# GDSC Test Documentation

***Release 1.1***

**Alex Herbert**

**Apr 01, 2022**

# Contents

GDSC Test provides utility functionality for writing Java tests including:

- Predicate library for single- or bi-valued test predicates using Java primitives
- Assertion utilities for asserting predicates
- Dynamic messages implementing `Supplier<String>`
- Configurable random seed utilities

Note: The GDSC Test library works for Java 1.8+.

The code is hosted on GitHub.

# CHAPTER 1

# Predicate Library

The GDSC predicate library is an extension of the `java.util.function` primitive predicates `DoublePredicate` and `LongPredicate` to all java primitives. These are functional interfaces for single or bi-valued predicates providing the following:

```
@FunctionalInterface
public interface TypePredicate {
    boolean test(type value);
    // default methods
}

@FunctionalInterface
public interface TypeTypeBiPredicate {
    boolean test(type value1, type value2);
    // default methods
}
```

where `<type>` is one of: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long` and `short` (see why all the primitives?).

The predicate functional interfaces provide:

- The logical `or`, `and` and `xor` operations with other predicates
- Negation to an opposite test

Standard predicates are provided to test equality and closeness within an absolute or relative error (see Relative Equality). Floating-point numbers can be compared with a units in the last place (ULP) tolerance.

Combined predicates exists to combine two single-valued predicates into a bi-valued predicate that tests each input value with a distinct predicate.

CHAPTER 2

# Assertion Utilities

Support for testing using a test framework is provided with a utility class that will test primitive value(s) using a single or bi-valued predicate. An `AssertionError` is generated when a test fails. For example a test for relative equality:

```
import uk.ac.sussex.gdsc.test.api.TestAssertions;
import uk.ac.sussex.gdsc.test.api.Predicates;

@Test
public void testRelativeEquality() {
    double relativeError = 0.01;
    double expected = 100;
    double actual = 99;

    DoubleDoubleBiPredicate areClose = Predicates.
↪doublesAreRelativelyClose(relativeError);

    // This will pass as 99 is within (0.01*100) of 100
    TestAssertions.assertTest(expected, actual, areClose);
}
```

A test for equality with units in the last place (ULP):

```
import uk.ac.sussex.gdsc.test.api.TestAssertions;
import uk.ac.sussex.gdsc.test.api.Predicates;

@Test
public void testUlpEquality() {
    int ulpError = 1;
    double expected = 100;
    double actual = Math.nextUp(expected);

    DoubleDoubleBiPredicate areClose = Predicates.doublesAreUlpClose(ulpError);

    TestAssertions.assertTest(expected, actual, areClose);
    TestAssertions.assertTest(expected, Math.nextUp(actual), areClose.negate());
}
```

All provided implementations of the `TypePredicate` or `TypeTypeBiPredicate` interface implement `Supplier<String>` to provide a text description of the predicate. This is used to format an error message for the failed test.

Nested arrays are supported using recursion allowing testing of matrices:

```
IntIntBiPredicate equal = Predicates.intsAreEqual();
Object[] expected = new int[4][5][6];
Object[] actual = new int[4][5][6];
TestAssertions.assertArrayTest(expected, actual, equal);
```

# CHAPTER 3

# Dynamic Messages

Support is provided for dynamic messages using `Supplier<String>` suppliers. These store updatable arguments to pass to `String.format(String, Object...)` for an error message, for example:

```
import uk.ac.sussex.gdsc.test.utils.functions.IndexSupplier;

final int dimensions = 2;
final IndexSupplier msg = new IndexSupplier(dimensions);
System.out.println(msg.get());
msg.setMessagePrefix("Index: ");
msg.set(0, 23); // Set index 0
msg.set(1, 14); // Set index 1
System.out.println(msg.get());
```

