Native/Unit Test Development Guidelines

This page is obsolete. It is replaced by HotSpot Unit Tests.

The purpose of these guidelines is to establish a shared vision on what kind of native tests and how we want to develop them for Hotspot using GoogleTest. Hence these guidelines include style items as well as test approach items.

First section of this document describes properties of good tests which are common for almost all types of test regardless of language, framework, etc. Further sections provide recommendations to achieve those properties and other HotSpot and/or GoogleTest specific guidelines.

Good test properties

Lightness

Use the most lightweight type of tests.

In Hotspot, there are 3 different types of tests regarding their dependency on a JVM, each next level is slower than previous

- `TEST` : a test does not depend on a JVM
- `TEST_VM` : a test does depend on an initialized JVM, but are supposed not to break a JVM, i.e. leave it in a workable state.
- `TEST_OTHER_VM` : a test depends on a JVM and requires a freshly initialized JVM or leaves a JVM in non-workable state

Isolation

Tests have to be *isolated*: not to have visible side-effects, influences on other tests results.

Results of one test should not depend on test execution order, other tests, otherwise it is becoming almost impossible to find out why a test failed. Due to hotspot-specific, it is not so easy to get a full isolation, e.g. we share an initialized JVM between all `TEST_VM` tests, so if your test changes JVM's state too drastically and does not change it back, you had better consider `TEST_OTHER_VM`.

Atomicity and self-containment

Tests should be *atomic* and *self-contained* at the same time.

One test should check a particular part of a class, subsystem, functionality, etc. Then it is quite easy to determine what parts of a product are broken basing on test failures. On the other hand, a test should test that part more-or-less entirely, because when one sees a test `FooTest::bar`, they assume all aspects of `bar` from `Foo` are tested.

However, it is impossible to cover all aspects even of a method, not to mention a subsystem. In such cases, it is recommended to have several tests, one for each aspect of a thing under test. For example one test to tests how `Foo::bar` works if an argument is `null`, another test to test how it works if an argument is acceptable but `Foo` is not in the right state to accept it and so on. This helps not only to make tests atomic, self-contained but also makes test name self-descriptive (discussed in more details in #Test names).

Repeatability

Tests have to be *repeatable*.

Reproducibility is very crucial for a test. No one likes sporadic test failures, they are hard to investigate, fix and verify a fix.

In some cases, it is quite hard to write a 100% repeatable test, since besides a test there can be other moving parts, e.g. in case of `TEST_VM` there are several concurrently running threads. Despite this, we should try to make a test as reproducible as possible.

Informativeness

In case of a failure, a test should be as *informative* as possible.

Having more information about a test failure than just compared values can be very useful for failure troubleshooting, it can reduce or even completely eliminate debugging hours. This is even more important in case of not 100% reproducible failures.

Achieving this property, one can easily make a test too verbose, so it will be really hard to find useful information in the ocean of useless information. Hence they should not only think about how to provide good information(#Error messages), but also when to do it(#Uncluttered output).

Testing instead of visiting

Tests should *test*.

It is not enough just to "visit" some code, a test should check that code does that it has to do, compare return values with expected values, check that desired side effects are done, and undesired are not, and so on. In other words, a test should contain at least one GoogleTest assertion and do not rely on JVM asserts.

Generally speaking to write a good test, one should create a model of the system under tests, a model of possible bugs(or bugs which one wants to find) and design tests using those models.

Nearness

Prefer having checks inside test code.

Not only does having test logic outside, e.g. verification method, depending on asserts in product code contradict with several items above but also decreases test's readability and stability. It is much easier to understand that a test is testing when all testing logic is located inside a test or nearby in shared test libraries. As a rule of thumb, the closer a check to a test, the better.

Asserts

Several checks

Prefer `EXPECT` over `ASSERT` if possible.

This is related to #Informativeness property of tests, information for other checks can help to better localize a defect's root-cause. One should use `ASSERT` if it is impossible to continue test execution or if it does not make much sense. Later in the text, `EXPECT` forms will be used to refer to both `ASSERT/EXPECT`.

When it is possible to make several different checks, but impossible to continue test execution if at least one check fails, you can use `::testing::Test::HasNonfatalFailure()` function. The recommended way to express that is `ASSERT_FALSE(::testing::Test::HasNonfatalFailure())`. Besides making it clear why a test is aborted, it also allows you to provide more information about a failure.

First parameter is expected value

In all equality assertions, expected values should be passed as the first parameter.

This convention is adopted by GoogleTest, and there is a slight difference in how GoogleTest treats parameters, the most important one is null detection. Due to different reasons, null detection is enabled only for the first parameter, that is to said `EXPECT_EQ(NULL, object)` checks that object is null, while `EXPECT_EQ(object, NULL)` checks that object equals to `NULL`, GoogleTest is very strict regarding types of compared values so the latter will generates a compile-time error.

Floating-point comparison

Use floating-point special macros to compare float/double values.

