Agitar™ TECHNOLOGIES

Observation Driven Testing:

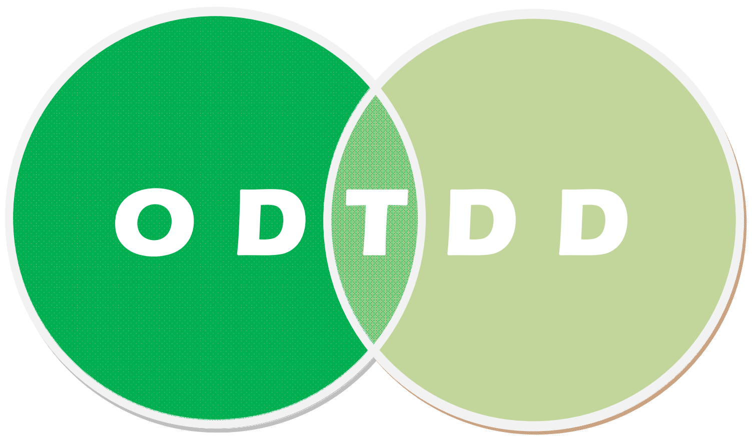
Yes, the code is doing what you want.
By the way, *what else* is it doing?

Executive Summary

Observation-Driven Testing (ODT) is a development methodology for organizations intent on maximizing the effectiveness of development teams while also maximizing code quality. ODT is complementary to, yet not dependent on, the widely adopted **Test-Driven Development** (TDD).

It has been observed that TDD is an excellent methodology for developing “clean code that works” (Ron Jeffries). Its many strengths include the ability to create code that does only what you want and to create a thorough set of automated tests. However, by itself, TDD is incomplete as a coding and testing methodology because of the fact that it creates code that does only what you want without taking into account unintended side effects. ODT fills that void, using automated tools to test the behavior of code and providing developers with actionable observations about possible unintended side effects.

If your Java development teams subscribe to the Test Driven Development methodology, or are considering moving in that direction, you need to know that ODT is strong where TDD is weak.



Highlights:

- Summary of TDD, with its basic steps and its strengths/weaknesses relative to unintended code behavior
- Summary of ODT, with its basic steps and its strengths/weaknesses relative to unintended code behavior
- How TDD and ODT are complementary, especially relative to unintended code behavior
- Summary of automated tools to implement ODT, for Java
- Sample ODT observations, of possible unintended code behavior
- Recommendations on when to apply ODT, relative to your project process history, development plans, and existing unit test suite completeness
- Description of a set of steps by which TDD and ODT can be used together, for enhanced test thoroughness and defect avoidance

What is Test-Driven Development (TDD)?

TDD is a very good technique for developing new code and maintaining existing code. It has been called "test-first programming" and "code unit tests first" development. It results in not only a complete set of automated tests at the "unit" level (with high code coverage) that protect your investment when each change is made, but also a code base that has been thoroughly tested step-by-step as it was being developed. These results are in contrast to the results of the all too common "code-first test-as-long-as-you-have-time-left-before-your-deadline" development techniques, which can result in incomplete test sets and limited testing cycles.

Let us assume that requirements for implementation are in a form that we will generically call Use Cases, whether formal or informal, and even if details are negotiable or TBD. Then, a simplified view of TDD is a process that repeatedly, in small increments, performs the following steps:

- **Red Bar** - Implement test(s) based on a Use Case or a developer observation of a requirement, and execute the set of tests, which will result in test failure(s), i.e., a Red Bar test result indicator.
- **Green Bar** - Implement code until the tests execute correctly, i.e., a Green Bar test result indicator.
- **Refactor** – Change code to eliminate the duplication created by focusing on getting tests to work quickly, which also reduces dependencies.

TDD performs implementation from a different angle than *architecture-driven development*. Architecture-driven development first focuses on defining designs that are clean, and often later making changes when specific cases reveal that changes are needed to make the code work. Alternatively, TDD first focuses on implementing code that works, and sometimes later making changes when general cases reveal that changes are needed to make the code meet requirements.

Note that TDD can be used successfully within various methodologies, including but not limited to those based on Agile methods such as eXtreme Programming (XP) and Scrum.

TDD is successful due to the human brain power that repeatedly performs its very focused steps, and that brings in ideas that add thoroughness and reduce duplication. There is limited automation in the process other than automated test execution/reporting and coding tools/IDEs. There is no automation to generate tests for code that doesn't exist.