Reports:

```
[0][0]
Index: [23][14]
```

All setters return the message as a `Supplier<String>` for testing within loops that require a message. For example using JUnit 5:

```
import uk.ac.sussex.gdsc.test.utils.functions.IndexSupplier;

int dimensions = 2;
IndexSupplier message = new IndexSupplier(dimensions);
message.setMessagePrefix("Index ");
// The message will supply "Index [i][j]" for the assertion, e.g.
message.set(0, 42);
Assertions.assertEquals("Index [42][3]", message.set(1, 3).get());

int size = 5;
int[][] matrix = new int[size][size];
for (int i = 0; i < size; i++) {
    message.set(0, i);
```

(continues on next page)

(continued from previous page)

```
    for (int j = 0; j < size; j++) {
        // The message will supply "Index [i][j]" for the assertion
        Assertions.assertEquals(0, matrix[i][j], message.set(1, j));
    }
}
```

CHAPTER 4

# Configurable Random Seed

Support for test randomness is provided using a single `byte[]` seed:

```
import uk.ac.sussex.gdsc.test.utils.TestSettings;

byte[] seed = TestSettings.getSeed();
```

A `RandomSeeds` class is provided to convert the `byte[]` to `int[]`, `long[]` or `long`.

The seed can be set using a Hex-encoded string property:

```
-Dgdsc.test.seed=123456789abcdf
```

If not set then the seed will be randomly generated using `java.security.SecureRandom` and logged using the configurable `java.util.logging.Logger`. This allows failed tests to be repeated by re-using the generated seed that triggered a test failure.

Extra support for seeded tests is provided for JUnit 5 using a custom `@SeededTest` annotation:

```
import uk.ac.sussex.gdsc.test.junit5.SeededTest;
import uk.ac.sussex.gdsc.test.utils.RandomSeed;

// A repeated parameterised test with run-time configurable seed
// and repeats
@SeededTest
public void testSomethingRandom(RandomSeed seed) {
    long value = seed.getAsLong();
    // Do the test with a seeded random source ...
}
```

The `@SeededTest` is a `@RepeatedTest`. Each repeat will have a unique random seed. The number of repeats can be set using a Java property:

```
-Dgdsc.test.repeats=5
```

An example implementation for test randomness is provided using Apache Commons RNG client API `UniformRandomProvider`, for example:

```
import uk.ac.sussex.gdsc.test.junit5.SeededTest;
import uk.ac.sussex.gdsc.test.rng.RngUtils;
import uk.ac.sussex.gdsc.test.utils.RandomSeed;

import org.apache.commons.rng.UniformRandomProvider;

// A repeated parameterised test with run-time configurable seed
// and repeats
@SeededTest
public void testSomethingRandom(RandomSeed seed) {
    UniformRandomProvider rng = RngUtils.create(seed.get());
    // Do the test ...
}
```

# CHAPTER 5

# Modular Design

GDSC Test is separated into different packages so that the desired functionality can be included as a project dependency.

| Package | Description |
|---|---|
| `uk.ac.sussex.gdsc.test.api` | Predicates and assertions |
| `uk.ac.sussex.gdsc.test.utils` | Utilities for error messages, logging and timing |
| `uk.ac.sussex.gdsc.test.junit5` | JUnit5 annotations |
| `uk.ac.sussex.gdsc.test.rng` | Provides a configurable random source |

CHAPTER 6

Contents

## 6.1 Installation

This package is a library to be used used by other Java programs. It is only necessary to perform an install if you are building other packages that depend on it.

### 6.1.1 Installation using Maven

1. Clone the repository:

```
git clone https://github.com/aherbert/gdsc-test.git
```

2. Build the code and install using Maven:

```
cd gdsc-test
mvn install
```

This will produce a gdsc-test-[module]-[VERSION].jar file in the local Maven repository. You can now build the other packages that depend on this code.