Because of floating-point number representations and round-off errors, regular equality comparison will not return true in most cases. There are special `EXPECT_FLOAT_EQ/EXPECT_DOUBLE_EQ` assertions which check that the distance between compared values is not more than 4 ULPs, there is also `EXPECT_NEAR(v1, v2, eps)` which checks that the absolute value of the difference between `v1` and `v2` is not greater than `eps`.

C string comparison

Use string special macros for C strings comparisons.

`EXPECT_EQ` just compares pointers' values, which is hardly what one wants comparing C strings. GoogleTest provides `EXPECT_STREQ and EXPECT_STRNE` macros to compare C string contents. There are also case-insensitive versions `EXPECT_STRCASEEQ, EXPECT_STRCASENE`

Error messages

Provide informative, but not too verbose error messages.

All GoogleTest asserts print compared expressions and their values, so there is no need to have them in error messages. Asserts print only compared values, they do not print any of interim variables, e.g. `ASSERT_TRUE((val1 == val2 && isFail(foo(8)) || i == 18)` prints only one value. If you use some complex predicates, please consider `EXPECT_PRED*` or `EXPECT_FORMAT_PRED` assertions family, they check that a predicate returns true /success and print out all parameters values.

However in some cases, default information is not enough, a commonly used example is an assert inside a loop, GoogleTest will not print iteration values (unless it is an assert's parameter). Other demonstrative examples are printing error code and a corresponding error message; printing internal states which might have an impact on results. One should add this information to assert message using << operator.

Uncluttered output

Print information only if it is needed.

Too verbose tests which print all information even if they pass are very bad practice. They just pollute output, so it becomes harder to find useful information. In order not print information till it is really needed, one should consider saving it to a temporary buffer and pass to an assert. test_memset_with_concurrent_readers.cpp#l71 is a good example how to do that.

Failures propagation

Wrap a subroutine call into `EXPECT_NO_FATAL_FAILURE` macro to propagate failures.

`ASSERT` and `FAIL` abort only the current function, so if you have them in a subroutine, a test will not be aborted after the subroutine even if ASSERT or FAIL fails. You should call such subroutines in `ASSERT_NO_FATAL_FAILURE` macro to propagate fatal failures and abort a test. `(EXPECT|ASSERT)_NO_FATAL_FAILURE` can also be used to provide more information.

Due to obvious reasons, there are no `(EXPECT|ASSERT)_NO_NONFATAL_FAILURE` macros. However, if you need to check if a subroutine generated a nonfatal failure (failed an `EXPECT`), you can use `::testing::Test::HasNonfatalFailure` function, or `::testing::Test::HasFailure` function to check if a subroutine generated any failures, see #Several checks

Naming and Grouping

Test group names

Test group names should be in CamelCase, start and end with a letter. A test group should be named after tested class, functionality, subsystem, etc.

This naming scheme helps to find tests, filter them and simplifies test failure analysis. For example, class `Foo` - test group `Foo`, compiler logging subsystem - test group `CompilerLogging`, G1 GC — test group `G1GC`, and so forth.

Filename

A test file must have `test_` prefix and `.cpp` suffix.

Both are actually requirements from the current build system to recognize your tests.

File location

Test file location should reflect a location of the tested part of the product.

1. All unit tests for a class from `foo/bar/baz.cpp` should be placed `foo/bar/test_baz.cpp` in `hotspot/test/native/` directory. Having all tests for a class in one file is a common practice for unit tests, it helps to see all existing tests at once, share functions and/or resources without losing encapsulation.
2. For tests which test more than one class, directory hierarchy should be the same as product hierarchy, and file name should reflect the name of the tested subsystem/functionality. For example, if a sub-system under tests belongs to `gc/g1`, tests should be placed in `gc/g1` directory.

Please note that framework prepends directory name to a test group name. For example, if `TEST(foo, check_this)` and `TEST(bar, check_that)` are defined in `hotspot/test/native/gc/shared/test_foo.cpp` file, they will be reported as `gc/shared/foo::check_this` and `gc/shared/bar::check_that`.

Test names

Test names should be in small_snake_case, start and end with a letter. A test name should reflect that a test checks.

Such naming makes tests self-descriptive and helps a lot during the whole test life cycle. It is easy to do test planning, test inventory, to see what things are not tested, to review tests, to analyze test failures, to evolve a test, etc. For example `foo_return_0_if_name_is_null` is better than `foo_sanity` or `foo_basic` or just `foo`, `humongous_objects_can_not_be_moved_by_young_gc` is better than `ho_young_gc`.

Actually using underscore is against GoogleTest project convention, because it can lead to illegal identifiers, however, this is too strict. Restricting usage of underscore for test names only and prohibiting test name starts or ends with an underscore are enough to be safe.

Fixture classes

Fixture classes should be named after tested classes, subsystems, etc (follow #Test group names rule) and have `Test` suffix to prevent class name conflicts.

Friend classes

All test purpose friends should have either `Test` or `Testable` suffix.

It greatly simplifies understanding of friendship's purpose and allows statically check that private members are not exposed unexpectedly. Having `FooTest` as a friend of `Foo` without any comments will be understood as a necessary evil to get testability.