TDD's strengths and weaknesses relative to unintended code behavior are:

Strengths:

- Defines code requirements more accurately than writing code-first.
- Allows fast code writing through the use of refactoring tools after tests are written.
- Avoids code bloat and feature creep by requiring testability for all implemented features.

Weaknesses:

- Tests/requirements developed before product implementation tend to emphasize only sunny day/success scenarios.
- Tests cannot be automatically generated.
- Tests do not cover cases that developers do not think of, which may include some general cases encompassing the interactions of many code units.

What is Observation-Driven Testing (ODT)?

ODT is a good technique for identifying code behavior that is unexpected. An automated tool can test code by executing it in many ways based on the contents of the code units (e.g., Java methods) and their relationships, and then provide its observations about code behavior to the developer for review. This exploratory testing of code has been termed "agitation" and is the key feature of AgitarOne – Agitator, from Agitar Technologies. In order to find these observations, each code unit is executed multiple times with various input values using exploratory techniques. Then, the developer is presented with normal and exception outcomes and observations in the form of code expressions, and with code coverage obtained during this testing. This gives several different perspectives on the code's actual behavior that can be used for review and validation. After adjusting code so it works as expected and has the desired number of constraints to be highly maintainable in the future, the developer can promote remaining observation expressions to test points.

We suggest that, in addition to executing available unit tests, the ODT process should be applied to small increments of development. A simplified view of ODT is a process that repeatedly, for small increments of code that might be checked in after a small amount of work (maybe an hour to a day's worth of effort), performs the following steps:

- **Agitate Code** – Request an automated tool to perform exploratory testing and provide its observations; and apply techniques to improve code coverage where code constrains coverage.
- **Adjust Code** – Until no Unexpected Observations exist, adjust code to correct the behavior and rerun your automated tests, then re-agitate.
- **Add Test Points** – For Expected Observations, promote them to test points to help constrain the code behavior as known by tests, creating additional expressions, if desired, to include as test points.

ODT's strengths and weaknesses relative to unintended code behavior are:

Strengths:

- Points out unintended behavior for which a developer would usually not write tests.
- Promotes good code construction by pointing out code that is difficult to test, prompting early refactoring.
- Provides an easy mechanism to add test points from observations of concrete behaviors.

Weaknesses:

- Code must exist before observations are made.
- Automated testing is for actual rather than intended code behavior.

How Are TDD and ODT Complementary?

TDD has great value as a software development process but can be incomplete in terms of the creation of code that does what you want without unintended side effects. ODT uses automated tools to test the behavior of code and provide actionable observations about possible unintended side effects. Therefore, because ODT is strong where TDD is weak, **ODT is an excellent complement to TDD as part of your development processes**. Using ODT with your TDD will reduce defects caused by unintended code behavior.

The following table summarizes how ODT complements TDD.

	TDD	ODT
Goal	Make sure code does only what you want	Make sure code does what you want without unintended side effects
What is performed during development and testing?	After requirements in some form exist (formal or informal), automated tests are created. Then code is created to cause failing tests to pass. Then code is refactored to reduce duplication (improving testability and maintainability).	After code exists, code is analyzed by automation to identify observations about behavior that may point out either defects or needs to specify further tests; then code may be changed, further tests created, assertions made, or code refactored.
Focus in reducing defects	Reduces opportunities for untested behaviors, relative to requirements and code coverage	Reduces opportunities for untested behaviors, relative to code base's behavior
Strengths in avoiding code defects from unintended side effects	Specific cases, based on known requirement	General cases, based on code behavior

What Tools Support ODT for Java Projects?

This section summarizes an automated tool, **AgitarOne**, which can support an ODT process for Java. **AgitarOne - Agitator** is a unique developer testing tool that uses *exploratory testing* techniques to execute code and provide actionable behavior information to the developer. The developer then can review each observation and determine whether to change code (to fix unintended behavior), create additional tests, promote observations to assertions, or refactor code for improved testability. Agitator is meant to be used during code development to reduce defects due to unintended code behavior. Only an automated tool can perform such a thorough analysis of behaviors.

AgitarOne - JUnit Generator is also available to support legacy code development. Some projects, such as those that have not used TDD, may not currently have a complete set of unit tests (i.e., with high level of code coverage across the project). Most organizations cannot find resources to go back and manually create a complete set of unit tests, even though they know these tests would have great value in reducing defects going forward. Having a complete set of characterization tests gives you confidence to make changes to existing code with minimized fear of regressions. JUnit Generator can be used to create such a set of tests. It provides a set of automatically-generated, passing tests (no user manipulation required) with 80% or better coverage, using the same exploratory testing techniques as Agitator, plus significant automatic mocking technologies. And it provides easy to use techniques to improve coverage levels from there if desired. The process is simple: automatically generate passing tests, make code changes, run tests, inspect test failures, account for unexpected failures and automatically re-generate tests so they are available for the next code change.