## 6.2 Why all the primitives?

Short answer: To provide **bi-valued predicates for primitives** for use in testing, in particular testing relative equality of nested arrays:

```
@Test
void testNestedArrays() {
    DoubleDoubleBiPredicate equal = Predicates.doublesAreRelativelyClose(1e-3);
    double[][] expected = {
        {1, 2, 30},
```

(continues on next page)

(continued from previous page)

```
        {4, 5, 6},
    };
    double[][] actual = {
        {1, 2, 30.01},
        {4.0001, 5, 6},
    };
    TestAssertions.assertArrayTest(expected, actual, equal);
}
```

### 6.2.1 Java Predicates

Java supports testing of primitives from `java.util.function` using:

```
@FunctionalInterface
public interface DoublePredicate {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param value the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    boolean test(double value);

    // Default methods
}
```

for the `double` primitive type and a similar `IntPredicate` and `LongPredicate` for the `int` and `long` primitive types.

The Java language can natively convert `float` to `double` and `byte`, `char`, and `short` to `int`. So the standard Java predicates can be used to test all primitive types except `boolean`. This can be supported with a simple cast to 1 or 0. Thus *Java predicates already support testing single-valued primitives*.

In the case of bi-valued predicates Java only provides `BiPredicate<T, U>`:

```
@FunctionalInterface
public interface BiPredicate<T, U> {

    /**
     * Evaluates this predicate on the given arguments.
     *
     * @param t the first input argument
     * @param u the second input argument
     * @return {@code true} if the input arguments match the predicate,
     * otherwise {@code false}
     */
    boolean test(T t, U u);

    // Default methods
}
```

Primitives must be tested via the auto-boxing of primitives in their respective class.

### 6.2.2 A Predicate Library

The motivation for the library was to provide **bi-valued predicates for primitives** for use in testing. By providing predicates for all primitives it ensures:

- The test is explicit
- The test can assume a range for the value and natural behaviour
- `boolean` is supported

An example is that the maximum difference between `int` values requires using `long` to prevent overflow and `long` differences require using `BigInteger`.

The predicate library provides the following generic interfaces:

```
@FunctionalInterface
public interface TypePredicate {
    boolean test(type value);
    // default methods
}

@FunctionalInterface
public interface TypeTypeBiPredicate {
    boolean test(type value1, type value2);
    // default methods
}
```

where <type> is one of: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long` and `short`.

The predicates copy the functionality of Java's `java.util.function.Predicate` by supporting default methods for `negate` and the logical combination using `or` and `and` but also add `xor`. However in contrast to the Java version these default interface methods return concrete classes and not lambda expressions. This is to support a feature useful for testing where classes implementing the interface also implement `Supplier<String>` to provide a description. Thus logical combination predicates can provide a logical combination of the description of their composed predicates.

Note that the library duplicates `DoublePredicate`, `IntPredicate` and `LongPredicate`. The GDSC Test versions do not extend their respective Java versions. This avoids a confusing API where predicates do not function identically due to argument types to default methods (`or` and `and`) either accepting `java.util.function` predicates or GDSC Test predicates.

Since these are functional interfaces it is easy to convert between the two using a method reference to the `test` method:

```
final int answer = 42;
uk.ac.sussex.gdsc.test.api.function.IntPredicate isAnswer1 = v -> v == answer;
java.util.function.IntPredicate isAnswer2 = isAnswer1::test;
uk.ac.sussex.gdsc.test.api.function.IntPredicate isAnswer3 = isAnswer2::test;
```

This allows using the GDSC test predicates within the standard Java framework:

```
uk.ac.sussex.gdsc.test.api.function.IntPredicate isEven = v -> (v & 1) == 0;
long even = IntStream.of(1, 2, 3).filter(isEven::test).count();
```

### 6.2.3 Code Generation

The simple and repetitive code for most predicates in the library is auto-generated from templates.

Generation uses the String Template library.

## 6.3 Relative Equality

### 6.3.1 Definition

The GDSC Test library contains assertion functionality for relative equality between numbers.

The implementation of relative equality expresses the difference $\delta$ between $a$ and $b$:

$$\delta = |a - b|$$

relative to the magnitude of $a$ and/or $b$.