OS/CPU specific tests

Guard OS/CPU specific tests by `#ifdef` and have OS/CPU name in filename.

For the time being, we do not support separate directories for OS, CPU, OS-CPU specific tests, in case we will have lots of such tests, we will change directory layout and build system to support that in the same way it is done in hotspot.

Miscellaneous

Hotspot style

Abide the norms and rules accepted in Hotspot style guide.

Tests are a part of Hotspot, so everything (if applicable) we use for Hotspot, should be used for tests as well. Those guidelines cover test-specific things.

Code/test metrics

Coverage information and other code/test metrics are quite useful to decide what tests should be written, what tests should be improved and what can be removed.

For unit tests, widely used and well-known coverage metric is branch coverage, which provides good quality of tests with relatively easy test development process. For other levels of testing, branch coverage is not as good, and one should consider others metrics, e.g. transaction flow coverage, data flow coverage.

Access to non-public members

Use explicit friend class to get access to non-public members.

We do not use GoogleTest macro to declare friendship relation, because, from our point of view, it is less clear than an explicit declaration.

Declaring a test fixture class as a friend class of a tested test is the easiest and the clearest way to get access. However, it has some disadvantages, here is some of them:

- Each test has to be declared as a friend
- Subclasses do not inheritance friendship relation

In other words, it is harder to share code between tests. Hence if you want to share code or expect it to be useful in other tests, you should consider making members in a tested class protected and introduce a shared test-only class which expose those members via public functions, or even making members publicly accessible right away in a product class. If it is not an option to change members visibility, one can create a friend class which exposes members.

Death tests

You can not use death tests inside `TEST_OTHER_VM` and `TEST_VM_ASSERT*`.

We tried to make Hotspot-GoogleTest integration as transparent as possible, however, due to the current implementation of `TEST_OTHER_VM` and `TEST_VM_ASSERT*` tests, you cannot use death test functionality in them. These tests are implemented as GoogleTest death tests, and GoogleTest does not allow to have a death test inside another death test.

External flags

Passing external flags to a tested JVM is not supported.

The rationality of such design decision is to simplify both tests and a test framework and to avoid failures related to incompatible flags combination till there is a good solution for that. However there are cases when one wants to test a JVM with specific flags combination, `_JAVA_OPTIONS` environment variable can be used to do that. Flags from `_JAVA_OPTIONS` will be used in `TEST_VM`, `TEST_OTHER_VM` and `TEST_VM_ASSERT*` tests.

Test-specific flags

Passing flags to a tested JVM in TEST_OTHER_VM and TEST_VM_ASSERT* should be possible, but is not implemented yet.

Facility to pass test-specific flags is needed for system, regression or other types of tests which require a fully initialized JVM in some particular configuration, e.g. with Serial GC selected. There is no support for such tests now, however, there is a plan to add that in upcoming releases.

For now, if a test depends on flags values, it should have `if (!<flag>) { return }` guards in the very beginning and `@requires` comment similar to jtreg `@requires` directive right before test macros. test_g1IHOPControl.cpp#l74 is an example of this temporary workaround. It is important to follow that pattern as it allows us to easily find all such tests and update them as soon as there is an implementation of flag passing facility.

In long-term, we expect jtreg to support GoogleTest tests as first class citizens, that is to say, jtreg will parse `@requires` comments and filter out inapplicable tests.

Flag restoring

Restore changed flags.

It is quite common for tests to configure JVM in a certain way changing flags' values. GoogleTest provides two ways to set up environment before a test and restore it afterward: using either constructor and destructor or `SetUp` and `TearDown` functions. Both ways require to use a test fixture class, which sometimes is too wordy. The simpler facilities like `FLAG_GUARD` macro or `*FlagSetting` classes could be used in such cases to restore/set values. Caveats:

- Changing a flag's value could break the invariants between flags' values and hence could lead to unexpected/unsupported JVM state.
- `FLAG_SET_*` macros can change more than one flag (in order to maintain invariants) so it is hard to predict what flags will be changed and it makes restoring all changed flags a nontrivial task. Thus in case one uses `FLAG_SET_*` macros, they should use `TEST_OTHER_VM` test type.

GoogleTest documentation

In case you have any questions regarding GoogleTest itself, its asserts, test declaration macros, other macros, etc, please consult its documentation.

TODO

Although this document provides guidelines on the most important parts of test development using GTest, it still misses a few items:

- Examples, esp for Miscellaneous#Access to non-public members
- test types: purpose, drawbacks, limitation
    - TEST_VM
    - TEST_VM_F
    - TEST_OTHER_VM
    - TEST_VM_ASSERT
    - TEST_VM_ASSERT_MSG
- Miscellaneous
    - Test libraries
        - where to place
        - how to write
        - how to use
    - test your tests
        - how to run tests in random order
        - how to run only specific tests
        - how to run each test separately
        - * check that a test can find bugs it is supposed to by introducing them
    - mocks/stubs/dependency injection
    - setUp/tearDown
        - vs c-tor/d-tor
        - empty test to test them
    - internal (declared in .cpp) struct/classes