One of the significant obstacles to successful TDD implementation is the amount of effort required to create harnesses for testing. On the one hand, this encourages development designs that make components more easily testable in isolation, because introducing more dependencies increases the testing effort. However, since the test is written before much of the system is implemented, there are inevitable cases where the developer will have to implement interfaces, stubs, mocks, etc. for the integration points as part of writing the test. This can be a substantial effort. The exploratory testing employed by AgitarOne's Agitator and JUnit Generator features automatically creates mock objects to use when executing the code under test. Thus, it can provide feedback about the behavior of the unit of code, even when the external integration points are incomplete.

Sample ODT Observations of Possible Unintended Code Behaviors

This section provides an example of first using a TDD process to create tests and code, then using an ODT process to identify and resolve unintended code behavior.

Use TDD

First, let's say our requirements call for a method to set a model number String for a product. In the Product class, the setter method should check the input parameter for the following:

- String cannot be null
- String cannot be blank
- String cannot contain newline or carriage return characters
- String cannot be greater than 40 characters long

Now we will write the tests to fit our requirements:

```
public void testSetModelWithNewline() throws Throwable {
    Product product =
        new Product("M-6249-56-Y", "testProductName", 100.0);
    try {
        product.setModel("XXX\nXXX");
        fail("Expected IllegalArgumentException to be thrown");
    } catch (IllegalArgumentException ex) {
        assertEquals(
            "ex.getMessage()",
            "Model cannot contain newline or return characters",
            ex.getMessage());
    }
}

public void testSetModelWithReturn() throws Throwable {
    Product product =
        new Product("M-6249-56-Y", "testProductName", 100.0);
    try {
        product.setModel("XXX\rXXX");
        fail("Expected IllegalArgumentException to be thrown");
    } catch (IllegalArgumentException ex) {
        assertEquals(
            "ex.getMessage()",
            "Model cannot contain newline or return characters",
            ex.getMessage());
    }
}

public void testSetModelEmpty() throws Throwable {
    Product product =
            new Product("M-6249-56-Y", "testProductName", 100.0);
        try {
            product.setModel("");
            fail("Expected IllegalArgumentException to be thrown");
        } catch (IllegalArgumentException ex) {
            assertEquals(
                "ex.getMessage()",
                "Model cannot be null or blank",
                ex.getMessage());
        }
    }

    public void testSetModel41Chars() throws Throwable {
        Product product =
            new Product("M-6249-56-Y", "testProductName", 100.0);
        try {

             product.setModel("41CharactersXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
            fail("Expected IllegalArgumentException to be thrown");
        } catch (IllegalArgumentException ex) {
            assertEquals(
                "ex.getMessage()",
                "Model cannot be longer than 40 characters",
                ex.getMessage());
        }
    }
        public void testSetModelNull() throws Throwable {
        Product product =
            new Product("M-6249-56-Y", "testProductName", 100.0);
        try {
            product.setModel(null);
            fail("Expected IllegalArgumentException to be thrown");
        } catch (IllegalArgumentException ex) {
            assertEquals(
                "ex.getMessage()",
                "Model cannot be null or blank",
                 ex.getMessage());
        }
    }
}
```

Next we will create the code that will make our tests pass. To improve readability, we will factor out the input checking from the setter into a separate validateModel() method:

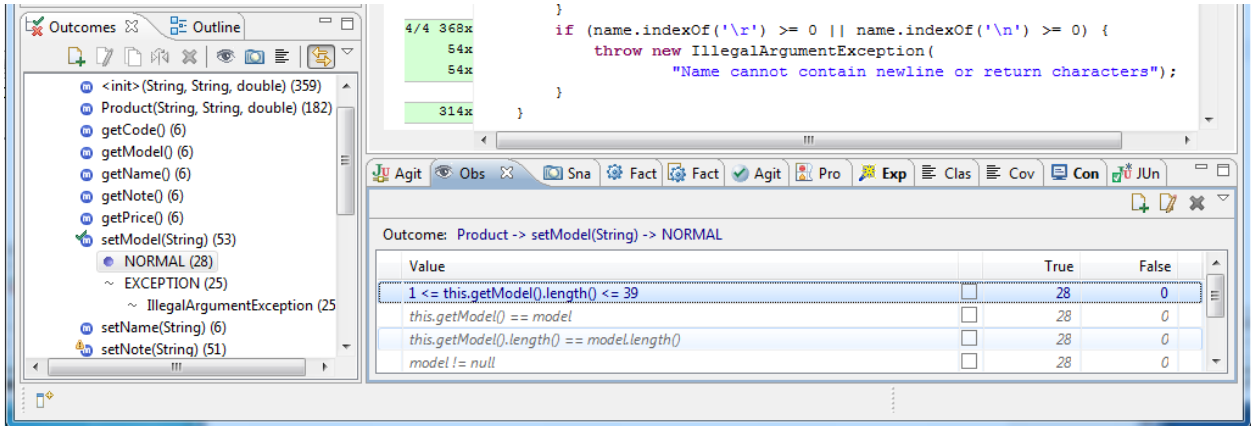
```
private String model;

public void setModel(String model) {
          validateModel(model);
          this.model = model;
}

public String getModel() {
    return model;
}
private void validateModel(String model) {
    if (model == null || model.length() == 0) {
        throw new IllegalArgumentException(
            "Model cannot be null or blank");
    }
    if (model.length() >= 40) {
        throw new IllegalArgumentException(
            "Model cannot be longer than 40 characters");
    }
    if (model.indexOf('\r') >= 0 || model.indexOf('\n') >= 0) {
        throw new IllegalArgumentException(
            "Model cannot contain newline or return characters");
    }
}
```

Use ODT

The new code created by a TDD process seems to fit our requirements just fine, indicated by our passing tests, but is it really doing what we want? Let's agitate it (that is, apply the ODT process) and see if we can spot any bad behavior:



```
4/4 368x        }
                if (name.indexOf('\r') >= 0 || name.indexOf('\n') >= 0) {
     54x            throw new IllegalArgumentException(
     54x                    "Name cannot contain newline or return characters");
                }
    314x    }
```

Outcome: Product -> setModel(String) -> NORMAL

Value		True	False
1 <= this.getModel().length() <= 39	☐	28	0
this.getModel() == model	☐	28	0
this.getModel().length() == model.length()	☐	28	0
model != null	☐	28	0

One glaring observation we can make here is that if we look at our NORMAL outcome from setModel(), it looks like agitation found that the string can be from 1 to 39 characters long but the requirement stated "not over 40". A simple off-by-one error here could be a problem in the future, so we should fix the code, re-run the tests and see if we can make any agitation assertions.

```
            if (model.length() > 40) {
                throw new IllegalArgumentException(
                        "Model cannot be longer than 40 characters");
        }
```

In the code above we have changed the '>=' symbol to a '>' symbol which should fix the problem. If we re-run the tests they all still pass, and if we re-agitate the code we have an observation that reads:

```
1 <= model.length() <= 40
```

This is now consistent with the specified requirements. At this point, we should promote that observation to an assertion. This is a quick alternative to adding another JUnit test, and AgitarOne integrates the results into the same dashboard report along with the outcomes of the JUnit tests.

Thus, the ODT process added to the TDD process improved the code quality.

Recommendations on When to Apply ODT

ODT can provide value regardless of whether you have been using TDD. This section provides recommendations on when to apply ODT, relative to your project's process history, development plans, and existing unit test suite completeness. Additionally, we will discuss when automated unit test generation can be used to shore up test deficiencies if you have not been using TDD or otherwise do not have a complete set of unit tests (i.e., low code coverage).

TDD can be a very good practice as part of a high quality software development process. But TDD is not totally thorough by itself. Even if you have 100% code coverage with your TDD-generated tests, your process can be enhanced using ODT. Furthermore, if TDD has been implemented only partially by not applying it to all portions of the project, or has not been used at all, your unit tests may be incomplete. And therefore automatic test generation should also be employed as you proceed from this point. Your process should be enhanced to use ODT for at least all newly developed and changed code, and possibly also for code that has a history of defects.

The following table summarizes the recommended processes.

TDD	ODT	Automated JUnit Test Generation
After requirements exist, tests are created, then code is created to cause failing tests to pass; then code is refactored to reduce duplication (improving testability and maintainability)	After code exists, code is analyzed by automation to identify observations about behavior that may point out either defects or needs to specify further tests or assertions; then code may be changed, further tests created, assertions made, or code refactored	After code exists and is tested and stable, additional unit tests are created by automation; then after next code change tests are executed and failed tests analyzed for potential unintended defects (regressions); then code may be fixed or refactored, or other tests created

The following table recommends when to use ODT, and also automatic test generation, based on your project's process history, future development plans, and state of existing unit tests.