A **symmetric** relative error is:

$$e = \frac{|a - b|}{\max(|a|, |b|)}$$

This term is identical if $a$ and $b$ are switched.

The equivalent **asymmetric** relative error $e$ is:

$$e_a = \frac{|a - b|}{|a|}$$

This is the distance that $b$ is from $a$, relative to $a$.

### 6.3.2 Testing Relative Equality

A simple test for relative equality may check that the equality is below a maximum threshold $e_{max}$. For example a **symmetric** test of relative equality can be:

$$\frac{|a - b|}{\max(|a|, |b|)} \leq e_{max}$$

Note that this will fail if both values are zero due to division by zero. It can be rearranged to avoid this:

$$|a - b| \leq \max(|a|, |b|) \times e_{max}$$

The expression is a test that the distance between two values is less than an error, relative to the largest magnitude. This is a test for convergence of two values.

The equivalent **asymmetric** test:

$$|a - b| \leq |a| \times e_{max}$$

is a test that the distance between two values is less than an error, relative to $a$. Since the test is asymmetric it can be used when $a$ is the known (expected) value and $b$ is an actual value to be tested, i.e. test if the actual value is close to the expected value using a relative error.

### 6.3.3 Value Around Zero

The relative equality uses the magnitude of the values to define the scale.

$$e = \frac{|a - b|}{max(|a|, |b|)}$$
$$e_a = \frac{|a - b|}{|a|}$$

In the limit $|a| = 0$ then $e_a = \infty$ but $e = 1$.

The **asymmetric** relative error is thus unbounded.

The limit for the **symmetric** relative error is *2* when $a$ and $b$ are of equal magnitudes and opposite signs, e.g.

$$a = 1$$
$$b = -1$$
$$e = \frac{|a - b|}{\max(|a|, |b|)}$$
$$e = \frac{2}{1}$$

This allows the relative error argument to be checked that it falls within the allowed range $0 \leq e \leq 2$.

### 6.3.4 Testing Close to Zero

The relative error increases as one of the test values approaches zero. The following demonstrates this for the **symmetric** relative error:

$$a = 0.01$$
$$b = 0.00001$$
$$e = \frac{|a - b|}{\max(|a|, |b|)}$$
$$e = \frac{0.00999}{0.01}$$
$$e = 0.999$$

The relative error $e$ will be large ($0.999$) although the absolute difference $\delta$ will be small ($0.00999$).

This can be handled by allowing the $\delta$ to be tested against a relative error threshold $rel_{max}$ **or** an absolute error threshold $abs_{max}$ :

$$|a - b| \leq \max(|a|, |b|) \times rel_{max}$$
$$|a - b| \leq abs_{max}$$

### 6.3.5 Test Predicates

The GDSC Test library contains predicates that test relative equality between two values.

Support is provided for **symmetric** relative equality using the name **AreRelativelyClose** which does not imply a direction. Support is provided for **asymmetric** relative equality using the name **IsRelativelyCloseTo** which implies a direction.

These can be constructed using a helper class:

```
double relativeError = 0.01;

DoubleDoubleBiPredicate areClose = Predicates.
↪doublesAreRelativelyClose(relativeError);

// The AreClose relative equality is symmetric
assert areClose.test(100, 99) : "Difference 1 should be <= 0.01 of 100";
assert areClose.test(99, 100) : "Difference 1 should be <= 0.01 of 100";

// The test identifies large relative error
assert !areClose.test(10, 9) : "Difference 1 should not be <= 0.01 of 10";
assert !areClose.test(9, 10) : "Difference 1 should not be <= 0.01 of 10";


DoubleDoubleBiPredicate isCloseTo = Predicates.
↪doublesIsRelativelyCloseTo(relativeError);

// The IsRelativelyCloseTo relative equality is asymmetric
assert isCloseTo.test(100, 99) : "Difference 1 should be <= 0.01 of 100";
assert !isCloseTo.test(99, 100) : "Difference 1 should not be <= 0.01 of 99";

// The test identifies large relative error
assert !isCloseTo.test(10, 9) : "Difference 1 should not be <= 0.01 of 10";
assert !isCloseTo.test(9, 10) : "Difference 1 should not be <= 0.01 of 9";
```

