Writing JPF Tests

TracNav

- JPFWiki - Welcome Page

  Introduction...

  Installing JPF...

  User Guide...

  Developer Guide

  - Design
  - Choice Generator
  - Partial Order Reduction
  - Attributes
  - Listener

    MJI...

  - Bytecode Factory
  - Logging
  - Report
  - Embedded
  - JPF tests
  - JPF project layout
  - Create a JPF project
  - Coding Conventions
  - Hosting update site

  Projects...

- Summer Projects
- External Projects
- Change(B)log

  About...

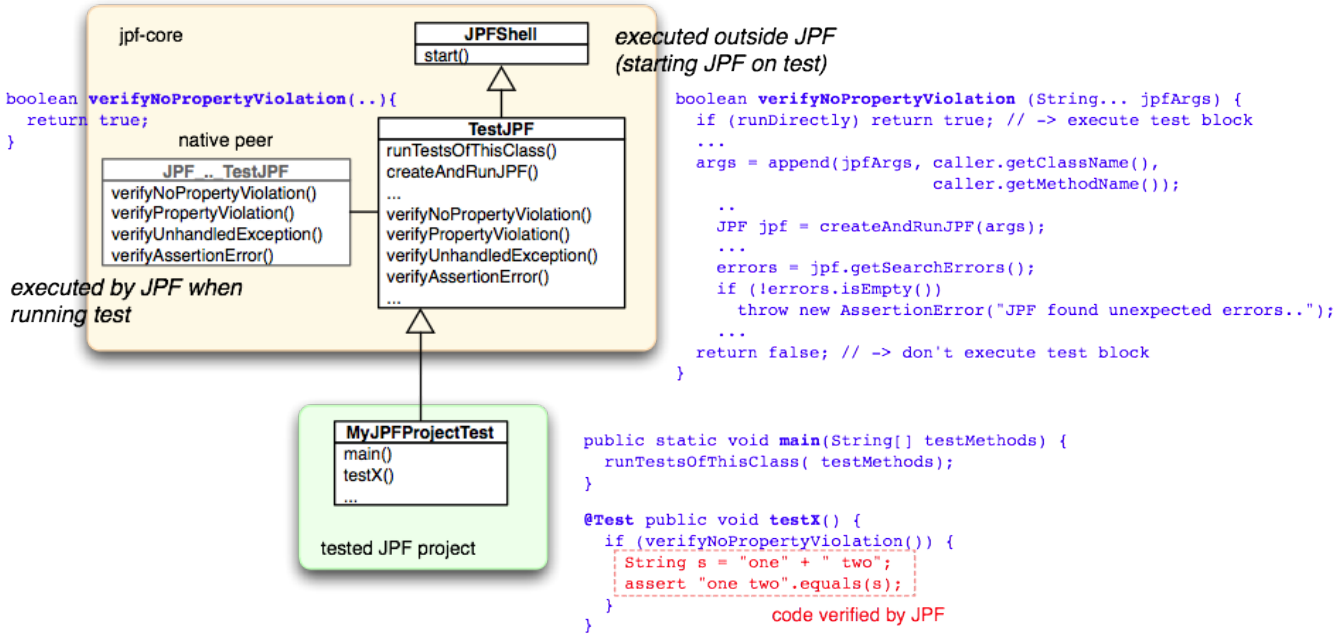
- Events
- Presentations
- Papers
- FAQ
- History?
- Support
- People?
- Playground
- Table of Context

As a complex runtime system for (almost) arbitrary Java programs, it goes without saying that JPF needs a lot of regression tests. You can find these under the `src/tests` directories in (hopefully) all JPF projects. All tests follow the same scheme, which is motivated by the need to run tests in a number of different ways:

1. as part of the Ant-based build system, i.e. from build.xml
2. as explicitly invoked JUnit tests
3. by directly running JPF on the test application (i.e. without JUnit, either as a JPF `shell` or via `RunTest.jar`)
4. by running the test application on a normal JVM

The rationale for this is to support various levels of inspection and debugging.

Each test conceptually consists of a test driver (e.g. executed under JUnit) which starts JPF from within its `@Test` annotated methods, and a class that is executed by JPF in order to check the verification goals. For convenience reasons, jpf-core provides infrastructure that enables you to implement both parts in the same class. This can be confusing at first - **the test class is used to start JPF on itself**.



The `main()` method of `TestJPF` derived classes always look the same and can be safely copied between tests:

```
public static void main(String[] testMethods){
 runTestsOfThisClass(testMethods);
}
```

This method serves two purposes. First, it is used to start the test outside JUnit, either on all `@Test` annotated instance methods, or just on the ones which names are provided as arguments. Second, it serves as the entry point for JPF when it executes the class. In this case, `TestJPF` takes care of invoking JPF on the test class and providing the name of the test method this was executed from.

Other than that, test classes just consist of (almost) normal `@Test` annotated JUnit test methods, which all share the same structure

```
import org.junit.Test;

@Test public void testX () {
 if (verifyNoPropertyViolation(JPF_ARGS){
   .. code to verify by JPF
 }
}
```

The trick is the call to `verifyNoPropertyViolation()`, or any of the other `verifyXX()` methods of `TestJPF`. If executed by the host VM, i.e. from JUnit, it starts JPF on the same class and the containing method, and returns `false`. This means the corresponding `if` block is **not** executed by the host VM.

When JPF is invoked, the argument to the main() method is set to the method name from which JPF got invoked, which causes `runTestsOfThisMethod()` to execute exactly this method again, but this time under JPF. Instead of re-executing the same `TestJPF.verifyX()` method again (and becoming infinitely recursive), we use a native peer `JPF_gov_nasa_jpf_util_test_TestJPF` which intercepts the `verifyX()` call and simply returns true, i.e. this time *only* the `if` block gets executed.

The rest of the host VM executed `TestJPF.verifyX()` checks the results of the JPF run, and accordingly throws an `AssertionError` in case it does not correspond to the expected result. The most common goals are

- **verifyNoPropertyViolation** - JPF is not supposed to find an error

- **verifyPropertyViolation** - JPF is supposed to find the specified property violation
- **verifyUnhandledException** - JPF is supposed to detect an unhandled excecption of the specified type
- **verifyAssertionError** - same for AssertionErrors
- **verifyDeadlock** - JPF is supposed to find a deadlock