	Process History with Respect to TDD	Future Development Plans	State of Existing Unit Tests	Recommended Process Enhancements
1	**Just starting development**, **and will be using TDD**	**All new code** to be developed (may have reuse from prior projects)	**None**	**Add ODT** to TDD
2	**All** parts of **existing code base** have been **developed using TDD**	**Much new code** still to be developed	**Fairly Complete** (very high coverage)	**Add ODT** to TDD
3	**Some** parts of **existing code base** have been **developed using TDD**, others not	**Much new code** still to be developed	**Fairly Complete** (very high coverage)	**Add ODT** to TDD; even if you plan to proceed without using TDD for new code, at least use ODT
4	**Some** parts of **existing code base** have been **developed using TDD**, others not [same as 3]	**Much new code** still to be developed	**Incomplete** (much less than 100% code coverage)	**Add ODT** to TDD; even if you plan to proceed without using TDD for new code, at least use ODT **Add automatic JUnit generation** to create a more complete safety net of unit tests
5	**No** parts of **existing code base** have been **developed using TDD**	**Much new code** still to be developed	**Fairly Complete** (very high coverage)	**Add ODT** to TDD; even if you plan to proceed without using TDD for new code, at least use ODT
6	**No** parts of **existing code base** have been **developed using TDD** [same as 5]	**Much new code** still to be developed	**Incomplete** (much less than 100% code coverage)	**Add ODT** to TDD; even if you plan to proceed without using TDD for new code, at least use ODT **Add automatic JUnit generation** to create a more complete safety net of unit tests
7	All, some, or no parts of existing code base have been developed using TDD [same as some above]	**Little or no new code is to be developed**; only minor maintenance changes planned	**Fairly Complete** (very high coverage)	**Apply ODT** to **areas of code** that have a **significant history of having defects**, to help find potential remaining defects
8	All, some, or no parts of existing code base have been developed using TDD [same as some above]	**Little or no new code is to be developed**; rather only minor maintenance changes planned	**Incomplete** (much less than 100% code coverage)	**Add automatic JUnit generation** to create a more complete safety net of unit tests **Apply ODT** to **areas of code** that have a **significant history of having defects**, to help find potential remaining defects

Steps to Use TDD and ODT Together For Maximized Defect Avoidance

This section provides a suggested set of steps by which TDD and ODT can be used together for maximized defect avoidance. Note that your processes and terminology may be slightly different, but the basic concept is that you would perform TDD then ODT in small, incremental steps during development. Reference earlier sections of this document for more detail on the processes.

Perform the following in small, incremental steps:

1. Apply TDD to develop a small set of tests and clean, working code, to the point that you are ready for checkin; possibly one or several hours of work up to a day's work.
    - Red Bar
    - Green Bar
    - Refactor
2. Apply ODT to improve that code, to the point that you have no remaining unintended observations.
    - Agitate Code
    - Adjust Code
    - Add Test Points

Conclusion: TDD + ODT = Increased Quality

It is never a good idea to put all of your eggs in one basket. No matter what type of development and testing philosophies you adapt, including TDD, there can be holes in your attempts to reduce defects and improve maintainability. ODT can help fill such holes with its automated code behavior observation. A result of successfully applying multiple techniques is that your code will be more constrained, more maintainable, and better tested. We believe that ODT using AgitarOne can improve your code quality.

Acknowledgements

We thank Kent Beck for his contributions to unit testing concepts, to the TDD process (see his book, Test-Driven Development By Example), and to ODT (as consultant to the company that created AgitarOne, the premier ODT tool and JUnit Generator tool for Java). We also acknowledge contributions of many who assisted with the concept and development of the agitation technology currently employed within the Agitator and JUnit Generator features of the AgitarOne product (www.agitar.com).

We also acknowledge a Kevin Lawrence blog entry of December 9, 2004 that summarized the value of using TDD together with the Agitator tool process (described in this paper as ODT) for improved results.   Kevin said "… specialize until the general case becomes apparent. This is what TDD teaches us. Each JUnit becomes a special case. It's easier to grok the special cases and it's easier to do the required setup for a special case. However it's often hard to see the general rule from a collection of special cases and that's where DbC [Design By Contract] comes in. Agitator is good at finding and enforcing the general rule – it will even check that the general rule holds as each individual JUnit case runs."

For more information about AgitarOne, please visit us at www.agitar.com.