Note that the predicates can be constructed using an absolute error tolerance which is combined with the relative equality test using an **Or** operator:

```
double relativeError = 0.01;
double absoluteError = 1;
DoubleDoubleBiPredicate areClose = Predicates.doublesAreClose(relativeError,␣
↪absoluteError);

// This would fail using relative error.
// The test passes using absolute error.
assert areClose.test(10, 9) : "Difference 1 should be <= 1";
assert areClose.test(9, 10) : "Difference 1 should be <= 1";
```

### 6.3.6 Test Framework Support

Testing relative equality within a test framework is simplified using predicates. For example a test for floating-point relative equality in `JUnit 5` must adapt the test for absolute difference:

```
double relativeError = 0.01;
double expected = 100;
double actual = 99;

// equal within relative error of expected
Assertions.assertEquals(expected, actual, Math.abs(expected) * relativeError);
```

This can be replaced with:

```
double relativeError = 0.01;
double expected = 100;
double actual = 99;
```

(continues on next page)

(continued from previous page)

```
// equal within relative error of expected
DoubleDoubleBiPredicate isCloseTo = Predicates.
↪doublesIsRelativelyCloseTo(relativeError);
Assertions.assertTrue(isCloseTo.test(expected, actual));
```

This will identify errors but the error message is not helpful.

In order to provide useful error messages for a `true/false` predicate the GDSC Test library contains a helper class for performing assertions that will raise an `AssertionError` if the test is `false`. The `TestAssertions` class is based on the `Assertions` design ideas of `JUnit 5`. It provides static assertion methods for pairs of all primitive types using any bi-valued test predicate to compare the two matched values. Arrays and nested arrays are supported using recursion.

This allows the test for equality to be extended to arrays and nested arrays:

```
double relativeError = 0.01;
double expected = 100;
double actual = 99;

DoubleDoubleBiPredicate areClose = Predicates.
↪doublesAreRelativelyClose(relativeError);

TestAssertions.assertTest(expected, actual, areClose);

// primitive arrays
double[] expectedArray = new double[] { expected };
double[] actualArray = new double[] { actual };
TestAssertions.assertArrayTest(expectedArray, actualArray, areClose);

// nested primitive arrays of matched dimension
Object[] expectedNestedArray = new double[][][] {{{ expected }}};
Object[] actualNestedArray = new double[][][] {{{ actual }}};
TestAssertions.assertArrayTest(expectedNestedArray, actualNestedArray, areClose);
```

If the predicate test fails then the `TestAssertions` class will construct a message containing the values that failed. Additionally all the predicates provided by the GDSC Test library support a description that will be added to the `AssertionError` message.

CHAPTER 7

# Issues

Please fill bug report in https://github.com/aherbert/gdsc-test/issues.

## 7.1 Change Log

**Contents**



### 7.1.1 Version 1.0.1

Patch release of GDSC Test.

| Change | Description |
|---|---|
| Update | Update build tools to support JDK 17. |
| Update | Remove redundant files from the gdsc-test-api jar. |

### 7.1.2 Version 1.0

First working version of GDSC Test.

GDSC Test 1.0 contains the following modules:

- `gdsc-test-build-tools` (required only for building the source)
- `gdsc-test-generator`

- `gdsc-test-api`
- `gdsc-test-utils`
- `gdsc-test-junit5` (requires JUnit 5)
- `gdsc-test-rng` (requires Commons RNG)
- `gdsc-test-docs` (documentation aggregator)
- `gdsc-test-examples`

Requires Java 8.

CHAPTER 8

# Indices and tables

- genindex
- search

# Index