceptor()** method 69

Agitar Management Dashboard 18

Agitar test runner view 12

agitation

improving 47

agitation configuration and test generation 12

Ant tasks 18

**assertInvariant()** method 56

assertion helpers 55

defined 56

example 56

failing 57

assertions

introduced 21

associated tests 17

auto-mocks 48

## C

characterization tests, introduced 1

class invariants 55, 56

class under test 17

class-based test data helpers 54

**clear()** method 70

**com.agitar.lib.TestHelper** 51

complexity 30, 31

concrete mocks 62

defined 49

migrating Agitator 3.0 projects 62

continuous integration 3

coverage 14

display, customizing 14

missing 26

**createDefaultConcreteMock()** method 62

customizing coverage display 14

## D

- dashboard reports 18
- dashboard reports and test failures 39
- default permissions 10
- deleting tests 44
- developer dashboards, and test failures 42
- diagnosing failures 33
- diagnostic report
  - summary 47

## E

- EasyMock 27
- enterRecordingMode()** 28
- enterTestMode()** 28
- errors 33
- examples
  - method 76
  - assertion helpers 56
  - concrete mock registration 63
  - Hibernate 63
  - intercepting method calls with certain parameters 75
  - InterceptorWithDefaultValue** class 76
  - InterceptorWithFactory** class 77
  - method interceptors 71
  - MethodInterceptor** interface 71
  - observation helpers 59
  - setup helpers 63
  - test data helpers 54–55
- exploring generated tests 22

## F

- failing assertions 57
- failing tests
  - deleting 44
  - diagnosing 33
  - re-generating 43
- failures 33
- find tests 17



## G

- generated test values 26
- generated tests 11
  - and agitation configuration 12
  - difficult to test 30
  - evaluating 30
  - exploring 22
  - failures 34
  - and Java security 10
  - and JNI 9
  - learning about your code 23
  - location 7
  - platform dependencies 34
  - rerunning 13
  - reviewing 21
  - run order dependencies 35
  - running 12
  - what gets mocked 31
- generating tests 6
- getDeclaringClass** method 68
- getProxyObject()** 28
- getReturnType()** method 68
- getSignature()** method 68
- global test data helpers 54

## H

- helpers 47
  - assertion helpers 55
  - introduced 48
  - observation helpers 55
  - requirements for test helpers 50
  - setup helpers 60
  - test data 51

## I

- improving results
  - agitation 47
  - test generation 47
- indirection 30, 31

influencing generated tests 47

**Interceptor** class 70

interceptors 67

**InterceptorWithDefaultValue** class 69

example 76

**InterceptorWithFactory** class 69

example 77

interfaces, mocking 27

invariants 55, 56

**invoke()** method 68, 69, 70

## J

Java security and test generation 10

JNI 9

JUnit runner 12

JUnit tests 6

basic structure 21

and JNI 9

location 7

reading 28

## L

learning from generated tests 23

location of generated tests 7

## M

Management Dashboard 18

method interceptors 67

defined 49

examples 71

**Interceptor** class 70

**InterceptorWithDefaultValue** class 69

**InterceptorWithFactory** class 69

**MethodInterceptor** interface 68

programming 70

requirements 70

restrictions 70

**MethodInterceptor** interface 68

method 76

full example 71

- intercepting only certain calls 75
- missing coverage 26
- missing tests, generating 8
- mock objects 27
  - what gets mocked 31
- Mockingbird 27
- Mockingbird**
  - enterRecordingMode()** 28
  - enterTestMode()** 28
  - example 28
  - getProxyObject()** 28
  - setException()** 28
  - setReturnValue()** 28
- Mockingbird APIs 28
- MockRunner 64
- mocks 62
- multithreaded code 18

## N

- names of test methods 24
- native code and test generation 9
- never mocked 31

## O

- object mother, *see* test data helper
- observation helpers 55
  - defined 56
  - examples 59

## P

- precedence of test data helpers 51

## R

- reading generated tests 28
- refining test data 50
- re-generating tests 43
- registerConcreteMock()** method 62
- regression testing 3
- regression tests
  - saving 17
- removing tests 44

- rerunning tests 13
- reviewing generated tests 21
- run order and generated tests 35
- run order and JUnit tests 35
- running JUnit tests 12

## S

- scoped test data helpers 54
- security 62
- setException()** 28
- setReturnValue()** 28
- setup helpers 60
  - example 63
- setUp()** methods 21
- setUpConcreteMocks()** method 61
- SetupHelper** interface 61
- setUpTestCase()** method 61
- setUpTestGeneration()** method 61
- stack trace 33
- summary diagnostic report 47
- super-runner** Ant task 18

## T

- team dashboards, and test failures 40
- tearDown()** methods 21
- tearDownTestCase()** method 61
- tearDownTestGeneration()** method 61
- test data 26
  - refining 50
- test data helpers 50
  - defined 49
  - examples 54
  - generating 52
  - global example 54
  - precedence 51
  - scoped example 54
  - writing 52
- test failures
  - importing 39
  - in dashboard 39

- test generation
  - improving 47
  - items never mocked 31
  - and Java security 10
  - missing tests 8
  - and multithreaded code 18
  - overview 1
  - process 1
  - troubleshooting 18
  - tuning 47
- Test Generation view 6
- test helpers 47
  - and concrete mocks 62
  - introduced 48
  - method interceptors 67
  - requirements 50
  - restrictions 50
  - security 62
  - setup helpers 60
  - test data helpers
- test method names 24
- TestHelper** interface 51
- tests
  - generating 6
  - location 7
- tests, finding 17
- troubleshooting 18

## U

- unit testing, defined 31

## V

- views
  - Agitar test runner 12
  - Test Generation 6

## W

- WithAggressiveMocks**
  - introduced 25